Міністерство освіти і науки України

Івано-Франківський національний технічний університет нафти і газу
Інститут інформаційних технологій

Кафедра комп'ютерних систем і мереж

Савків Христина Андріївна

УДК 007

БАКАЛАВРСЬКА РОБОТА

Оцінювання мережевої безпеки комп'ютерних систем побудованих на основі парадигми LP

Комп'ютерна інженерія
(назва освітньої програми)

123 – Комп'ютерна інженерія
(шифр і назва спеціальності)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Здобувач освітнього ступеня _____ *Савків Х.А.*

Науковий керівник _____ *Заячук Я. І., к.т.н., доцент*

Допущено до захисту
Завідувач кафедри
д.т.н., професор /*С. І. Мельничук*/
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2025 рік

Івано-Франківський національний технічний університет нафти і газу

Інститут *Інформаційних технологій*

Кафедра *Комп'ютерних систем і мереж*

Освітньо-кваліфікаційний рівень *бакалавр*

Спеціальність *123 – Комп'ютерна інженерія*

ЗАТВЕРДЖУЮ:

Зав. кафедрою КСМ

С.І. Мельничук

«05» травня 2025 року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Савків Христині Андріївній

(прізвище, ім'я, по батькові)

1. Тема проекту (роботи*) Оцінювання мережевої безпеки комп'ютерних систем побудованих на основі парадигми LP*

керівник проекту (роботи) *Заячук Ярослав Іванович, доцент* .

затверджені наказом вищого навчального закладу від *05.05.2025 № 275/7*

2. Строк подання студентом проекту (роботи) *12 червня 2025р.*

3. Вихідні дані до роботи *Методичні вказівки, технічна література* .

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) *1. Аналіз предметної області та специфікації вимог до додатку інформаційного супроводу видавництва 2. Методи, моделі та алгоритми, формальна модель міркування 3.Реалізація власного рішення.*

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) .

6. Консультанти розділів роботи

7. Дата видачі завдання *29 січня 2025 р.* .

№ з/п	Назва етапів дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Примітка
1	*Збір інформації, вивчення літератури та пошук додаткової інформації*	*Лютий 2025р*	
2	*Аналіз предметної області та специфікації вимог до додатку інформаційного супроводу видавництва*	*Березень 2025р*	
3	*Методи, моделі та алгоритмиформальна модель міркування*	*Квітень 2025р*	
4	*Реалізація власного рішення*	*Травень 2025р*	
5	*Оформлення додатків, дипломної роботи*	*Червень 2025р*	

Студент _____ *Савків Х.А.*

Керівник роботи _____ *Заячук Я.І..*

АНОТАЦІЯ

Система NetScan1 являє собою передове інженерне рішення для автоматизованого аналізу безпеки комп’ютерних мереж, яке успішно вирішує проблему виявлення багатоступеневих атак і прогнозування потенційних загроз у складних мережевих середовищах. Її основна сила полягає в застосуванні логічного програмування на мові Datalog, що забезпечує чітку формалізацію мережевих атак і дозволяє створювати модульні, декларативні правила для моделювання сценаріїв компрометації. Використання системи XSB із табличним обчисленням гарантує високу продуктивність, дозволяючи обробляти мережі з тисячами хостів за секунди, що підтверджено експериментами на синтетичних мережах, де аналіз 2000 хостів займав 15.8 секунди. Модульна архітектура системи, яка включає сканер конфігурації, логічний рушій і інтерфейс користувача, забезпечує гнучкість, дозволяючи інтегрувати різноманітні джерела даних, такі як OVAL і NVD, через XSLT-перетворення, а також замінювати компоненти, наприклад, сканери, без зміни ядра.

ABSTRACT

The NetScan1 system is an advanced engineering solution for automated computer network security analysis, which successfully solves the problem of detecting multi-stage attacks and predicting potential threats in complex network environments. Its main strength lies in the use of logic programming in the Datalog language, which provides a clear formalization of network attacks and allows you to create modular, declarative rules for modeling compromise scenarios. The use of the XSB system with table calculations guarantees high performance, allowing you to process networks with thousands of hosts in seconds, which is confirmed by experiments on synthetic networks, where the analysis of 2000 hosts took 15.8 seconds. The modular architecture of the system, which includes a configuration scanner, a logic engine and a user interface, provides flexibility, allowing you to integrate various data sources, such as OVAL and NVD, through XSLT transformations, as well as replace components, such as scanners, without changing the core.

ЗМІСТ

ВСТУП

У сучасному світі інформаційні технології відіграють ключову роль у всіх сферах людської діяльності, а комп'ютерні системи стають основою для обробки, зберігання та передачі даних. Зі зростанням складності цих систем та їх інтеграції в мережеві середовища питання забезпечення мережевої безпеки набуває критичної важливості. Порушення безпеки можуть призвести до втрати даних, фінансових збитків та підриву довіри до інформаційних систем

Парадигма логічного програмування (LP), яка базується на формальних моделях міркування та декларативному підході, відкриває нові можливості для аналізу та управління вразливостями в мережевих системах. Вона дозволяє створювати точні моделі для виявлення атак, аналізу конфігурацій і специфікації політик безпеки, що робить її перспективною для розробки систем захисту. Однак складність сучасних мереж і швидке зростання кіберзагроз вимагають нових підходів до аналізу безпеки, які враховують як теоретичні, так і практичні аспекти.

Актуальність теми зумовлена стрімким розвитком кібератак, які експлуатують вразливості програмного забезпечення, та обмеженнями традиційних методів аналізу безпеки, що часто не справляються з динамічною природою розподілених систем. Існуючі рішення, такі як інструменти сканування вразливостей або системи виявлення вторгнень, нерідко мають проблеми з точністю, масштабованістю або адаптацією до специфічних мережевих топологій, що підкреслює потребу в розробці нових підходів на основі парадигми LP.

Метою роботи є аналіз мережевої безпеки комп’ютерних систем, побудованих на основі парадигми логічного програмування, для розробки ефективного рішення для виявлення та управління вразливостями.

Завданнями дослідження є аналіз сучасних рішень для забезпечення мережевої безпеки, розробка формальної моделі міркування для аналізу вразливостей, створення правил взаємодії та специфікації політики безпеки, а

також реалізація і тестування власного рішення для оцінки ефективності запропонованого підходу.

Об’єктом дослідження є процеси аналізу та забезпечення мережевої безпеки в комп’ютерних системах, що охоплюють виявлення вразливостей, моделювання атак, конфігурацію хостів і оцінку безпеки.

Предметом дослідження є методи, моделі та алгоритми, що застосовуються для аналізу мережевої безпеки на основі парадигми LP, зокрема формальні моделі міркування, правила взаємодії, специфікації вразливостей, а також інструменти для аналізу журналів даних і топології мережі.

Для досягнення мети використано методи аналізу літературних джерел, формального моделювання, порівняльного аналізу рішень, емпіричного програмування, тестування гіпотетичних сценаріїв атак, а також оцінки ефективності системи.

Розроблене рішення передбачає створення архітектури системи для аналізу вразливостей, використання формальних моделей для виявлення слідів атак, адаптивну конфігурацію хостів і специфікацію політик безпеки, що підвищує точність і ефективність захисту мереж. Очікується, що впровадження запропонованого рішення сприятиме підвищенню безпеки комп’ютерних систем, спрощенню управління вразливостями та встановленню нових стандартів для аналізу безпеки в розподілених мережах.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА СПЕЦИФІКАЦІЇ ВИМОГ ДО ДОДАТКУ ІНФОРМАЦІЙНОГО СУПРОВОДУ ВИДАВНИЦТВА

1.1 Уразливості програмного забезпечення та керування мережевою безпекою

Робота з уразливістю програмного забезпечення на мережевих хостах створює проблему для адміністрування мережі. За останні 15 років у програмному забезпеченні (та інформаційних системах загалом) було виявлено дедалі більше вразливостей безпеки. Згідно зі статистикою, опублікованою CERT/CC, центральною організацією для звітування про інциденти безпеки, кількість повідомлених вразливостей значно зросла за останні п’ять років (рис. 1.1). Очікується, що швидкість, з якою з’являться нові вразливі можливості програмного забезпечення, продовжить збільшуватися в осяжному майбутньому. З огляду на тисячі нових уразливостей, які виявляються щороку, підтримувати 100% рівень виправлення є недопустимим, а іноді й небажаним для більшості організацій. Хоча в багатьох випадках виправлення приходять відразу після звітів про вразливості, люди не завжди застосовують виправлення відразу з різних причин [3]. Поспішно написані патчі є нестабільними та можуть навіть викликати більше помилок. Виправлення ядра операційної системи часто вимагає перезавантаження, що впливає на доступність таким чином, що може бути непомірно дорогим для деяких організацій.

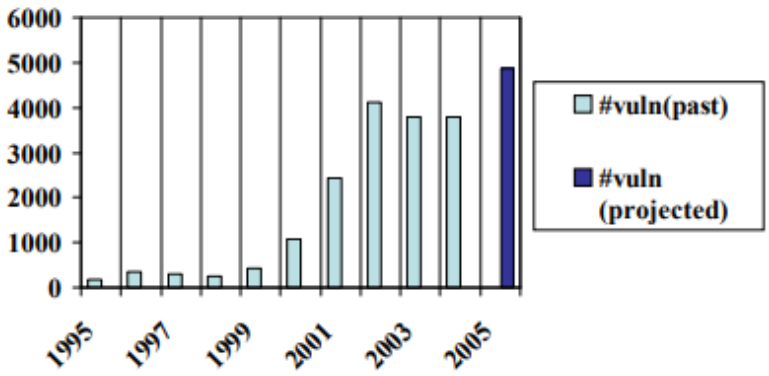

Рисунок 1.1– Кількість вразливостей, про які повідомляє CERT

Таким чином, нерідкі випадки, коли мережевий адміністратор продовжує запускати програмне забезпечення з помилками протягом певного періоду часу після повідомлення про помилку. У рамках дисциплінованої корпоративної програми управління ризиками менеджери з безпеки повинні приймати рішення про те, які інформаційні системи є найбільш критичними, і визначати пріоритетність контрзаходів безпеки для таких систем. Вони повинні переконатися, що будь-яке потенційне використання невиправлених помилок не відбудеться, або навіть якщо це станеться, це не завдасть шкоди. Однією із щоденних обов'язків адміністраторів є читання звітів про вразливості з різних джерел і визначення того, які саме вразливості можуть поставити під загрозу безпеку керованої ними мережі. Деякі помилки можуть не використовуватися в налаштуваннях локальної мережі. Навіть якщо ними можна скористатися, доступ, отриманий зловмисником, може бути не більшим, ніж той, який йому вже дозволено.

Наприклад, у мережі на рисунку 1.2 можуть існувати вразливості на веб-сервері машини. Але якщо помилку на веб-сервері можна використовувати лише локально[1]і всі користувачі з обліковими записами на веб-сервері є надійними, немає безпосередньої небезпеки експлуатувати.

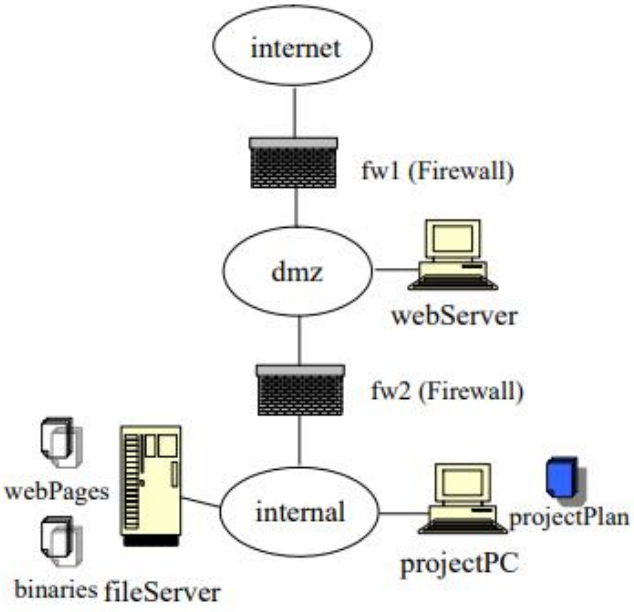

Рисунок 1.2 – Приклад мережі

Якщо помилку можна використовувати віддалено[2] але брандмауер fw1 блокує трафік до вразливого порту, машина все ще в безпеці. Якщо брандмауер дозволяє доступ до вразливого порту (можливо, для звичайного доступу до веб-сервера), але наслідком потенційного експлойту є лише те, що зловмисник може читати веб-сторінки, це також безпечно, оскільки дані все одно мають бути загальнодоступними.

У зв'язку з появою нових вразливостей оцінка їхнього впливу на безпеку мережевої інфраструктури є важливою для вибору правильних заходів протидії: виправлення та перезавантаження, зміна конфігурації брандмауера, демонтування розділу файлового сервера тощо. На жаль, спосіб зламу мережі не завжди очевидний. Для прикладу мережі на рисунку 1.2, якщо одного дня буде повідомлено про нову вразливість програми веб-сервісу на веб-сервері, це не буде неминучою загрозою для конфіденційних даних projectPlan, що зберігаються на робочій станції. Однак в залежності від комплектації двох брандмауерів (fw1 і fw2), конфігурації файлового сервера та конфігурації робочої станції це може бути не так.

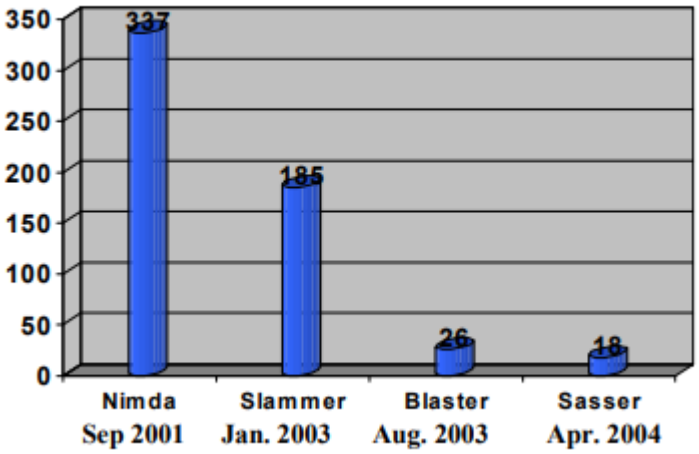

Рисунок 1.3 – Вікно вразливості до експлойту (у днях)

Наприклад, багато корпорацій використовують спільний доступ до файлів NFS для монтування розділів файлової системи на файлових серверах. NFS є незахищеним протоколом і приймає довірчі відносини на основі хоста. Якщо

комп'ютер клієнта скомпрометовано, зловмисник потенційно може отримати доступ до всіх файлів, які експортуються до клієнта. Таким чином, якщо зловмисник з Інтернету може спочатку скомпрометувати веб-сервер, використовуючи вразливість, він потенційно може змінити файли, що зберігаються на файловому сервері. Якщо спільні виконувані файли зберігаються в розділі, експортованому на веб-сервер, цілісність виконуваних файлів буде порушена — зловмисник може встановити троянську програму. Якщо той самий розділ також змонтовано робочою станцією, користувач на цій машині може запустити програму троянського коня, таким чином надаючи зловмисникові доступ до робочої станції. У результаті конфіденційні дані projectPlan потенційно можуть бути передані зовнішньому зловмиснику.

Щоб виявити ці потенційні шляхи атаки в мережі, потрібно не лише перевірити параметри конфігурації кожного елемента мережі — машин, брандмауерів,маршрутизатори тощо, але також враховуйте всі можливі взаємодії між ними. Проведення цього багатохостового, багатоетапного аналізу вразливості людьми є схильним до помилок і трудомістким. Автоматизація цього процесу оцінювання є важливою, враховуючи той факт, що вікно між часом, коли повідомляється про вразливість, і часом, коли вона використовується у великих масштабах, суттєво зменшилося [3] (рис. 1.3). Захисники мереж і систем тепер можуть планувати, що у них буде лише кілька днів, щоб застосувати контрзаходи для захисту вразливих систем і сервісів, підключених до публічних мереж. Щоб погіршити ситуацію, мережі, які використовуються в організаціях, стають більшими та складнішими. На жаль, сучасні технології досі не змогли забезпечити адекватних методологій для досягнення автоматичного керування мережевою безпекою. У результаті управління конфігурацією мережі в сучасному світі все ще значною мірою залежить від людського досвіду. Відповідно до опитування, проведеного Асоціацією індустрії обчислювальних технологій, серед усіх порушень безпеки, про які повідомили 900 опитаних організацій у 2004 році, 84% з них були спричинені людськими помилками. Експоненціальне зростання інцидентів безпеки, про які повідомляється CERT (рис. 1.4), показує, що існує

нагальна потреба в ефективній методології для автоматизації управління мережевою безпекою.

1.2 Попередні роботи з аналізу вразливостей

Автоматичний аналіз вразливості може датуватися Куангом [4] і COPS [17]. Куанг формалізує семантику безпеки UNIX як набір правил і проводить пошук способів зламу системи на основі цих правил. COPS — це засіб перевірки безпеки UNIX, який містить набір правил Kuang. NetKuang [54] розширив набір правил у Kuang для захоплення конфігураційної інформації, яка впливає на безпеку в мережі, наприклад файлу .rhosts, і, отже, здатний міркувати про неправильну конфігурацію пайки в мережі машин UNIX. У той час, коли були розроблені Kuang і NetKuang, уразливості програмного забезпечення не стали серйозною проблемою для безпеки мережі, а масштаби мережевих атак були набагато меншими, ніж сьогодні.

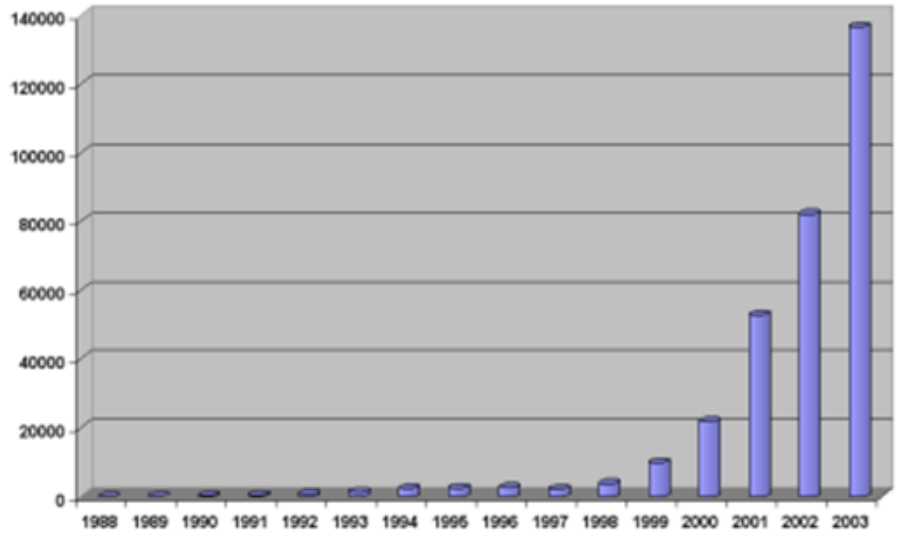

Рисунок 1.4 – Інциденти безпеки, про які повідомляється в CERT

Правила Kuang і NetKuang обмежені кількома сценаріями атак і жорстко закодовані в реалізації. Немає включення сторонніх знань безпеки, таких як поради щодо вразливості. Такий поетапний підхід більше не може задовольнити потреби безпеки для загроз, з якими стикаються сьогодні комп'ютерні мережі.

Щоб інструмент аналізу безпеки був життєздатним із мінливими загрозами, логіка аргументації має бути формально визначена та відокремлена від реалізації. Офіційна специфікація повинна мати можливість включати інформацію від сторонніх агентств, які надають визначення вразливостей програмного забезпечення. Аргументація має бути обґрунтованою в теорії та ефективною на практиці.

Левітт і Темплтон запропонували модель вимог і умов для комп'ютера на галсах [48], яка, по суті, визначає перед- і післяумову кожного кроку атаки.

Це дозволяє поєднувати кілька кроків атаки таким чином, щоб попередні кроки створювали необхідні умови для успіху наступних, що призводило до виявлення шляхів атаки, які не були очевидними, розглядаючи кожен компонент окремо. Модель Левітта має чітку семантику для атак і є набагато гнучкішою, ніж моделі на основі сигнатур. Ця ідея була використана в різних роботах з аналізу вразливостей. З точки зору конкретних механізмів моделювання та аналізу, було запропоновано два підходи: перевірка моделі та пошук у графі залежностей від використання.

Використання перевірки моделі в аналізі вразливості мережі вперше було запропоновано Річі та Амманом [43]. У підході перевірки моделі мережа моделюється як система переходу між станами. Інформація про конфігурацію кодується як змінні стану. Крок атаки моделюється як перехідне відношення між двома станами. Відношення переходу задається у вигляді (S_1, C_2), де C_1 – значення булевих змінних, що характеризують передумови атаки, а S_2 представляє постумову атаки. Шлях атаки проявляється як послідовність дійсних переходів між станами з початкового стану, що веде до стану, де властивість безпеки мережі порушена. Засіб перевірки моделі може перевіряти модель на часову формулу, яка може виражати такі властивості, як «усі стани, досяжні з S_0 задовольнить дану властивість безпеки", де S_0 – відомий початковий стан мережі. Якщо формула задовольняє модель, жодні шляхи атаки не можуть призвести до поганої ситуації. Якщо формула не задовольняє модель, засіб перевірки моделі може вивести послідовність переходів станів, яка закінчується в стані, в якому властивість

безпеки не зберігається. Цей контрприклад трасування показує шлях атаки, який призводить до порушення властивості безпеки.

Перевага підходу перевірки моделі полягає в тому, що можна використати потужність міркування готових засобів перевірки моделей, а не писати налаштований механізм аналізу. Однак потрібно бути обережним, щоб уникнути комбінаторного вибуху, який часто виникає під час перевірки моделі. У розробці програмного забезпечення люди запропонували різніпрагне зробити перевірку моделі швидкою при перевірці властивостей безпеки великих програмних систем [21, 1, 53]. Однак не було жодної роботи, яка б показувала, що методи, які можуть прискорити перевірку моделі під час перевірки програмного забезпечення, також можуть прискорити аналіз безпеки мережі. Єдині експериментальні дані, які ми можемо знайти, які показують продуктивність і масштабованість використання перевірки моделі для аналізу вразливості мережі, знаходяться в роботі Шейнера та ін. [46]. У статті описано експериментальне налаштування, яке складається з трьох машин, маршрутизатора та брандмауера. Кількість атомарних атак у моделі становить чотири. Час роботи інструменту в цьому прикладі становить близько 5 секунд. Коли приклад розширено двома додатковими хостами, чотирма додатковими атомарними атаками, кількома новими вразливими місцями та гнучкими конфігураціями брандмауера, інструменту знадобилося 2 години, щоб знайти всі шляхи атак, з яких 5 хвилин витрачається на перевірку моделі, а решта часу витрачається на створення графіка атак. Цей результат не дав переконливих доказів того, що перевірка моделі добре підходить для аналізу безпеки мережі. На даний момент залишається сумнівним, чи працюватиме такий підхід для великих мереж із тисячами хостів.

Перевірка моделі призначена для вивчення багатих часових властивостей системи переходу між станами. Хоча така експресивна здатність має вирішальне значення для перевірки властивостей програмного забезпечення та паралельних систем, незрозуміло, чи повна потужність аргументації корисна для аналізу безпеки мережі. Однією з проблем використання стандартного засобу перевірки моделі як механізму аналізу є те, що більшість послідовностей переходів станів у

моделі фактично не потрібно перевіряти для цілей аналізу безпеки мережі. Для мережевих атак можна припустити властивість монотонності, за якої перевірка може бути значно прискорена.

Монотонність Властивість монотонності стверджує, що отримання більших привілеїв може лише допомогти зловмиснику у подальшому скомпрометуванні системи.

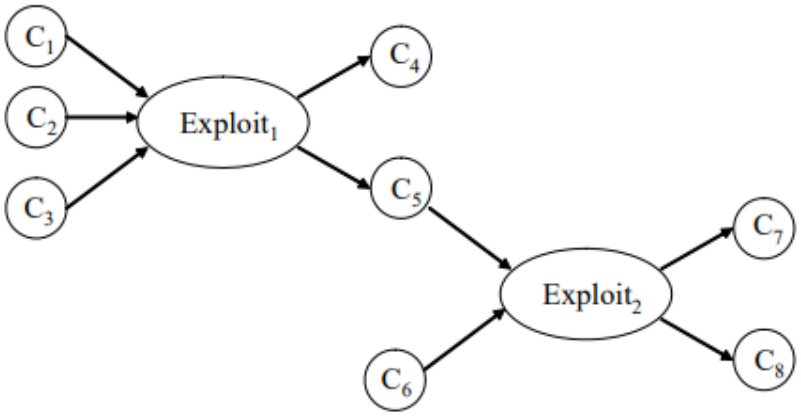

Рисунок 1.5 – Граф залежностей експлойтів

Наприклад, якщо є це два веб-сервери, які можуть бути скомпрометовані зловмисником, атака на один із них зазвичай не впливає на його здатність атакувати інший [3]. Таким чином, як тільки аналіз виводить, що зловмисник може отримати певні привілеї, цей факт може залишатися вірним протягом решти процесу аналізу. Немає необхідності повертатися назад. Однак у стандартній перевірці моделі необхідно перевірити всі можливі шляхи — ті, у яких факт є істинним, і ті, що ні. Коли ми маємо справу з великими мережами, на кожному кроці буде велика кількість варіантів переходу між станами, і це відстеження призведе до втрати значної кількості обчислювальної потужності. У гіршому випадку це може призвести до експоненціального зростання. Часткове зменшення порядку [35, 19] може підвищити цю проблему в системах програмного забезпечення для перевірки моделі. Однак не було показано, як застосувати цю техніку для перевірки моделі безпеки мережі. На основі властивості монотонності Ammann запропонував підхід, коли залежності між експлойтами моделюються в

структурі графа, а аналіз атак стає проблемою пошуку графа [2]. На рисунку 1.5 показано частину графіка залежностей експлойтів. Вузол на графіку є або умовою, або експлойтом. Умова є логічна змінна, що представляє певний стан системи, наприклад, чи встановлено певну версію програмного забезпечення на машині. Експлойт може статися, якщо всі його передумови вірні. Якщо умова C_iє передумовою експлойта e, буде перехід від вузла, що представляє C_i до вузла e. Після виконання експлойту стан мережевої системи зміниться. У монотонній системі зміна стану лише призводить до того, що більше умов є істинними. Ці умови є постумовами експлойту, і від експлойту буде перевага до кожної з його постумов. Оскільки кількість умов і експлойтів пропорційна розміру мережі, розмір графіка також пропорційний розміру мережі. Алгоритм пошуку можна розглядати як процес маркування графа, де позначений вузол умови є істинним, а непозначений — хибним. Експлойт-вузол можна позначити, якщо позначено всі його попередники (передумови). Тоді всі його наступники (постумови) також будуть позначені, якщо їх не було. Після того як вузол позначено, він залишатиметься позначеним назавжди. Алгоритм припиняє роботу, якщо більше не можна позначити вузли. Оскільки кожен вузол і ребро буде відвідано лише один раз, час виконання поліноміальний за розміром графа.

Цей алгоритм на основі графіків, заснований на припущенні монотонності, дозволяє уникнути потенційного експоненціального вибуху під час перевірки моделі. Однак алгоритм жорстко закодований як програмний код, і немає чіткої специфікації властивостей, що перевіряються, і взаємодії в мережі. Робота, описана в цій роботі, передбачає таку саму властивість монотонності, але використовує підхід, заснований на логіці, який формально визначає кожен відповідний елемент у міркуванні та їх взаємодії. У результаті він може поєднувати різноманітну інформацію та інструменти разом, створюючи наскрізну автоматичну систему.

Як і механізми аналізу, також існує два підходи до представлення графіків атак. В одному з них кожна вершина на графі представляє стан усієї мережевої системи, а ребра представляють кроки атаки, які змушують мережу переходити з

одного стану в інший. Ми називаємо це графіком атаки стану мережі, і він відповідає аналізу на основі перевірки моделі. Інший підхід відповідає алгоритму пошуку графів на основі властивості монотонності, де граф атаки по суті є частиною графа залежності від експлойту, яка сприяє атаці.

Шейнер та ін. детально досліджував автоматичну генерацію та аналіз графіків атак стану мережі на основі перевірки символьної моделі [46]. Філліпс і Свілер також вивчали аналіз уразливості мережі на основі графіків атак мережі [38], хоча вони не використовували методи перевірки моделі, а розробили налаштований інструмент для створення графів атак [47]. Графи мережевих атак страждають від експоненціального вибуху. У роботі Шейнера автори повідомляють, що час роботи їх інструменту зростає з 5 секунд до 2 годин, коли розмір мережі зростає з 3 хостів до 6 хостів (з іншими параметрами також зростає пропорційно). [4]. Потенційний простір станів зростає з 2^{91} до 2^{229}, а простір досяжних станів зростає від 101 до 6190. У роботі Свілера та ін. [47] автори також обговорювали питання вибуху графа та запропонували декілька методів полегшення, але експериментальних результатів не було надано. З іншого боку, графіки атак на основі залежності від експлойту є поліноміальними, оскільки окремі умови, а не цілі стани мережі, представлені як вузли. Хоча існує лише поліноміальна кількість умов, кількість усіх можливих станів є експоненціальною.

Проблема з графіками атак стану мережі полягає в тому, що вони не використовують властивість монотонності. Оскільки запуск однієї атаки не зменшує здатність зловмисника здійснити іншу, порядок, у якому виконуються незалежні кроки атаки, не має значення. Але цей порядок є чітким у графіках атак із станом мережі, що призводить до експоненціальної кількості надлишкових шляхів атаки, які відрізняються лише порядком кроків атаки. Метод, запропонований Swiler, et al. [47], щоб усунути ці надлишкові шляхи атаки, насправді є неявним використанням графа залежності від експлойту шляхом дотримання загального порядку щодо умов мережі.

1.3 Мова специфікації

Важливим кроком в аналізі безпеки мережі є визначення мережевих елементів і способу їх взаємодії в машиночитаному форматі. Тоді механізм автоматичного аналізу може обчислити можливі шляхи атаки на основі специфікації. Чітка специфікація має вирішальне значення для створення життєздатного інструменту аналізу. Безпека – це проблема, яка стосується кожного аспекту системи. В атаці може бути використано як навмисну, так і ненавмисну поведінку компонентів системи. Будь-яка система, яка жорстко кодує знання безпеки в реалізації, приречена на невдачу перед обличчям постійно зростаючих загроз. Враховуючи швидкість, з якою повідомляють про нові вразливості, автоматичний інструмент повинен мати можливість приймати як вхідні дані формальну специфікацію помилок безпеки. Чітка специфікація логіки аналізу полегшує інтеграцію таких експертних знань із незалежних джерел, таких як CERT, CVE та інші агентства, що повідомляють про помилки. Методології атак розвиваються разом із винайденням нових технологій, які забезпечують більш складну взаємодію між елементами мережевої системи. Будь-який інструмент аналізу безпеки не в змозі зафіксувати всі ці взаємодії. Вказівка цих взаємодій у формальній декларативній мові полегшує розуміння того, що можна, а що не можна обробляти інструментом, і вдосконалювати інструмент, коли це необхідно. Процес аналізу також повинен знати багатопараметри конфігурації кожної машини в мережі, а також маршрутизаторів, брандмауерів і комутаторів. Нещодавно були розроблені різні інструменти сканування, які можуть надати цю конфігураційну інформацію [52, 6, 7]. Чітка специфікація логіки аналізу дає змогу виділяти різноманітну конфігураційну інформацію та використовувати відповідні інструменти для її збору, замість того, щоб заново винаходити велосипед.

Чіткості специфікації раніше не приділялося належного значення. У підході до перевірки моделі стан мережі моделюється як набір логічних змінних, кожна з яких представляє певний стан мережі. Взаємодії безпеки визначені як відносини переходу стану. Хоча це кодування можна зробити модульним і розширюваним,

його штучність ускладнює його розуміння для людей. У графі залежності від експлойтів умови мережі закодовані як мітки на графіку. Взаємодії безпеки кодуються як ребра графа. Цьому кодуванню також бракує рівня ясності, який забезпечує формальна мова специфікації. Tidwell та ін. запропонував мову для моделювання інтернет-атак [49]. Однак мова надто складна, і незрозуміло, наскільки легко використовувати сторонні відомості про безпеку або дані сканера цією мовою.

Робота, описана в цій роботі, вирішує проблему шляхом прийняття логічного підходу. Взаємодії між елементами мережі формально специфікуються в мові логічного програмування Datalog [11]. Datalog є синтаксичною підмножиною Prolog, тому специфікація також є програмою, яку можна завантажити в стандартне середовище Prolog і виконати. Datalog має чітку декларативну семантику та монотонну логіку, що робить його особливо придатним для аналізу мережевих атак. Datalog популярний у дедуктивних базах даних, і кілька десятиліть роботи над розробкою механізмів міркування для баз даних дали інструменти, які можуть ефективно оцінювати Datalog [41, 51]. Використання цих механізмів оцінки дозволяє аналізувати великі корпоративні мережі з тисячами машин. Більш глибокою причиною прийняття логічного підходу єщо він враховує людські міркування, а це саме те, що сьогодні повинен робити системний адміністратор, керуючи безпекою мереж. Систему міркування, описану в цій роботі, можна розглядати як експертну систему, яка полегшує тягар міркувань про великі та складні системи з людей, чий мозковий потенціал не встигає за масштабом завдання.

2 МЕТОДИ, МОДЕЛІ ТА АЛГОРИТМИФОРМАЛЬНА МОДЕЛЬ МІРКУВАННЯ

2.1 Перегляд журналу даних

Проектована система використовує Datalog [11] як мову для моделювання елементів мережі та їх взаємодії. Спочатку ми розглянемо деякі терміни.

Синтаксично Datalog є підмножиною Prolog [12] з обмеженими формами речень. Літерал, $p(t_1, \ldots, t_k)$ є предикатом, застосованим до його аргументів, кожен з яких є константою або змінною. У формалізмі Прологу змінна - це ідентифікатор, який починається з великої літери. Константа - це така, яка починається з малої літери. Нехай $Л_0, Л_1, \ldots Л_п$ бути літералами, пропозиція Horn у Datalog має вигляд:

$$Л_0 :- Л_1, \ldots, Л_п$$

Семантично це означає, якщо $Л_1, \ldots, Л_п$вірні тоді $Л_0$ також вірно. Ліва сторона називається головою, а права – тілом. Речення з порожнім тілом називається фактом. Речення з непорожнім тілом називається правилом. Істотна відмінність між Datalog і Prolog полягає в тому, що Datalog має чисте оголошення. Порядок речень у програмі Datalog не має значення для її логічного значення та результату оцінки. Тоді як у Prolog такий порядок важливий і впливає на результат оцінювання [12] через стратегію пошуку в глибину та оператори побічного ефекту, такі як «cut».

Datalog часто використовується в дедуктивних базах даних. У таких налаштуваннях кортежі даних у базі даних представлені як факти Datalog, а дедуктивний механізм реалізовано як програма Datalog, яка працює на вхідних даних із бази даних. Факти Datalog, що представляють вихідну базу даних, називаються розширеною базою даних (EDB), а факти Datalog, обчислені дедуктивним механізмом, називаються інтенсіональною базою даних (IDB). Складність обчислення того, чи мається на увазі літерал програмою Datalog із

вхідних даних EDB (тобто чи є літерал в IDB), поліноміальна за розміром EDB [14]. У цій роботі ми називаємо предикат EDB примітивним предикатом, а предикат IDB — похідним предикатом.

Datalog також використовувався як мова безпеки для вираження політик контролю доступу [15, 31]. Декларативна семантика Datalog робить визначення таких понять, як делегування, простим. Ефективність Datalog і існуючих готових механізмів оцінки Datalog [41, 51] роблять такі мови легко використовуваними на практиці.

Є багато переваг використання Datalog як формальної моделі аргументації в аналізі безпеки, який обговорюється в цій роботі. У порівнянні з графом залежностей експлойтів, Datalog є формальною декларативною логічною мовою, яка забезпечує чітку специфікацію. Як і в підході перевірки моделі, для проведення аналізу можна використовувати готовий логічний механізм. Але на відміну від перевірки моделі, час виконання програми Datalog поліноміальний за розміром вхідних даних. Логічні механізми були оптимізовані протягом десятиліть для ефективної обробки великих наборів даних, що робить Datalog особливо придатним для аналізу безпеки великих і складних мереж.

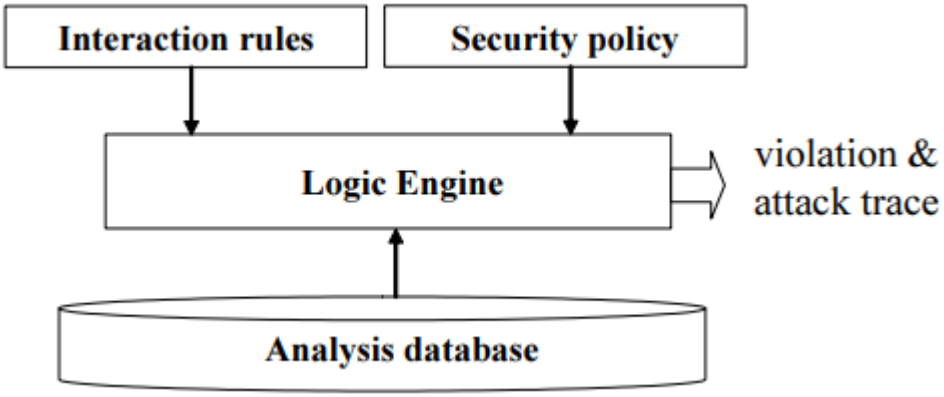

Рисунок 2.1 – Структура аналізу

2.2 Структура аналізу порушення та слід атаки

Структура аналізу ядра проектованої системи показана на рисунку 2.1.

База даних аналізу — це набір фактів Datalog, які представляють стан мережі та консультативну інформацію про вразливості програмного забезпечення. Розділ 3 детально обговорить, як заповнити цю базу даних. Правила взаємодії — це пропозиції Datalog, які визначають, як різні частини мережі можуть взаємодіяти та впливати на безпеку. Це правила міркування, які можуть імітувати те, що зловмисник може робити в мережі, враховуючи інформацію про конфігурацію в базі даних аналізу. Політика безпеки визначає кінцеву властивість, яку системний адміністратор хоче зберегти для мережі. У проектованій системі політика — цепрості кортежі Datalog, які перераховують законні доступи до даних принципалами.

2.3 Правила взаємодії

Правила взаємодії проектованої системи визначають семантику: різних типів уразливостей та їх експлойтів, звичайної поведінки програмного забезпечення, що впливає на безпеку, та багатопрохідного доступу до мережі. Багато з цих правил залежать від операційної системи. Правила, розглянуті в цій роботі, застосовуються до операційних систем сімейства Unix. В даний часв проектованій системі близько 20 правил. Правила проектованої системи ретельно розроблені таким чином, щоб інформація про конкретні вразливості була виключена. Правила взаємодії характеризують загальні методології атак (наприклад, «дистанційне використання помилки переповнення буфера» або «клієнтська програма троянського коня»), а не конкретні вразливості. Таким чином, правила не потрібно часто змінювати, навіть якщо часто повідомляють про нові вразливості. Правила також не залежать від конкретних конфігурацій конкретного налаштування мережі, тому їх можна застосовувати на різних сайтах.

Типи констант

У Datalog термін є константою або змінною. Datalog є нетипізованою мовою, тому предикат можна застосовувати до довільних термінів. Однак, щоб

зробити речення Datalog значущим, аргументи предиката повинні приймати значення з певних доменів. У цьому розділі наведено типи, які використовуються в правилах взаємодії Datalog.

1. Ведучий. У цій роботі хост представлено як символічне ім’я, таке як webServer і fileServer. У реальній реалізації він представлений як діапазон IP-адрес.

2. Протокол. Протокол транспортного або прикладного рівня, наприклад tcp, udp і rpc.

3. Порт. Число, що розрізняє різні служби в межах одного протоколу.

4. Директор. Символічне ім’я, що представляє певну групу людей, наприклад, співробітника та зловмисника.

5. Дані. Символічне ім’я, що представляє абстрактне поняття елемента даних, наприклад веб-сторінки та проектПлан.

6. Рядок. Рядки в одинарних лапках використовуються для представлення шляхів файлової системи, ідентифікаційних номерів уразливості тощо.

7. Діапазон експлуатації. Або localExploit, або remoteExploit.

8. Наслідки експлуатації. Одна з чотирьох можливостей: конфіденційність, цілісність, ескалація привілеїв і дос.

9. Програма.Назва програми в системі, наприклад httpd.

10. Користувач/група. Ім'я користувача або групи в системі.

11. Доступ. Читання, запис або виконання.

Наступні кілька розділів описують правила взаємодії, які охоплюють різні аспекти сценаріїв атак і семантику операційної системи, що впливає на безпеку.

Уразливість — це ненавмисна поведінка в системі програмного забезпечення, яку може використати зловмисник для порушення безпеки хоста. Нижче наведені предикати, задіяні в правилах про вразливості. Аргументи предикатів представлені у вигляді змінних, хоча в конкретному правилі вони можуть бути або змінними, або константами.

vulExists(Host, Program, ExploitRange, ExploitConsequence) — це похідний предикат, який вказує, що вразливість існує в Програмі на хості та має певний

ExploitRange та ExploitConsequence. Це похідний предикат. Програма – це повний шлях до виконуваного файлу, який містить помилку безпеки. ExploitRange є локальним або віддаленим, що вказує на те, чи можна використати помилку локально чи віддалено. Два загальних значення для ExploitConsequence: privilegeEscalation, що означає, що успішний експлойт дозволить зловмиснику виконати довільний код, і dos, що означає, що зловмисник може призвести до збою програми (відмова в обслуговуванні).

vulExists(Host, ID, Program) — це примітивний предикат, який вказує, що вразливість з ідентифікаційним ідентифікатором існує в програмі на хості. vulProperty(ID, ExploitRange, ExploitConsequence) — це примітивний предикат, який визначає діапазон використання та наслідки вразливості за допомогою ідентифікатора.

bugHyp(Host, Program, Range, Consequence) — це динамічний предикат, який створює гіпотетичну помилку в програмі на хості, яка має ExploitRange та ExploitConsequence. Детальніше про використання динамічних предикатів для проведення гіпотетичного аналізу обговорюється дальше.

dependentOn(Host, Program, Library) — це примітивний предикат, який визначає, що програма на хості залежить від бібліотеки, де тип бібліотеки також є «Програма».

Нижче наведено правила обчислення інформації про вразливі місця на хості. %% вводить рядок коментаря.

```
vulExists(H, Prog, Range, Consequence):-                    %%Advisory
                vulExists(H, ID, Prog),
                vulProperty(ID, Range, Consequence).

vulExists(H, Prog, Range, Consequence):-                    %%Hypothetical Bug
                bugHyp(H, Prog, Range, Consequence).

vulExists(H, Prog, Range, Consequence):-                    %%Library Bug
                vulExists(H, Library, Range, Consequence),
                dependsOn(H, Prog, Library).
```

Правила експлойту

Спочатку ми представляємо кілька предикатів, які використовуються в правилах використання. execCode(P,H,UserPriv) — це похідний предикат, який визначає, що принципал P може виконувати довільний код із привілеєм UserPriv на машині H.

netAccess(P, Src, Dst, Protocol, Port) — це похідний предикат, який визначає, що принципал P може надсилати пакети з машини Src до порту на машині Dst через протокол.

networkService(H, Prog, Protocol, Port, User) — це примітивний предикат, який вказує, що службова програма Prog працює на хості H як користувач User. Він прослуховує порт Порт протоколу Протокол. Наприклад, networkService(webServer, httpd, tcp, 80, apache) означає, що на машині webServer програма мережевої служби httpd працює як apache користувача та прослуховує порт 80 протоколу tcp.

setuidProgram(H, Prog) - це примітивний предикат, який визначає, що Prog є setuid[1] програма на хості H. Виконуваний файл належить Користувачеві.

clientProgram(H, Prog) — це примітивний предикат, який вказує, що Prog є клієнтською програмою, яка під час виконання може відкрити з'єднання з сервером через мережу. malicious(P) — це примітивний предикат, який визначає, що принципал P буде атакувати мережеву систему, щоб отримати незаконні привілеї.

inkompetent(P) — це примітивний предикат, який вказує на те, що принципал P не обережний у використанні комп'ютерів, і його поведінка може бути використана зловмисником. Теоретично передумови для використання певної помилки програмного забезпечення можуть бути довільними. На практиці переважна більшість експлойтів відбувається дуже схожим чином. Більшість помилок безпеки спричинені переповненням буфера, коли зловмисник створює спеціально створений вхід, який може вийти за межі пам'яті стека чи купи програми. Таким чином зловмисник може вставити код у пам'ять і змінити покажчик повернення в стеку, щоб змусити програму перейти до впровадженого

коду, який може бути програмою-оболонкою, яка дозволить зловмисникові виконувати довільний код. Навіть якщо введений код не може бути виконано, зловмисник все одно може призвести до збою програми та, таким чином, спричинити відмову в обслуговуванні. Якщо вхідні дані програми з помилками надходять із мережі, цю помилку можна використати віддалено (це називається дистанційним підвищенням привілеїв). Інакше зловмиснику спочатку потрібно буде мати якісь локальні привілеї на машині, де запущена програма. Якщо програма є програмою setuid, виконання програми локально на зловмисному введенні може дозволити зловмиснику отримати root (так зване локальне підвищення привілеїв). Нижче наведено два правила для віддаленого та локального підвищення привілеїв.

```
execCode(Attacker, Host, User) :-            %%Rule-remote-privilege-escalation
        malicious(Attacker),
        vulExists(Host, Program, remoteExploit, privilegeEscalation),
        networkService(Host, Program, Protocol, Port, User),
```

Тобто, якщо програма, запущена на хості, містить уразливість, яку можна віддалено використовувати, наслідком якої є підвищення привілеїв, програма з помилками працює від імені користувача та прослуховує протокол і порт, а зловмисник може надсилати шкідливі пакети до служби через мережу, тоді зловмисник може виконати довільний код на машині як користувач. Це правило можна застосувати до будь-якої вразливості, яка відповідає шаблону. Змінна підкреслення, наприклад _AttackSrc — це анонімна змінна в Datalog — змінна, яка з'являється лише один раз у реченні, і тому її значення не має значення. У цьому правилі вказується, що сервісна програма приймає пакети з будь-якої клієнтської машини, тому можна розпочати атаку з будь-якого хоста, який може надіслати пакет на сервер. Це консервативне наближення, оскільки деякі мережеві служби можуть обмежувати мережевий доступ до певних клієнтських хостів, наприклад, за допомогою TCP wrap pers [50]. У таких випадках більш точне правило

потребує визначення дійсних клієнтів замість використання дикого приведення.

```
execCode(Attacker, Host, User) :-              %%Rule-local-privilege-escalation
        malicious(Attacker),
        vulExists(Host, Prog, localExploit, privilegeEscalation),
        setuidProgram(Host, Prog),
        fileOwner(Host, Path, User),
        execCode(Attacker, Host, _SomeUser).
```

Тобто, якщо зловмисник може спочатку скомпрометувати обліковий запис (_SomeUser) на комп'ютері, і в програмі setuid, яка належить User, є локально використана помилка підвищення привілеїв, тоді зловмисник може отримати привілей User. Знову анонімна змінна _SomeUser вносить в правило консервативне наближення. Якщо локальний користувач, обліковий запис якого зламано зловмисником, не може виконатипрограма setuid, експлойт взагалі не може виникнути. Більш точне правило включало б перевірку виконуваного файлу Prog.

Інший вид експлойту відбувається на стороні клієнта мережевої програми. Клієнтська програма — це програма, яка встановлює зв'язок із сервером, наприклад веб-браузер, поштовий клієнт або програма обміну повідомленнями. Щоб використати помилку в цих програмах, жертва повинна спочатку ініціювати з'єднання із сервером, який може надавати зловмисні дані від зловмисника. Наприклад, скомпрометована веб-сторінка може містити програми Java, які використовують віртуальну машину Java в Internet Explorer, поштовий сервер може доставляти листи, які містять хробаків, які використовують уразливості в Outlook Express, сервер обміну повідомленнями MSN може передавати зловмисні запити, намагаючись використати вразливості в клієнті MSN. Щоб ці експлойти відбулися, жертва повинна спочатку щось зробити: переглянути скомпрометовану веб-сторінку, натиснути посилання в зловмисному електронному листі, відкрити клієнт месенджера тощо. Уважний користувач може уникнути таких експлойтів, дотримуючись заходів обережності. Наприклад, він ніколи не переглядає веб-сторінки з невідомих джерел, ніколи не клацає вкладення в небажаних електронних листах і блокує запити до месенджера від невідомих користувачів

месенджера. Ми класифікуємо цих обережних людей як «компетентних» користувачів. Зазвичай системні адміністратори вважаються компетентними. Але звичайний працівник, який використовує комп'ютер, вважається «некомпетентним». Цю класифікацію може надати системний адміністратор і відобразити в політиці безпеки.

Нижче наведено правило використання для віддаленого використання клієнтської програми.

```
execCode(Attacker, Host, User) :-            %%Rule-exploit-remote-client
        malicious(Attacker),
        vulExists(Host, Program, remoteExploit, privilegeEscalation),
        clientProgram(Host, Program),
        incompetent(P),
        hasAccount(P, User).
```

Тіло правила вказує, що:

1) Програма є вразливою до віддаленого експлойту.

2) Програма є мережевим клієнтським додатком.

3) Некомпетентний принципал P має обліковий запис User на машині.

Наслідком експлойту є те, що зловмисник може виконати довільний код з правами некомпетентного користувача.

Після успішного застосування експлойту зловмисник може отримати додаткові привілеї, використовуючи додаткові вразливості. Наприклад, після того, як він скомпрометував обліковий запис локального користувача через успішний віддалений експлойт, він може далі використовувати локальну помилку, щоб стати root. Він також може робити інші дії, дозволені операційною системою. Наприклад, він може читати або змінювати файли на машині або запускати атаки на інші машини звідти. У наступних розділах пояснюються правила взаємодії, які фіксують ці сценарії.

Доступ до файлу

Наступні предикати використовуються для обчислення доступу до файлу,

який принципал може мати на машині Unix.

accessFile(P, H, Access, Path) — це похідний предикат, який вказує, що принципал P може отримати доступ до файлів, указаних у Path на машині H. Доступ може бути читанням, записом або виконанням.

localFileProtection(H, User, Access, Path) — це похідний предикат, який визначає, що користувач на машині H може мати вказаний доступ до шляху до файлу. fileAttr(H, Path, R1,W1,X1,R2,W2,X2,R3,W3,X3) визначає файл UNIX у бітах данини. Наприклад, якщо на робочій станції машини атрибутом файлу /home/projectPlan.pdf є rw-r-----, відповідним предикатом буде інтерпретуються як 1 для відповідних бітів доступу.

Наступне правило говорить, що якщо зловмисник P може виконати довільний код на машині H з привілеями користувача, він може мати будь-який доступ користувача до файлів. Нижче наведено правила обчислення прав доступу до файлів у системі UNIX.

```
localFileProtection(H, User, Access, Path) :-
                fileOwner(H, Path, User),
                ownerAccessible(H, Access, Path).

localFileProtection(H, User, Access, Path) :-
                inGroup(User, Group),
                fileGroupOwner(H, Path, Group),
                groupAccessible(H, Access, Path).

localFileProtection(H, User, Access, Path) :-
                worldAccessible(H, Access, Path).

ownerAccessible(H, read, Path) :-
                fileAttr(H, Path, r,_,_,_,_,_,_,_,_).

groupAccessible(H, read, Path) :-
                fileAttr(H, Path, _,_,_,_,r,_,_,_,_).

worldAccessible(H, read, Path) :-
                fileAttr(H, Path, _,_,_,_,_,_,_,r,_,_).

ownerAccessible(H, write, Path) :-
                fileAttr(H, Path, _,w,_,_,_,_,_,_,_).
```

```
groupAccessible(H, write, Path) :-
                fileAttr(H, Path, _,_,_,_,_,w,_,_,_,_).

worldAccessible(H, write, Path) :-
                fileAttr(H, Path, _,_,_,_,_,_,_,_,w,_).

ownerAccessible(H, exec, Path) :-
                fileAttr(H, Path, _,_,x,_,_,_,_,_,_,_).

groupAccessible(H, exec, Path) :-
                fileAttr(H, Path, _,_,_,_,_,_,x,_,_,_).

worldAccessible(H, exec, Path) :-
                fileAttr(H, Path, _,_,_,_,_,_,_,_,_,x).
```

Значення правил не пояснюються. Примітивний предикат fileOwner, fileGroupOwner і fileAttr можна легко отримати з виводу команди «ls -l». Примітивний предикат inGroup можна обчислити з результату команди «groups» або з бази даних каталогу (наприклад, OpenLDAP), якщо вона використовується системою для підтримки інформації про користувача та групу.

Троянські програми

Троянський кінь — це шкідлива програма, яка маскується під доброякісну програму. Наприклад, програма для читання PDF-файлів троянської програми може передати вміст файлу зловмиснику, а SSH-клієнт троянської програми може викрасти пароль або особистий ключ користувача. Троянський кінь може навіть встановити задні двері в систему, що дозволяє зловмиснику проникнути пізніше.

Якщо зловмисник порушує цілісність файлової системи на машині,він може замінити законні програми, такі як Adobe Acrobat Reader або ssh, своєю версією троянського коня.

Коли невинний користувач запускає такі програми, зловмисник може отримати привілей у системі:

```
execCode(Attacker, H, User) :- malicious(Attacker),
                               accessFile(Attacker, H, write, Path),
                               not setuidProgram(H, Path),
                               localFileProtection(H, User, exec, Path).

execCode(Attacker, H, Owner) :- malicious(Attacker),
                                accessFile(Attacker, H, write, Path),
                                setuidProgram(H, Path),
                                fileOwner(H, Path, Owner),
                                localFileProtection(H, User, exec, Path).
```

Це ще один приклад консервативної апроксимації в правилах взаємодії. Зазвичай користувач може ненавмисно запустити програму троянського коня, лише якщо її вкинуто в каталог, включений у змінну середовища користувача «PATH». Але це правило вказує, що якщо зловмисник може змінювати файли в будь-якому каталозі, він може отримати привілеї будь-якого користувача, який може виконувати код (навіть якщо він не знаходиться в каталогах PATH користувача).

Семантика NFS

Існують також атаки, які використовують звичайну поведінку програмного забезпечення. Наприклад, розмовляючи із системними адміністраторами, ми виявили, що слабкі місця безпеки в системі обміну файлами NFS сприяли багатьом вторгненням у наш кампус. NFS не була розроблена з урахуванням безпеки, хоча зараз вона широко використовується і досить популярна. Сценарії для використання незахищеного обміну файлами існують у мережі та вже використовуються. NFS призначений для використання в локальному середовищі, де хости довіряють один одному, тобто будь-який запит на доступ до файлу від законного хоста в списку експорту сервера вважається дійсним. Ці довірчі відносини на основі хоста добре працюють за звичайних умов, але спричинятимуть серйозні проблеми, коли мережа зазнає зловмисних атак. Якщо зловмисник може скомпрометувати один обліковий запис на клієнтській машині,

він може надсилати запити NFS на сервер від імені будь-якого іншого користувача. Чи будуть запити виконані, залежить від конфігурації сервера NFS. Семантика NFS визначається такими правилами Datalog:

```
accessFile(P, Server, Access, Path) :-
        malicious(P),
        netAccess(P, Client, Server, rpc, 100003),
        nfsExportInfo(Server, Path, Client, Access, RootSquash, insecure),
        nfsUserMap(UserClient, UserServer, RootSquash),
        localFileProtection(Server, UserServer, Access, Path).

accessFile(P, Server, Access, Path) :-
        malicious(P),
        netAccess(P, Client, Server, rpc, 100003),
        execCode(P, Client, root),
        nfsExportInfo(Server, Path, Client, Access, RootSquash, secure),
        nfsUserMap(UserClient, UserServer, RootSquash),
        localFileProtection(Server, UserServer, Access, Path).

nfsUserMap(User, User, no_root_squash).

nfsUserMap(root, anonymous, root_squash).

nfsUserMap(User, User, root_squsn) :-
        nonvar(User),
```

Протокол NFS базується на RPC (віддаленому виклику процедури), і його призначений номер RPC — 100003. Примітивний предикат nfsExportInfo визначає конфігурацію на сервері NFS — який шлях експортується до якої клієнтської машини та в яких параметрах. У першому правилі параметр «захищений» експортованого ресурсу вимкнено, що означає, що запит не обов'язково надходить із привілейованого порту на клієнтській машині. [2]бути прийнятим сервером. У другому правилі ввімкнено опцію «захищений», тому атакуючий принципал P повинен спочатку отримати root на клієнтській машині. NfsUserMap відображає користувача в запиті NFS на користувача на сервері. Коли «корінь _squash" увімкнено, користувача root буде зіставлено з анонімним користувачем. Інакше ідентифікатор користувача не буде змінено після зіставлення.

Не проповідуйте _root користувач _викликає бібліотечну функцію, яка повідомляє, чи є ідентифікатор користувача «root». З формальної точки зору бібліотечну функцію можна розглядати як примітивний предикат із фізичною реалізацією, відмінною від звичайних предикатів EDB: вони явно не зберігаються

в базі даних, а реалізуються як процедури, які обчислюються під час оцінки програми Datalog. Однак перед викликом такої бібліотеки слід виконати базову перевірку, щоб гарантувати властивість безпеки [11], оскільки виклик бібліотеки за допомогою неінстанційованої змінної може повернути нескінченну кількість результатів (наприклад, існує нескінченна кількість некореневих імен користувачів). Такі небезпечні програми, як правило, створюють виняток у функції бібліотеки під час оцінювання.

На клієнтській машині NFS віддалені спільні файли монтуються в локальній файловій системі, щоб законні користувачі могли отримати доступ до файлів на сервері так, ніби вони знаходяться на клієнтській машині. Підключення полягає в тому, що якщо зловмисник може порушити цілісність файлів на сервері NFS, це вплине на користувачів клієнтської машини.

Тобто, коли принципал отримує доступ до файлів у частині, яка експортується на клієнтську машину, він також у певному сенсі «отримує доступ» до файлів на клієнті, який змонтував цю частину. nfsMounted — це примітивний предикат, наданий сканером, який визначає, що частина на сервері змонтована на локальному шляху клієнта машини.

Облікові дані користувача

Облікові дані — це певна кваліфікація, яка дозволяє користувачеві отримати доступ до системи. Паролі та ключі користувача є двома прикладами. Облікові дані часто зберігаються на машині з певним захистом. В Unix паролі користувачів криптографічно хешуються, а хеш-значення зберігається в «тіньовому» файлі, який читає лише користувач root. Приватні ключі для сеансів SSH зберігаються в домашньому каталозі користувача із захистом парольною фразою. Як тільки зловмисник отримує локальний доступ до машини, він може викрасти облікові дані користувача та скомпрометувати більше машин. Існує багато способів викрадення облікових даних користувача на комп’ютері. Якщо зловмисник стає адміністратором на машині, він може отримати тіньовий файл і провести атаку за словником, щоб вгадати неправильно вибрані паролі. Якщо зловмисник скомпрометує один обліковий запис користувача на машині, він може

отримати закритий ключ користувача, який зберігається в його домашньому каталозі. Якщо ключ не захищений парольною фразою, він може увійти на будь-яку машину, де відповідний відкритий ключ включено в конфігурацію SSH. Зловмисник може навіть встановити програму реєстрації натискань клавіш або троянську програму SSH для запису пароля або фрази-паролю користувача. Що ще гірше, необережний користувач може використовувати той самий пароль на різних сайтах. Якщо його обліковий запис буде зламано на одному сайті, його облікові записи будуть зламанітакож скомпрометовані на інших сайтах. Зловмисник також може отримати пароль некомпетентного користувача за допомогою соціальної інженерії.

Оскільки існує дуже багато ситуацій, коли облікові дані користувача можуть бути скомпрометовані зловмисником, можливо, не варто точно моделювати відмінності між усіма цими сценаріями. Швидше, наближена модель може досить добре служити нашій меті. Предикат principal Compromised (P1, P2) вводиться, щоб вказати, що облікові дані принципала P1 скомпрометовано принципалом P2 (зловмисником). Наступні два правила визначають, за яких умов скомпрометовано облікові дані принципала principal Compromised.

У першому правилі головна жертва має обліковий запис на хості H. Зловмисник отримує повний контроль над хостом як root.

```
principalCompromised(Victim, Attacker) :- hasAccount(Victim, H, User),
                                          execCode(Attacker, H, root),
                                          malicious(Attacker).

principalCompromised(Victim, Attacker) :- incompetent(Victim),
                                          malicious(Attacker).
```

У цьому випадку облікові дані жертви будуть зламані зловмисником. Як тільки зловмисник повністю скомпрометує машину, він може отримати тіньовий файл системи, отримати файл приватного ключа кожного користувача або встановити реєстратори натискань клавіш або троянського SSH-клієнта.

Цілком ймовірно, облікові дані користувача машини будуть вкрадені.

Друге правило охоплює ситуації, коли зловмиснику не потрібно отримати root, щоб вкрасти облікові дані. У цьому випадку жертва є «некомпетентною» з точки зору безпеки. Він може піддатися соціальній інженерії, використовувати пароль, який легко вгадати, або не захистити парольною фразою свій файл приватного ключа. Для такого користувача його акаунт може бути скомпрометований у будь-який момент. Отже, немає іншої передумови для правила.

Наведені вище два правила є консервативними в тому сенсі, що висновок є найгіршим-сценарій випадку. Навіть якщо корінь машини скомпрометовано, облікові дані користувача не обов'язково скомпрометовано, якщо він виявляє надзвичайну пильність і йому пощастить. Подібним чином те, що системний адміністратор класифікує особу як «некомпетентну», не обов'язково означає, що її облікові дані мають бути скомпрометовані зловмисником. Консервативне наближення в правилах взаємодії призводить до помилкових позитивних результатів аналізу. Однак це спрощує процес міркування та результуюче дерево атак, залишаючи системному адміністратору право вирішувати, чи дерево атак є реалістичним.

Після того, як облікові дані принципала скомпрометовано, зловмисник може скомпрометувати облікові записи принципала на будь-якій машині, до якої зловмисник має доступ.

Щоб використати викрадені облікові дані для компрометації іншого хоста, зловмиснику потрібен певний доступ до машини. Або він може увійти в машину як інший користувач, а потім виконати команду «su», або він зможе отримати доступ до служби входу (наприклад, sshd) на хості. Два правила для canAccessHost(P, H) визначають ці два випадки.

```
execCode(Attacker, Host, User) :- principalCompromised(Victim, Attacker),
                                  hasAccount(Victim, Host, User),
                                  canAccessHost(Attacker, Host).

canAccessHost(P, H) :- execCode(P, H, _SomeUser).

canAccessHost(P, H) :- logInService(H, Protocol, Port),
                       netAccess(P, _AttackSrc, H, Protocol, Port).

logInService(H, Protocol, Port) :- networkService(H, sshd, Protocol, Port, _User).
```

2.4 Топологія мережі

Потік пакетів у мережі впливає на здатність зловмисника здійснювати атаки. Потік пакетів контролюється міжмережевими екранами, маршрутизаторами, комутаторами та іншими аспектами топології мережі. Проектована система використовує абстрактну модель, список контролю доступу до вузла (HACL), для опису топології мережі.

Список керування доступом до хосту

Список керування доступом до хостів визначає всі доступи між хостами, дозволені мережею. Він складається з набору кортежів Datalog такої форми:

hacl (Source, Destination, Protocol, DestPort).

Це означає, що комп'ютер-джерело може досягати DestPort на комп'ютері-приймачі через протокол. HACL — це абстракція кінцевих ефектів фізичної топології, правил брандмауера, налаштувань конфігурації маршрутизаторів і комутаторів тощо. Він сумісний зі специфікаціями високого рівня, які використовуються в багатьох інструментах автоматичного керування мережею [22, 23, 5, 10]. Ці інструменти можна використовувати для надання цієї інформації.

Аспекти конфігурації не охоплені HACL

У великій корпоративній мережі багато функцій мережевих конфігурацій впливають на безпеку. Нижче наведено деякі аспекти, які не охоплює HACL. Ми залишаємо моделювання цих функцій на майбутнє.

Спуфінг — поширена методологія атаки. Наприклад, шляхом підробки адреси джерела пакета запиту NFS зловмисник може обманом змусити сервер NFS повірити, що запит надходить від законного клієнта експорту. Деякі механізми конфігурації мережі, такі як списки контролю доступу на комутаторах, можуть запобігти підробці адреси до розширити. Щоб зафіксувати цю конфігураційну інформацію в HACL, кожен запис hacl потрібно доповнити іншим полем, яке вказує на «справжнє» джерело пакетів. Ще один аспект, який не враховується HACL, полягає в тому, через яку зону може проходити певний пакет. Це актуально під час атак підслуховування. Якщо компанія дозволяє незахищений зв'язок усередині корпоративної мережі, важливо з'ясувати, чи зможе зловмисник винюхати конфіденційну інформацію, коли він заволодіє однією машиною в корпоративній мережі. Однак HACL визначає лише кінцеві точки зв'язку; проміжні кроки не фіксуються. Одним із можливих способів моделювання цього є додавання до кожного запису hacl списку зон, які дозволено проходити пакету.

Багатошаговий доступ до хосту

Предикат netAccess(P, Src, Dst, Protocol, Port) визначає, що принципал P може надсилати мережеві пакети з машини Src до порту на хості Dst через протокол. Нижче наведено правила отримання предиката.

```
netAccess(P, H1, H2, Protocol, Port) :- execCode(P, H1, _User),
                                        hacl(H1, H2, Protocol, Port).
```

Якщо принципал P має локальний доступ на машині H1 як деякі Користувач і мережа дозволяють H1 отримати доступ до H2 через протокол і порт, тоді принципал може отримати доступ до хоста H2 через протокол і порт. Це правило дозволяє міркувати про багатохостові атаки, коли зловмисник спочатку отримує доступ до одного комп'ютера в мережі та запускає наступні атаки звідти.

2.5 Специфікація політики

Політика безпеки — це єдина інформація, яку повинен надати локальний адміністратор. У проектованій системі політика безпеки визначає, який принципал до чого може отримати доступ

даних. Кожному принципалу та фрагменту даних присвоюється символічне ім’я, яке відображається на конкретній сутності за допомогою зв’язувальної інформації. Кожна заява про політику має форму дозволу (Принципал, Доступ, Дані). Аргументи можуть бути константами або змінними. Нижче наведено приклад політики:

```
allow(_Everyone, read, webPages).
allow(employee, _Access, projectPlan).
allow(sysAdmin, _Access, Data).
```

Кожен доступ є анонімними змінними. Політика говорить, що будь-хто може читати веб-сторінки; працівник може мати довільний доступ до projectPlan; і sysAdmin можуть мати довільний доступ до довільних даних. Все, що явно не дозволено, заборонено.

Мова політики, представлена в цьому розділі, проста, і її легко вказати правильно. Однак система аргументації проектованої системи також може працювати зі складнішими політиками. Наприклад, у проектованій системі можна використовувати загальний Пролог як мову політики. Більше обговорень політики та перевірки політики можна знайти в розділі пізніше.

Основні елементи та елементи даних, згадані в політиці проектованої системи, є лише символічними іменами. Вони зіставляються з конкретними об’єктами за допомогою принципального зв’язування та зв’язування даних. Прив’язка принципала відображає символ принципала на його облікові записи користувачів на хостах мережі або зону мережі, з якої працює принципал. Формат обов'язкової інформації такий hasAccount(Принципал, Хост, Обліковий запис), або розташований (основний, зона), де Принципал — це символічне ім’я для принципала, Обліковий запис — це ім’я облікового запису користувача на хості,

до якого принципал має доступ, а Зона — мережева зона, з якої принципал може працювати. приклади:

```
hasAccount(employee, workStation, ralph).
hasAccount(employee, workStation, james).
hasAccount(sysAdmin, webServer, root).
located(attacker, internet).
```

Обліковий запис, пов’язаний з користувачем, не обов’язково відповідає конкретному обліковому запису на машині. Це може означати групу облікових записів, які мають однаковий рівень привілеїв. Наприклад,

```
hasAccount(employee, workStation, employeeAccount).
hasAccount(sysAdmin, webServer, root).
located(attacker, internet).
```

Тут employeeAccount представляє будь-які звичайні облікові записи користувачів у системі. Прив'язка принципала також може містити інформацію, що описує поведінку користувача. Це зловмисні та некомпетентні предикати, згадані раніше. приклад:

```
incompetent(employee).
malicious(attacker).
```

Прив’язка даних відображає символ даних на його фізичне розташування в мережі, як правило, шлях до файлу на машині. Формат палітурки такий dataBind(Дані, Хост, Шлях). Шляхом може бути каталог, у цьому випадку кортеж dataBindDir означатиме, що «всі файли в каталозі прив’язані до символу даних Data»: dataBindDir(Дані, Хост, DirPath).

Приклад зв’язування даних:

```
dataBind(projectPlan,workstation,‘/home/projectPlan.txt’).
dataBindDir(webPages, webServer, ‘/www’).
```

2.6 Обговорення

Використання заперечень у моделі. Pure Datalog є монотонним, тобто додавання більшої кількості положень лише збільшить кількість фактів, які можна

вивести. Це відповідає монотонності атак, а це означає, що отримання більшої кількості привілеїв лише збільшує здатність зловмисника здійснювати більше атак. Однак, якщо в літералах дозволено заперечення, Datalog більше не є монотонною логікою. Введення заперечення також вплине на складність виконання програм Datalog [14]. Однак мати заперечення іноді корисно. Наприклад, припустимо, що є дія, яку зловмисник може виконати, лише якщо певний параметр конфігурації програми відсутній. Це можна змоделювати за допомогою наступного правила Datalog: action(Host, attacker) :- option_absent(Host, program, op).

Це вимагатиме від сканера звіту про відсутність параметра конфігурації, що означає, що сканер повинен мати знання про всі можливі значення параметрів для кожної програми. Якщо заперечення дозволено, правило можна переписати так: action(Host, attacker) :- not option(Host, program, op).

Дозволяючи заперечення примітивного предикату, сканеру потрібно буде повідомляти лише параметри програми, які встановлені, створюючи набагато простіший дизайн. Такого роду заперечення розшаровуються. У програмі Datalog із стратифікованими запереченнями існує частковий порядок предикатів, такий, що один предикат може залежати від заперечення іншого предиката, лише якщо перший строго менший за останній. Така форма заперечення не впливає на поліноміальну складність Datalog [14].

Немонотонні атаки

Навіть із запереченням Datalog не може моделювати загальні немонотонні атаки. Атака є немонотонною, якщо постумови, створені кроком атаки, можуть перешкоджати зловмисникові запускати нові атаки. Одним із прикладів є атаки, які компрометують доступність. Зловмисник може вивести з ладу файловий сервер, але тоді він не зможе встановити троянську програму на файловий сервер і спонукати когось запустити його. Такі сценарії не можуть бути правильно змодельовані Datalog, навіть із запереченням. Припустимо, предикат dos(H) (відмова в обслуговуванні) означає, що доступність хоста H порушена. Однією з наївних спроб є змоделювати наведений вище сценарій як наступне правило.

execTrojanHorse(victim, H) :- trojanHorseInstalled(attacker, H), not dos(H).

Жертва може запустити програму троянського коня на хості H, якщо програму троянського коня встановлено зловмисником на H, а хост H все ще живий (тобто не піддається атаці типу «відмова в обслуговуванні»). Однак це не є правильною характеристикою передумови. not dos(H) означає, що «зловмисник не може викликати стан відмови в обслуговуванні на H», тоді як ми хочемо сказати, що «зловмисник не повинен викликати стан відмови в обслуговуванні на H, щоб інсталювати троянського коня». Зрозуміло, що вони не рівнозначні.

Іншим прикладом немонотонної атаки є ситуація, коли крок атаки вимагає відсутності певних параметрів конфігурації. Якщо цей параметр встановлено спочатку в системі, але зловмисник може його видалити, атака все одно може бути успішною. Однак у Datalog, якщо відомо, що щось є істинним, неможливо ввести його заперечення, не зробивши логіку міркувань непослідовною. Таким чином, неможливо моделювати такі сценарії безпосередньо в Datalog із запереченнями.

Проблема полягає в тому, що при немонотонних атаках окремі шляхи атакистати важливим. Умови можуть змінюватися в будь-який бік на шляху: від false до true або від true до false. Перевірка моделі може проводити міркування на рівні кожного шляху, хоча це створює занадто багато накладних витрат для більш поширених монотонних атак. Для простих немонотонних сценаріїв, подібних до згаданих вище, буде достатньо простої модифікації правил атаки та деякого консервативного наближення.

Для прикладу встановлення троянського коня консервативне наближення правила виглядає наступним чином.

execTrojanHorse(victim, H) :- trojanHorseInstalled(attacker, H).

Тобто ми припускаємо, що хост H буде доступний після того, як зловмисник встановить на ньому троянську програму. Зловмисник просто вирішив не скомпрометувати доступність H, якщо тільки його здатність встановити троянського коня не залежить від того, що він виводить з ладу

машину H. У цьому випадку правило отримає більше, ніж зловмисник може насправді досягти, що є консервативним наближенням.

Для прикладу, коли зловмисник змінює параметри конфігурації, ми використовуємо приклад, представлений в останньому розділі, і змінюємо його наступним чином.

```
action(H, attacker) :- not_option(H, program, op).

not_option(H, Prog, Op) :- not option(H, Prog, Op).
not_option(H, Prog, Op) :- option_removed(H, Prog, Op).
```

Тобто ми вводимо новий предикат, а не option, щоб представити здатність зловмисника зробити опцію неіснуючою. Є дві можливості:

1) параметр не встановлено спочатку;

2) опція видалена зловмисником.

Таким чином, немонотонні атаки вимагають більш детального аналізу окремих шляхів атаки та часових співвідношень між кроками атаки. Однак за консервативним наближенням багато випадків все ще можна змоделювати в Datalog.

База даних аналізу

Розділ 2 описав правила Datalog для обчислення IDB (похідних) предикатів на основі фактів у базі даних аналізу. Ці факти представлені EDB (примітивними) предикатами в наступних категоріях:

- Факти про вразливості програмного забезпечення (такі як vulExists і vulProperty);

- Факти про конфігурацію машини (такі як експорт, fileAttr і networkService);

- Факти про топологію мережі (список контролю доступу до вузла);

- Факти про принципала та дані (обов'язкова інформація).

У цьому розділі детально обговорюється, де і як отримати ці кортежі Datalog.

2.7 Специфікація вразливості

У зв'язку з великою кількістю програмних помилок, пов'язаних із безпекою, багато агентств, які повідомляють про вразливості, починають надавати офіційні специфікації для повідомлених уразливостей. Формальна специфікація, яка зчитується машиною, а також інструменти, які їх обробляють автоматично, полегшує обмін звітами про вразливості та своєчасну роботу з ними. Специфікація вразливості складається з двох частин: специфікації розпізнавання та специфікації семантики.

Специфікація розпізнавання

Специфікація розпізнавання описує набір тестів конфігурації машини, які, якщо вони відповідають дійсності, показують наявність уразливості в тестованій системі. Для цього існує кілька мов, наприклад мова сценаріїв атак Nessus [6] (NASL) і мова відкритої оцінки вразливостей [52] (OVAL). Проектована система використовує OVAL для специфікації розпізнавання. Інтерпретатор OVAL може приймати визначення OVAL і сканувати машину на наявність вразливостей. На рисунку 2.2 наведено приклад визначення OVAL для вразливості в програмі «Ethereal», аналізаторі мережевих протоколів.

Необхідно виконати три тести, і всі вони мають бути вірними, щоб зробити висновок про наявність уразливості на машині. Три тести перераховані в елементі <criteria> і вказані в елементі <tests>. Кожен тест складається з поля об'єкта, яке ідентифікує частину конфігураційної інформації, яку потрібно перевірити, і поля даних, яке визначає значення параметра конфігурації, яке зробить тест вірним. Сканер OVAL виконає ці тести відповідно до специфікації та виведе результат. У прикладі, якщо всі три тести відповідають дійсності, уразливість, визначена номером CVE CVE-2003-0081[1], існує на машині, що тестується. Як і вхідні дані, вихідні дані також є формально визначеною схемою XML (схема результатів OVAL). Результат тесту легко витягти з вихідних даних і перетворити його на кортежі Datalog такої форми:

Наразі деякі бази даних уразливостей містять інформацію про семантику

зареєстрованих помилок безпеки програмного забезпечення. Такою базою даних про вразливості є NVD (National Vulnerability Database), розроблена Національним інститутом стандартів і технологій. Серед інформації, наданої NVD, два атрибути пов'язані зі специфікацією семантики: експлуатаційний діапазон і тип втрати.

```
<?xml version="1.0" encoding="UTF-8"?>
<oval xmlns="http://oval.mitre.org/XMLSchema/oval"  ...>
    ...
    <definitions>
        <definition id="OVAL54" class="vulnerability">
            <affected family="redhat">
                <redhat:platform>Red Hat Linux 9</redhat:platform>
                <product>Ethereal</product>
            </affected>
            ...
            <description>Format string vulnerability in packet-socks.c of
                        the SOCKS dissector for Ethereal 0.8.7 through
                        0.9.9 allows remote attackers to execute
                        arbitrary code via SOCKS packets containing
                        format string specifiers.
            </description>
            <reference source="CVE">CVE-2003-0081</reference>
            ...
            <criteria>
                <software operation="AND">
                    <criterion test_ref="rrt-201"
                               comment="Red Hat 9 is installed" />
                    <criterion test_ref="rut-201"
                               comment="ix86 architecture" />
                    <criterion test_ref="rvt-206"
                               comment="ethereal version is less
                                        than 0.9.11-0.90.1" />
                </software>
            </criteria>
        </definition>
    </definitions>
    <tests>
        <rpminfo_test id="rrt-201" check="at least one"
                      comment="Red Hat 9 is installed"
                      xmlns="http://oval.mitre.org/XMLSchema/oval#redhat">
            <object>
                <name operator="equals">redhat-release</name>
            </object>
            <data operation="AND">
                <version datatype="int" operator="equals">9</version>
            </data>
        </rpminfo_test>
        ...
    </tests>
</oval>
```

Рисунок 2.2 – Визначення OVAL

Діапазон експлуатації. Уразливість може стати причиною «локальної» та/або «віддаленої» атаки.

Нижче наведено визначення локальних і віддалених атак.

1. Місцевий.

Атаки, які використовують уразливість, можуть бути запущені безпосередньо на систему, яка атакується. Зловмисник повинен мати попередній доступ до системи, щоб розпочати атаку локально. Атака все ще визначається локальною, якщо зловмисник має віддалену оболонку для входу до хосту та ініціює атаку на цей хост, доки вона спрямована на компонент, видимий лише користувачам, які ввійшли в систему.

2. Дистанційний.

Атаки, які використовують уразливість, можуть бути запущені через мережу проти системи без попереднього доступу користувача до системи.

Діапазон експлойтів пов’язаний з передумовою експлойтів. Як обговорювалося в розділі 2.3.3, ці попередні умови класифікуються відповідно до діапазону використання помилки та характеру програми з помилками (клієнт чи сервер).

Тип втрати. Типи втрат мають форму традиційних трьох властивостей безпеки («доступність», «конфіденційність» і «цілісність»), а також нової категорії під назвою «захист безпеки».

Конфіденційність. Уразливість отримує позначку «конфіденційності», якщо вона дозволяє атаку, яка може призвести до витоку конфіденційної інформації в системі.

Цілісність. Уразливість отримує позначку «цілісність», якщо вона дозволяє атаку, яка може призвести до модифікації конфіденційної інформації в системі.

Доступність. Уразливість отримує мітку «доступність», якщо вона дозволяє атаку, яка безпосередньо перешкоджає користувачеві (людині чи машині) отримати доступ до певного системного ресурсу. Це також називається відмовою в обслуговуванні.

Захист безпеки. Уразливість отримує позначку «захист безпеки», якщо вона

дозволяє атаку, яка надає зловмисникові привілеї в системі, яких він не повинен мати. Це також називається підвищенням привілеїв. Мітка «захист безпеки» може з'являтися окремо або в трьох інших варіаціях: «захист безпеки (отримання доступу суперкористувача)», коли атака дає хакеру повний контроль над системою, «захист безпеки (отримання доступу користувача)», коли атака дозволяє хакеру частково контролювати систему, «захист безпеки (інше)», коли атака дає хакеру інші привілеї в системі.

Атрибути доступності, конфіденційності та цілісності включаються в опис здатності вразливості, лише якщо використання вразливості безпосередньо порушує одну з цих властивостей. Наприклад, якщо вразливість може надати зловмиснику підвищені привілеї, дозволяючи таким чином зловмиснику порушувати доступність, лише атрибут «захист безпеки» буде вірним. Однак, якщо одна вразливість активує два різних уразливості (як це типово для вразливостей переповнення буфера), одна з яких порушує захист безпеки, а інша безпосередньо доступність, тоді обидва атрибути будуть істинними.

Тип втрати вразливості вказує на наслідки експлойту. Значення перших двох видів втрат досить розпливчасте. Незрозуміло, яка інформація в системі може бути виточена або змінена. На щастя, два найпоширеніші типи втрат: підвищення привілеїв і відмова в обслуговуванні[2], мають відносно чіткі значення та вони моделюються в правилах міркування проектованої системи. Проектована система не розрізняє три варіанти вразливостей захисту безпеки — отримання доступу суперкористувача, користувача чи іншого доступу. Чи дозволить експлойт отримати доступ суперкористувача чи лише доступ звичайного користувача, зазвичай залежить від конфігурації програми з помилками. Якщо програма запущена від імені root, успішний експлойт потенційно може призвести до компрометації root. Інакше зловмисник може отримати доступ лише для звичайного користувача. Сама мітка gain-other-privilege не надає достатньо семантичної інформації. Як консервативне наближення, проектована система розглядає його однаково з двома іншими.

Для двох типів втрат, які не мають чіткого семантичного значення, проста зміна в базі даних NVD зробить їх більш корисними. Для багатьох уразливостей, позначених як «конфіденційність» або «цілісність», успішний експлойт дозволить зловмиснику прочитати або записати всі файли в системі, до яких уразлива програма має доступ. База даних NVD може надавати дві підкатегорії для цих двох типів втрат: «усі» та «інші», де «всі» означає втрату конфіденційності (цілісності) усієї інформації, до якої має доступ уразлива програма, а «інше» означає деяку іншу невизначену інформацію. Тоді ми можемо розробити правила міркування проектованої системи для обробки «всіх» випадків.

Моделювання поділу привілеїв

Розділення привілеїв - це техніка, яка використовується для стримування та обмеження впливу помилок програмування [39]. Програма мережевої служби, яка працює від імені root, може відправляти непривілейований дочірній процес для обробки вхідних даних мережі. Помилка в непривілейованому процесі не призводить до компрометації привілейованого процесу. Більшість віддалених експлойтів залежить від надсилання зловмисно створеного пакета до службової програми, щоб викликати переповнення буфера. Адреса повернення програми перезаписується переповненням буфера, що змушує програму переходити до коду, впровадженого шкідливим пакетом, як правило, програмою оболонки. Поділ привілеїв робить такі компроміси дуже значнимискладно, якщо не неможливо, тому що переповнення буфера відбудеться в непривілейованому процесі, компрометація якого дасть зловмиснику дуже обмежений доступ до системи.

```
execCode(Attacker, Host, User) :-     %%Rule-remote-privilege-escalation
      malicious(Attacker),
      vulExists(Host, VulID, Program),
      vulProperty(VulID, remoteExploit, privEscalation),
      networkService(Host, Program, Protocol, Port, User),
      not privilegeSeparated(Host, Program),
      netAccess(Attacker, Host, Protocol, Port).
```

Правило Rule-remote-privilege-escalation у підрозділі 2.3.3 не враховує, якщо програма використовує технологію розділення привілеїв. Це робить наступне змінене правило:

Ця модифікація для врахування розділення привілеїв підкреслює можливість розширення правил взаємодії проектованої системи. Безпека характеризується взаємодією між атаками та захистом. Поява нових методологій атаки та застосування нових методів захисту є вічно триваючими процесами. Це вимагає, щоб правила взаємодії були здатні адаптуватися до змін на останній арені безпеки, як показано на прикладі поділу привілеїв. До того, як цей метод був прийнятий, успішний експлойт дозволяв зловмиснику отримати привілеї процесу програми з помилками, який фіксується старим правилом. З новою технологією це не обов'язково так. Але модифікація правила проста: додавання нового предикату, щоб визначити, чи працює програма в режимі розділення привілеїв. Щоб обчислити цей предикат, сканеру потрібно отримати відповідну інформацію про конфігурацію. Окрім цього, жодні інші правила змінювати не потрібно.

Використовуйте наслідки, окрім ескалації привілеїв

Іншим поширеним наслідком експлойту є відмова в обслуговуванні або втрата доступності. Це часто трапляється, коли атака переповнення буфера порушує цілісність програмного стека або купи та призводить до збою програми. Порівняно з ескалацією привілеїв, це менш важкий наслідок. Але якщо доступність важлива для програми, це може бути проблемою для системного адміністратора.

Конфіденційність і втрата цілісності є двома менш поширеними наслідками використання. Існує велика різноманітність інформації, яка може бути виточена або змінена, але офіційної специфікації для цього немає. Таким чином, використання вразливостей лише з цими двома наслідками не моделюється поточними правилами взаємодії проектованої системи. Якби правила взаємодії проектованої системи моделювали ці наслідки, використовуючи інформацію в поточних базах даних, потрібно було б зробити дуже консервативне наближення.

Наприклад, втрата конфіденційності означає, що вся інформація, яка зберігається на машині, доступна програмі з помилками, може бути передана зловмиснику.

Якщо практика покаже, що ці немодельовані наслідки дійсно відіграють важливу роль у незначній частині випадків атак, нам потрібно буде ввести нові предикати та правила проектованої системи для їх моделювання та міркування. Як і у випадку запровадження ескалації привілеїв, це буде лише локальна зміна системи міркувань і, отже, не буде схильною до помилок. Однак може знадобитися додаткова інформація про вразливості, якої немає в поточних базах даних про помилки. Ми розглядаємо це як процес пропозиції для спільноти, яка повідомляє про помилки, щодо того, яку інформацію слід повідомляти у формальному, машинозчитуваному форматі.

Отримати інформацію про семантику з бази даних уразливостей так само просто, як написати запит до бази даних для отримання відповідних атрибутів. Потім результат можна перетворити на такі факти Datalog, як:

```
vulProperty('CVE-2002-0392', remoteExploit,
privilegeEscalation).
```

2.8 Конфігурація хоста

Для проведення аналізу проектованої системи необхідна різна інформація про конфігурацію машини. Більшість з них можна легко отримати, виконавши певні команди ОС або переглянувши певні конфігураційні файли. Нижче наведено список предикатів для інформації про конфігурацію хоста та відповідні способи їх отримання. Першим параметром усіх предикатів є ім'я хоста, конфігурація якого описана.

1. networkService (хост, програма, протокол, порт, користувач).

Визначає протокол і номер порту, під яким прослуховує програму мережевої служби, а також користувача, якому належить процес. Цей кортеж Datalog перетворюється з результатів команди «netstat». На рисунку 2.3 наведено приклад результату виконання «netstat -l -p» на машині Linux:

```
Proto Recv-Q Send-Q Local Address Foreign Address State  PID/Program name
tcp    0      0      *:32768       *:*             LISTEN 1596/rpc.statd
tcp    0      0      localhost.localdo:32769  *:*  LISTEN 1715/xinetd
tcp    0      0      *:42117       *:*             LISTEN 9703/rpc.mountd
tcp    0      0      *:sunrpc      *:*             LISTEN 1577/portmap
tcp    0      0      *:x11         *:*             LISTEN 1943/X
tcp    0      0      *:ssh         *:*             LISTEN 29414/sshd
tcp    0      0      localhost.localdoma:ipp  *:*  LISTEN 1786/cupsd
tcp    0      0      localhost.localdom:smtp  *:*  LISTEN 1735/sendmail: acce
tcp    0      0      *:959         *:*             LISTEN 9683/rpc.rquotad
udp    0      0      *:32768       *:*                    1596/rpc.statd
udp    0      0      *:nfs         *:*                    -
udp    0      0      *:32770       *:*                    -
udp    0      0      *:32771       *:*                    9703/rpc.mountd
udp    0      0      *:924         *:*                    1596/rpc.statd
udp    0      0      *:956         *:*                    9683/rpc.rquotad
udp    0      0      *:sunrpc      *:*                    1577/portmap
udp    0      0      *:631         *:*                    1786/cupsd
```

Рисунок 2.3 – Приклад виведення команди «netstat»

Кожен рядок виводу можна перетворити на кортеж Datalog форми netstat-l-p(H,Prot,Recv-Q,Send-Q,Local,Foreign,State,PID,ProgName). Однак інформації про користувача, якому належить процес, немає. Для цього потрібно виконати команду «ps -Af», результати якої можна подібним чином перетворити на кортежі Datalog такого формату:

```
ps-Af(H,UID,PID,PPID,C,STIME,TTY,TIME,CMD).
```

Тоді предикат networkService можна обчислити за допомогою такої простої програми Datalog:

```
networkService(H, Program, Protocol, Port, User) :-
    netstat-l-p(H, Prot, _, _, Local, _, _, PID, Program),
    ps-A(H, User, PID, _, _, _, _, _, _).
```

2. setuidProgram(хост, програма).

Цю інформацію можна отримати, виконавши наступну команду в кореневому каталозі файлової системи.

"find . -perm +4000 -exec ls -l \;"

3. clientProgram(хост, програма).

Цей предикат вказує, що програма є клієнтською програмою на хості. Клієнтська програма — це мережева програма, яка встановлює з'єднання з сервером. Веб-браузери, клієнти електронної пошти та програми обміну миттєвими повідомленнями є прикладами клієнтських програм. Оскільки Program — це повний шлях до виконуваного файлу, цей предикат можна обчислити шляхом зіставлення останнього компонента шляху з попередньо визначеним списком відомих імен клієнтських програм, наприклад «firefox, pine, gaim...». Зокрема, для кожного елемента в списку[4], наступна команда знайде шлях до виконуваного файлу програми на хості (ІМ'Я – це імена виконуваних файлів у списку).

4. Це результат виконання команди «ls -l» у файлі, указаному в Path. 9 атрибутів відповідають бітам дозволу на файл в Unix.

```
fileAttr(Host, Path, RO,WO,XO,RG,WG,XG,RW,WW,XW)

fileOwner(Host, Path, Owner)

fileGroupOwner(Host, Path, GroupOwner)
```

Наприклад, якщо виконати команду «ls -l /home/projectPlan.txt» на головній робочій станції, буде отримано такий результат:

```
-rw-r--r--    1 xou      grad        0 Mar 19 14:46 projectPlan.txt
```

Тоді відповідні кортежі Datalog є такими:

```
fileAttr(workStation, '/home/projectPlan.txt', r,w,-,r,w,-,-,-,-)

fileOwner(workStation, 'home/projectPlan.txt', xou).

fileGroupOwner(workStation, 'home/projectPlan.txt', grad).
```

Немає необхідності зберігати інформацію «ls -l» кожного файлу на машині. Сканувати потрібно лише «важливі» файли. Це файли, які мають прив'язку даних (тобто вони зіставляються за допомогою кортежу прив'язки даних до символу даних), файли, які є виконуваними двійковими файлами (зазвичай знаходяться в каталогах, що містять компонент «bin» у системах Linux), та інші файли, атрибут яких є релевантним для безпеки. Сканеру можна надати список таких каталогів і виконати таку команду в системі Linux:

"find DIR -exec ls -l \;",

where DIR is the diretory being scanned.

5. nfsExportInfo(Server, PATH, Client, Access, RootSquash, Secure).

Цей кортеж представляє конфігурацію NFS на сервері. Файл конфігурації NFS на основі ядра в Linux – це «/etc/exports». Кожен рядок у файлі представляє зв'язок експорту, який можна вказати в Datalog. Наприклад, рядок у "/etc/exports" може виглядати так /public webServer(rw,async) Це означає, що каталог /public на сервері NFS експортується на веб-сервер хоста з доступом для читання та запису. «Корінь сквош» і параметри «захищений» увімкнено за замовчуванням, інакше їх потрібно явно вказати як «без кореневого сквошу» і «небезпечний». Після внесення змін у файл експорту потрібно виконати команду «exportfs», щоб перенести зміни до таблиці спільного доступу, яка зазвичай розташована за адресою «/var/lib/nfs/etab». Кожен рядок у цій таблиці містить усі параметри експортованої частки у фіксованому порядку та з явними значеннями. Цю таблицю можна використовувати для обчислення предикату nfsExportInfo.

6. nfsMounted(Client, ClientPath, Server, ServerPath, Access).

Вказує, що ServerPath на сервері монтується на ClientPath машини Client як розділ NFS. Цей кортеж Datalog можна отримати, виконавши таку команду на клієнтській машині:

«df -t nfs -P»

Цей предикат також можна обчислити з необробленого вмісту "/etc/mtab".

Що і коли сканувати Сканування машин є важливим процесом аналізу безпеки мережі. Оскільки безпека є складною проблемою, будь-яка інформація про машину потенційно може бути корисною для визначення здатності супротивника виконувати дії.

Однак неможливо перенести повне зображення хоста для аналізу через проблеми масштабованості та конфіденційності. Більш практичним підходом є лише сканування інформації, яка вимагається поточними правилами взаємодії, і зміна сканера, коли з'являються нові методології атак, які потребують додаткової інформації для аналізу. В ідеалі сканування виконується асинхронно на кожному окремому хості щоразу, коли його конфігурація суттєво змінюється. Коли надходить новий звіт про вразливість, слід провести аналіз уже зібраних даних. Однак за поточної архітектури розпізнавання вразливості виконується як частина процесу сканування, а не як частина аналізу бази даних, створеної під час сканування. Хоча це забезпечує кращу модульність (ми використовуємо готовий сканер OVAL для розпізнавання вразливостей замість того, щоб писати власний OVAL-сумісний засіб розпізнавання вразливостей), воно також має кілька недоліків. По-перше, щоразу, коли надходить новий звіт про вразливість, потрібно сканувати кожен хост. Хоча аналіз у Datalog виконується швидко, процес сканування займає набагато більше часу та не масштабується у великій глобальній мережі. По-друге, з дерева атак, створеного механізмом аналізу проектованої системи, буде видно лише наявність певної вразливості на машині. Однак певна інформація про помилку, наприклад номер версії програмного забезпечення, встановленого на хості, параметри конфігурації програми обслуговування тощо, буде корисною для системних адміністраторів, щоб швидко зрозуміти, що пішло не так у конфігурації. І по-третє, сканер повинен отримувати вхідні дані від третіх сторін, що збільшує ймовірність того, що зловмисники введуть інформацію про компрометацію машин. Таким чином, буде вигідно відокремити процес розпізнавання від сканера OVAL і помістити його в систему аналізу проектованої системи.

Проектована система моделює мережеві конфігурації як абстрактні списки керування доступом до вузла (HACL), які є списком кортежів такого формату:

hacl(Джерело, Адресат, Протокол, Порт).

Це означає, що машина в джерелі може досягати цільової мережі через мережеву службу, визначену протоколом і портом.

За останні десять років було проведено дослідження автоматичного керування конфігурацією мережі [22, 23, 5, 10]. Деякі з цих робіт дали інструменти, які можуть керувати маршрутизаторами, комутаторами та брандмауерам відповідно до глобальних політик, подібних до HACL [22, 5, 10, 26, 25]. Проектована система має намір інтегрувати ці інструменти як частину процесу збору інформації, автоматично надаючи абстрактний список HACL.

У проектованій системі значення принципала та елементів даних у термінах конкретних облікових записів користувачів, імен файлів тощо надаються інформацією про зв'язування принципала та даних. Наразі обов'язкова інформація є частиною політики та має надаватися вручну системним адміністратором. Можна помістити основну зв'язувальну інформацію в базу даних LDAP і запитувати її під час перевірки.

2.9 Зібрати все разом

Рисунок 2.4 ілюструє архітектуру проектованої системи із показаними джерелами даних бази даних аналізу. Сканери проектованої системи, які працюють на кожному окремому хості, надають інформацію про конфігурацію машини. Smart Firewall [10] забезпечує конфігурацію мережі в термінах кортежів HACL. LDAP надає принципову зв'язувальну інформацію.

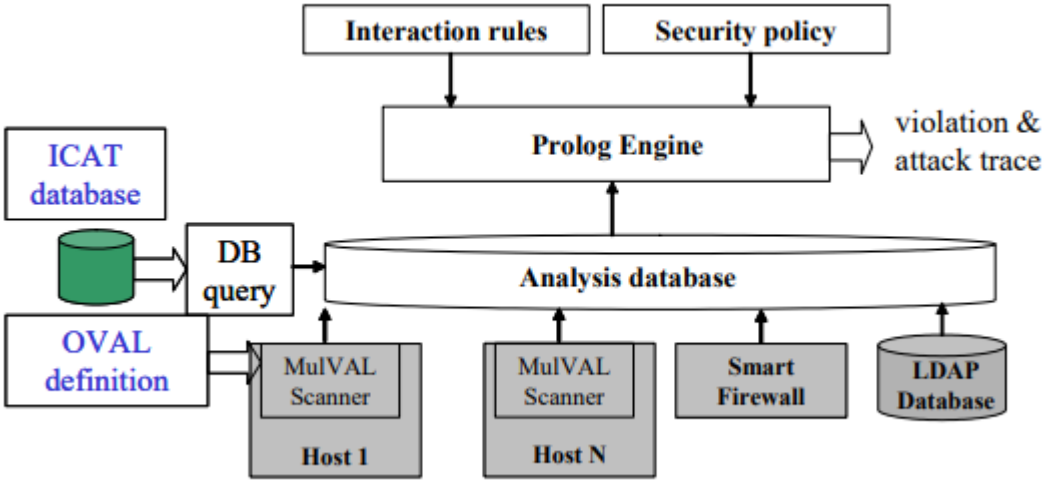

Рисунок 2.4 – Повна структура проектованої системи

Безпека політику визначає системний адміністратор, якому також необхідно визначити прив'язку даних як частину політики. Джерела даних ліворуч походять від сторонніх агентств, які повідомляють про помилки. Вони надають формальну специфікацію програмних помилок у визначеннях OVAL та з бази даних NVD.

Базовий аналіз. Модель аргументації та формальний опис конфігурації, розглянуті в попередніх двох розділах, створюють основу для проведення різних видів аналізу безпеки. Можна розглядати правила взаємодії в моделі аргументації як формалізовані експертні знання про безпекову взаємодію. Процес аналізу проектованої системи, по суті, полягає в застосуванні знань безпеки до конфігураційних даних і отримання властивостей мережі. Це процес логічної дедукції, і для конкретної логічної мови, що використовується в проектованій системі, проблема зводиться до оцінки Datalog. У цьому розділі коротко розглядаються стратегії оцінки Datalog і обговорюються два основних аналізи — симуляція атаки та перевірка політики.

Оцінка журналу даних і XSB

Datalog має дві основні стратегії оцінки: оцінка знизу вгору та оцінка зверху вниз [11]. Під час висхідного оцінювання правила програми Datalog застосовуються до вхідних фактів у EDB, щоб отримати нові факти в IDB, доки нові факти не будуть отримані. У низхідному оцінюванні, враховуючи мету,

правила застосовуються назад знайти підцілі, які мають бути істинними, щоб задовольнити вихідну мету, доки всі підцілі не досягнуть вхідних фактів. Перевага оцінки «знизу вгору» полягає в тому, що кожен факт обчислюється лише один раз. Для програми Datalog існує щонайбільше поліноміальна кількість фактів, які можна вивести. Якщо кожен факт обчислюється лише один раз, оцінка гарантовано буде поліноміальною. Оцінка зверху вниз має ту перевагу, що обчислюються лише факти, пов'язані з метою запиту. Однак наївна реалізація низхідної оцінки може обчислювати факт кілька разів, що призводить до неефективності.

Стандартна система Prolog працює за принципом «зверху вниз»: кожне правило перевіряється по порядку, а також кожна підціль правила. Він не пам'ятає, які факти вже були обчислені, тому факт може бути обчислений кілька разів, якщо він потрібен у різних місцях процесу пошуку в глибину. Це може бути проблемою для продуктивності. Серйознішою проблемою в цих двигунах Prolog є те, що цикли в правилах Datalog можуть призвести до неперервного виконання, а порядок пунктів, а також підцілей у пункті впливає на результат виконання. Наприклад, нижче наведено специфікацію Datalog для обчислення транзитивного закриття.

```
reachable(v1, v3) :- reachable(v1, v2), reachable(v2, v3).
reachable(v1, v2) :- edge(v1, v2).
```

Припустімо, факти про край такі:

```
edge(node1, node2).
edge(node1, node3).
edge(node2, node3).
```

Виконання наступного запиту в стандартній системі Prolog призведе до нескінченного циклу без виведення єдиного результату.

```
| ?- досяжний (вузол1,V).
```

Якщо ми змінимо порядок двох правил на досяжність, запит виведе три результати (правильно), перш ніж перейти до нескінченного циклу.

```
| ?- reachable(node1,V).
V = node2 ? ;
V = node3 ? ;
V = node3 ? ;
```

Коли мова заходить про моделювання комп'ютерних атак, цикли є звичайним явищем. Наприклад, зловмисник може змінити файли користувача, якщо він може виконати довільний код від імені користувача. Але можливо, що він може виконувати довільний код як користувач тому, що він змінив деякі виконувані файли та встановив програму троянського коня. Зокрема, наступні два правила взаємодії можуть спричинити цикли у виведенні.

```
accessFile(P, H, Access, Path) :- execCode(P, H, User),
                                  localFileProtection(H, User, Access, Path).

execCode(Attacker, H, User) :- malicious(Attacker),
                               accessFile(Attacker, H, write, Path),
                               not setuidProgram(H, Path),
                               localFileProtection(H, User, exec, Path).
```

Наявність циклів у правилах взаємодії цілком правомірна з точки зору семантики безпеки взаємодії. Вимагати, щоб правила взаємодії були вільними від циклу, є не тільки занадто обмежувальним, але й надзвичайно складним, якщо не неможливим. На жаль, ці цикли введуть нескінченні цикли в стандартній системі Prolog, яка переглядає програму Datalog операційно, а не декларативно.

XSB [41] — це система, яка обчислює добре обґрунтовану семантику логічних програм [20]. XSB підтримує табличне виконання, яке є різновидом техніки запам'ятовування. Простіше кажучи, виконується обчислення табличного предиката лише один раз, і результат зберігається в таблиці для повторного використання. Ефект від таблиці подвійний. По-перше, він, по суті, реалізує алгоритм динамічного програмування, щоб факти про табличний предикат не обчислювалися повторно під час виконання логічної програми. По-друге, якщо

табличний предикат бере участь у циклі під час оцінки, XSB виявить це і не ввійде в цикл. Як наслідок, цикли в програмах Datalog не вводитимуть обчислення без завершення, а порядок пунктів не впливає на результат виконання. Ця перевага робить XSB ідеальним кандидатом для логічного механізму в проектованій системі.

Властивості оцінки Datalog у XSB

Обґрунтованість і повнота Обґрунтованість і повнота стверджують, що:

1) будь-який результат аналізу повинен бути логічним наслідком правил взаємодії проектованої системи і вхідних фактів;

2) аналіз здатний обчислити всі такі логічні наслідки. Існують різні поняття семантики для Datalog, які формально визначають, що означають логічні наслідки [16, 20, 11].

Ця семантика збігається для програм Datalog із стратифікованим запереченням — єдиним видом заперечень, що використовується в проектованій системі. Система XSB може ефективно обчислювати добре обґрунтовану семантику [20], яка фіксує інтуїтивно зрозумілу висхідну семантику похідних програм Datalog. Оскільки проектована система використовує XSB як свій логічний механізм, обґрунтованість і повнота XSB у обчисленні добре обґрунтованої семантики гарантує, що аналіз у проектованій системі є надійним і повним.

На складність аналізу проектованої системи впливає складність даних правил взаємодії Datalog. Складність даних — це час оцінки програми Datalog відносно введених даних із фіксованою програмою Datalog. Для чистої програми Datalog існує лише постійна кількість предикатів, і максимальна арність предикатів також постійна. Оскільки аргумент предиката може лишенадходять із вхідної області, розмір якої пропорційний розміру вхідних даних, існує лише поліноміальна кількість фактів, які можуть бути отримані програмою Datalog. Отже, складність даних чистого Datalog щонайбільше поліноміальна. Насправді Datalog - це повні дані для P [14]. Введення стратифікованого заперечення не впливає на поліноміальну складність Datalog [14].

У XSB, якщо кожен предикат таблиці, то кожен факт буде обчислюватися лише один раз, і час виконання програми Datalog гарантовано буде поліноміальним. Однак маніпуляції з таблицями також створюють накладні витрати, які можуть протидіяти перевагам, які приносить динамічне програмування. Наразі ми вводимо лише достатню кількість предикатів, щоб уникнути нескінченних циклів у програмах. Однак точна складність процесу обґрунтування проектованої системи залежить від правил взаємодії та вхідних даних. У розділі показано деякі експериментальні результати, які ілюструють швидкість механізму міркування на великих синтезованих вхідних даних.

Симуляція атаки. Мета симуляції атаки — з'ясувати, які привілеї може отримати зловмисник, запускаючи багатоетапні атаки на кілька хостів у мережі. Нехай G — мета атаки, I — правила взаємодії проектованої системи, а D — вхідні дані в базу даних аналізу. Завдання симуляції атаки полягає в тому, щоб визначити, чи є значення $I \wedge D \Rightarrow G$. Це можна обчислити за допомогою XSB, спочатку завантаживши I та D, а потім виконавши запит: ? − G. Наприклад, наступний запит відповідає на запитання «чи може основний зловмисник отримати права root на будь-якій машині?»

```
?- execCode(attacker, H, root).
```

Система XSB автоматично знайде всі можливі способи досягнення цієї мети шляхом застосування правил взаємодії до вхідних даних. Якщо запит виконано успішно, змінні, що з'являються в запиті, матимуть рішення — екземпляри, які роблять запит істинним. У цьому випадку рішення для змінної H вказуватиме набір машин, корінь яких може бути скомпрометований зловмисником. XSB виведе наступне рішення (ні в останньому рядку вказує на те, що більше екземплярів не можна обчислити).

```
H = workStation;

H = fileServer;

H = webServer;

no
```

Оскільки модель політики в проектованій системі враховує конфіденційність і цілісність даних, ми хотіли б обчислити доступ зловмисника до файлів, які відповідають символам даних у політиці. Це робить наступна програма Datalog.

```
access(P, Access, Data) :- dataBind(Data, H, Path),
                           accessFile(P, H, Access, Path).
```

Тобто, якщо Дані зберігаються на машині H під шляхом Path, і принципал P може отримати доступ до файлів у цьому шляху, тоді P може отримати доступ до Data. Щоб обчислити всі доступи до даних, які можуть бути отримані шляхом запуску багатоетапних атак на багато хостів, потрібно лише виконати такий запит:

```
?- access(P, Access, Data).
```

Перевірка політики

Симуляція атак може обчислити всі привілеї, які зловмисник може отримати, запускаючи багатоетапні атаки на кілька хостів. Якщо ми виведемо всі ці привілеї, обсяг інформації буде величезним для перетравлення системним адміністратором. По-перше, не всі привілеї, які може отримати зловмисник, шкідливі. І багато разів одна привілей включаєбагато інших. Наприклад, якщо зловмисник отримує root на машині, він може отримати доступ до кожного окремого файлу на цій машині. Не дуже корисно виводити всі доступи до файлів, які зловмисник може мати у скомпрометованій системі. У проектованій системі політика використовується для фільтрації необроблених вихідних даних симуляції атаки та виведення лише основних небажаних привілеїв. Ці важливі результати охоплюють високорівневу мету адміністрування безпеки. Наприклад, якщо кінцевою метою адміністрування безпеки є захист конфіденційності та цілісності конфіденційної інформації, системний адміністратор може визначити політику доступу до даних у проектованої системи і перевірити систему на

відповідність їй. Це можна зробити за допомогою наступної простої програми Datalog.

```
policyViolation(P, Access, Data) :- access(P, Access, Data),
                                    not allow(P, Access, Data).
```

Більше правил

Політика доступу до даних — не єдина політика, яку підтримує проектована система. Перевірку політики проектованої системи можна розглядати як двоетапний процес. На першому етапі виконується лише симуляція атаки, і вона обчислює всі привілеї, які зловмисник може отримати, запускаючи багатоетапні атаки на багато хостів. Цей крок має поліноміальний час виконання. На другому етапі необроблені дані доступу порівнюються з даною політикою високого рівня, щоб відфільтрувати істотний небажаний доступ. Ця частина може бути виконана незалежно від симуляції атаки та може мати вищу складність, ніж поліноміальна, залежно від виразної потужності мови політики.

Система аргументації проектованої системи підтримує загальний Пролог як мову політики. Наприклад, політика юридичної фірми може вказувати, що ніхто не може отримати доступ до юридичних документів двох клієнтів, які мають конфлікт інтересів. Це можна визначити за допомогою наступної програми Prolog (насправді Datalog).

Генерація дерева атак

Поняття дерева атак було вперше введено Брюсом Шнайером [45]. У проектованій системі,дерево атак - це траса, яка показує кроки потенційного шляху атаки, що ведуть допевної мети. Кожен вузол листя дерева атак - це кортеж даних, що представляєконфігураційну інформацію або початкові привілеї зловмисника. Кожен внутрішній вузол – цеDatalog кортеж, що представляє привілеї, які зловмисник може отримати, запускаючи багатоетапні атаки,багатохостових атак. Формально, нехай v - внутрішній вузол в дереві атак, і вінмає k нащадків v1, ...vk. Тоді має існувати правило взаємодії p:-p1, ...pk та підстановкапідстановка θ, така, що v = [θ]p, і vi = [θ]pk для i = 1...k. Іншими

словами, дужевнутрішній вузол отримується зі своїх дочірніх вузлів шляхом застосування одного з правил взаємодії упроектованій системі.

Дерева атак важливі для системних адміністраторів, щоб зрозуміти, як зловмисник може досягти своєї мети. Зловмисник може досягти своєї мети, а також для прийняття рішень щодо виправлення ситуації. На рисунку 2.5 показаноприклад дерева атак. Це дерево атак демонструє потенційний шлях атаки в прикладі мережі. Ім'я правила, яке використовується для отримання кожного внутрішнього вузла, позначено награфіку позначено ім'ям правила, яке було використано для отримання кожного внутрішнього вузла. Можна почати знизу і слідувати крокам зловмисника. По-першевикористовуючи помилку в серверній програмі, він отримує локальний доступ до сервера. Потім він отримує локальний доступ до сервера.

```
|-- policyViolation(attacker,read,projectPlan)
    |-- dataBnd(projectPlan,workStation,/home)
    |-- accessFile(attacker,workStation,read,'/home')
    Rule: execCode implies file access
        |-- execCode(attacker,workStation,root)
        Rule: Trojan horse installation
            |-- malicious(attacker)
            |-- accessFile(attacker,workStation,write,'/usr/local/share')
            Rule: NFS semantics
                |-- nfsMounted(fileServer,'/export',read,
                                workStation, '/usr/local/share')
                    |-- nfsExportInfo(fileServer,/export,read,workStation)
                    |-- nfsMountInfo(workStation,/usr/local/share,
                                    fileServer,/export)
                |-- accessFile(attacker,fileServer,write,'/export')
                Rule: NFS shell
                    |-- malicious(attacker)
                    |-- execCode(attacker,webServer,apache)
                    Rule: remote exploit of a server program
                        |-- malicious(attacker)
                        |-- vulExists(webServer,CAN-2002-0392,httpd,
                                      remoteExploit,privEscalation)
                        |-- networkServiceInfo(webServer,httpd,tcp,80,apache)
                        |-- netAccess(attacker,webServer,tcp,80)
                        Rule: direct network access
                            |-- located(attacker,internet)
                            |-- hacl(internet,webServer,tcp,80)
                    |-- nfsExportInfo(fileServer,/export,write,webServer)
                    |-- hacl(webServer,fileServer,rpc,100003)
    |-- localFileProtection(workStation,root,read,/home)
|-- not allow(attacker,read,projectPlan)
```

Рисунок 2.5 – Дерево атаки проектованої системи

Для доступу до файлового сервера NFS він використовує таку програму, як NFSshell. За семантикоюNFS, як тільки зловмисник може змінити файли на сервері, клієнтська машинаяка монтує цю частину сервера, це також вплине на клієнтську машину.

Таким чином, зловмисник має можливістьможливість встановити троянську програму на клієнтську машину. Комп'ютер будескомпрометована троянським конем, а конфіденційні дані, що зберігаються на ній, будуть передані зловмиснику.зловмиснику.

У проектованій системі аналіз атак виконується програмою на мові Пролог, тому дерево атакице дерево виведення, або доказ, успішного виконання запиту на Пролозі. Існує декілька способівгенерувати докази на мові Пролог. Проектована системавикористовує підхід мета-програмування для генерації доведень.Мета-інтерпретатор у проектованій системі обробляє табуляцію так, що навіть програмиз побічними ефектами можуть коректно виконуватися в інтерпретаторі.

Генерація графів атак

Хоча дерева атак, згенеровані мета-інтерпретатором, слугують для візуалізаціївізуалізації шляхів атак, ця методологія також має кілька недоліків. Мета-інтерпретація програми Pro-log програми на порядок повільніше, ніж виконання її безпосередньо на Пролозі.Більше того, навіть якщо існує лише поліноміальна кількість фактів, які можуть бути отримані за допомогоюпрограмою Datalog, кількість згенерованих дерев доказів може бути експоненціальною у найгіршому випадку.

Система XSB включає програму-обґрунтування [44, 36], яка може обчислювати доказипохідних літералів під час роботи програми, усуваючи таким чином потребу в мета-інтерпретації.

Докази зберігаються у базі даних на мові Пролог і можуть бути вилучені для візуалізації. Згідно з результатами тестування [36], онлайн-обґрунтування вносить лише 8%накладних витрат на виконання програми, що набагато краще, ніж мета-інтерпретація. Щоб уникнутиекспоненціального розростання дерев доведень, можна виводити ациклічний граф, який візуалізуєлогічні зв'язки між

похідними літералами. Розмір таких графів є поліноміальним на розмір програми. Наразі генерація графів атак не реалізованау проектованій системі. Для того, щоб згенерований граф атак був більш корисним, необхідно створити зручний інструментякий може досліджувати інформацію, надану обґрунтуванням, був би необхідний. Це виходить за рамки бакалаврської роботи і залишається для подальшої роботи.

3 РЕАЛІЗАЦІЯ ВЛАСНОГО РІШЕННЯ

Система NetScan1 розроблена для автоматизації аналізу безпеки комп'ютерних мереж із фокусом на виявлення багатоступеневих атак, спричинених програмними вразливостями, помилками конфігурації та потенційними невідомими загрозами. Сучасні мережі, що включають тисячі хостів, різноманітне програмне забезпечення та складні політики доступу, роблять ручний аналіз безпеки неможливим. Традиційні інструменти, такі як Nessus чи OpenVAS, виявляють окремі вразливості, але не здатні моделювати комбіновані атаки чи прогнозувати вплив ще не виявлених проблем. NetScan1 розв'язує цю проблему, використовуючи логічне програмування на мові Datalog для формалізації мережевих атак, інтегруючи дані з джерел безпеки, таких як OVAL і NVD, і реалізуючи механізм оцінки стійкості мережі до гіпотетичних вразливостей, що дозволяє прогнозувати потенційні загрози навіть за відсутності відомих уразливостей.

Основна ідея полягає в тому, щоб формалізувати мережу як набір логічних фактів і правил, які описують конфігурацію, вразливості та можливі дії зловмисника. Система дозволяє вводити гіпотетичні вразливості в модель мережі та перевіряти, чи можуть вони призвести до порушення політики безпеки, наприклад, несанкціонованого доступу до конфіденційних даних. Цей підхід реалізовано через логічний рушій, який обробляє великі обсяги даних і генерує дерева атак для візуалізації шляхів компрометації. У цьому розділі детально описано інженерну реалізацію NetScan1, включаючи архітектуру, алгоритми, технічні деталі, експериментальні результати та пропозиції щодо вдосконалення, спираючись на матеріали документа.

3.1 Архітектура системи NetScan1

Система NetScan1 побудована як модульна платформа, що складається з

трьох основних компонентів, які взаємодіють для збору даних, аналізу безпеки та представлення результатів. Перший компонент — сканер конфігурації — відповідає за збір інформації про мережу. Він аналізує конфігураційні файли, такі як /etc/exports для NFS, журнали брандмауера та звіти сканерів безпеки, щоб отримати дані про хости, встановлене програмне забезпечення, активні мережеві служби та політики доступу. Сканер генерує набір Datalog-кортежів, які формалізують стан мережі. Наприклад, для мережі з веб-сервером, файловим сервером і робочими станціями сканер може створити такі кортежі:

```
hacl(webServer, fileServer, nfsProtocol, 2049).
networkService(webServer, httpd, tcp, 80, apache).
nfsExported(fileServer, '/export', webServer, read).
vulExists(webServer, 'CVE-2002-0392', httpd, remoteExploit, privEscalation).
```

Ці кортежі описують, що веб-сервер має доступ до файлового сервера через NFS на порту 2049, працює служба httpd на порту 80, каталог /export доступний для читання, а також існує вразливість CVE-2002-0392, яка дозволяє віддалену ескалацію привілеїв. Сканер використовує Perl-скрипти для парсингу конфігурацій і XSLT-перетворення для конвертації звітів OVAL у Datalog-формат, що забезпечує сумісність із різними джерелами даних.

Другий компонент — логічний рушій — побудовано на базі системи XSB, яка є розширенням Prolog із підтримкою табличного обчислення. Рушій обробляє Datalog-кортежі та набір правил, збережених у файлі attack_rules.P, для моделювання можливих атак. Табличне обчислення дозволяє кешувати результати проміжних запитів, що значно прискорює аналіз великих мереж. Наприклад, якщо запит execCode виконується для кількох хостів, XSB зберігає результати, уникаючи повторних обчислень. Рушій підтримує чотири основні алгоритми: симуляцію атак, перевірку політики безпеки, оцінку впливу гіпотетичних вразливостей і генерацію дерев атак.

Третій компонент — інтерфейс користувача — відповідає за візуалізацію результатів аналізу. Він генерує дерева атак у форматі DOT, які можна відобразити за допомогою Graphviz, і пропонує рекомендації щодо усунення вразливостей, такі як зміна правил брандмауера або конфігурації NFS. Наприклад, дерево атак може виглядати так:

```
digraph attack_tree {
    node1 [label="Exploit CVE-2002-0392 on webServer"];
    node2 [label="Gain root access"];
    node3 [label="Modify /export via NFS"];
    node4 [label="Execute trojan on workstation"];
    node1 -> node2 -> node3 -> node4;
}
```

Така структура забезпечує модульність системи, дозволяючи замінювати сканери (наприклад, Nessus на Nmap), додавати нові джерела даних або модифікувати правила без зміни ядра. Це робить NetScan1 гнучким інструментом, придатним для аналізу різноманітних мереж, від малих локальних до великих розподілених.

3.2 Реалізація гіпотетичного аналізу

Однією з ключових особливостей NetScan1 є механізм оцінки стійкості мережі до невідомих вразливостей шляхом введення гіпотетичних умов у модель мережі та аналізу їхнього впливу на безпеку. Цей процес дозволяє перевірити, чи може невідома вразливість у певному програмному забезпеченні призвести до порушення політики безпеки, наприклад, несанкціонованого доступу до конфіденційних даних. У NetScan1 це реалізовано через логічне програмування, де гіпотетична вразливість додається до бази даних як факт, після чого логічний рушій оцінює всі можливі наслідки.

Формально, гіпотетична вразливість вводиться через правило Datalog, яке описує її існування на певному хості, в конкретному програмному забезпеченні, із

заданим типом експлуатації та наслідками. Наприклад, для веб-сервера з програмою httpd правило може виглядати так:

```
vulExists(webServer, hypVul1, httpd, remoteExploit,
privEscalation) :- bugHyp(webServer, httpd, remoteExploit,
privEscalation).
```

Тут webServer — хост, hypVul1 — ідентифікатор гіпотетичної вразливості, httpd — програмне забезпечення, remoteExploit — тип експлуатації (віддалений доступ), privEscalation — наслідок (ескалація привілеїв). Факт bugHyp додається до бази даних, і рушій перевіряє, чи може зловмисник використати цю вразливість для порушення політики безпеки, наприклад, отримати доступ до даних projectPlan:

```
policyViolation(attacker, read, projectPlan).
```

Алгоритм оцінки реалізовано через метапрограму в XSB, яка ітерує через усі програми, встановлені на хостах, і перевіряє вплив гіпотетичних вразливостей. Приклад коду для однієї вразливості:

```
with_one_advisory(Sw, Range, Consequence,
policyViolation(attacker, read, projectplan)) :-
    with_hypothesis([bugHyp(_h, Sw, Range, Consequence)],
policyViolation(attacker, read, projectplan)).
```

Цей код додає гіпотетичну вразливість для програми Sw (наприклад, httpd), виконує запит policyViolation і видаляє факт після завершення. Алгоритм працює наступним чином: для мережі з n типами програм і k гіпотетичних вразливостей перевіряється (nk)\binom{n}{k}(kn) комбінацій. Наприклад, якщо в мережі 20 типів програм (n = 20), то для однієї вразливості (k = 1) виконується 20 ітерацій, а для двох (k = 2) — (202)=190\binom{20}{2} = 190(220)=190 ітерацій. Кожна ітерація включає додавання факту bugHyp, виконання запиту та очищення бази даних.

Для оптимізації продуктивності XSB використовує табличне обчислення, яке кешує результати запитів. Наприклад, якщо запит execCode оцінює можливість виконання коду на кількох хостах, проміжні результати зберігаються, що зменшує час повторних обчислень. Для мережі з 1000 хостів і 20 типами програм аналіз із однією гіпотетичною вразливістю займає 12 секунд, а з двома — 273 секунди через комбінаторну складність. Щоб зменшити кількість ітерацій, NetScan1 групує хости з ідентичною конфігурацією, розглядаючи їх як один логічний хост. Це реалізовано через правило:

```
groupHost(Host, GroupID) :- sameConfig(Host, Config),
configID(Config, GroupID).
```

Дані про вразливості інтегруються через XSLT-перетворення звітів OVAL. Наприклад, XSLT-скрипт для конвертації:

```
<xsl:templatematch="oval:definition">
<xsl:text>vulExists('</xsl:text>
<xsl:value-ofselect="host"/>
<xsl:text>', '</xsl:text>
<xsl:value-ofselect="@id"/>
<xsl:text>', </xsl:text>
<xsl:value-ofselect="software"/>
<xsl:text>, remoteExploit, privEscalation).</xsl:text>
</xsl:template>
```

Цей скрипт генерує Datalog-кортежі, які додаються до бази даних перед аналізом.

3.3 Практичне тестування системи

Для оцінки ефективності NetScan1 було проведено експерименти на реальних і синтетичних мережах. Один із тестових сценаріїв передбачав аналіз

мережі з веб-сервером у демілітаризованій зоні (DMZ), файловим сервером і робочими станціями у внутрішній зоні. Файловий сервер експортував каталог /export через NFS для веб-сервера та робочих станцій, а політика безпеки вимагала захисту конфіденційності та цілісності даних projectPlan на робочих станціях. Конфігурація мережі була представлена Datalog-кортежами:

```
hacl(webServer, fileServer, nfsProtocol, 2049).
hacl(fileServer, workstation, nfsProtocol, 2049).
nfsExported(fileServer, '/export', webServer, read).
nfsExported(fileServer, '/export', workstation, read).
vulExists(webServer, 'CVE-2002-0392', httpd, remoteExploit, privEscalation).
networkService(webServer, httpd, tcp, 80, apache).
malicious(attacker).
```

Тестування проводилося на двох апаратних платформах: сканер працював на хості з Red Hat Linux 9 (Pentium III 730 МГц, 128 МБ RAM), а логічний рушій — на ПК з Windows XP (Pentium 4 2.8 ГГц, 513 МБ RAM). XSB було налаштовано з увімкненим табличним обчисленням:

```
xsb --nobanner --quietload -e "set_tabling(on)."
```

Логічний рушій виявив порушення політики безпеки, повернувши такі результати:

```
|?- policyViolation(Adversary, Access, Resource).
Adversary = attacker, Access = read, Resource = projectPlan;
Adversary = attacker, Access = write, Resource = webPages;
Adversary = attacker, Access = write, Resource = projectPlan;
```

Дерево атак показало, що зловмисник може використати вразливість CVE-2002-0392 у httpd для отримання root-доступу на веб-сервері, модифікувати файли в /export через NFS, додати троянську програму, яка виконується на робочій станції, що монтує /export, і скомпрометувати дані projectPlan. Для усунення атаки

було виконано реконфігурацію: веб-сторінки перенесено на локальний диск веб-сервера, а доступ із DMZ до внутрішньої зони заблоковано. Це реалізовано через оновлення конфігурації NFS:

```
mv /export/webPages /var/www/html
echo"/export workstation(ro,no_root_squash)"> /etc/exports
exportfs -r
```

та додавання правила брандмауера:

```
iptables -A INPUT -s webServer -d fileServer -p tcp --dport 2049 -j DROP
```

Після змін NetScan1 підтвердив, що порушення політики безпеки усунуто.

Для оцінки гіпотетичного аналізу було проведено експеримент, у якому видалено всі відомі вразливості з моделі мережі, після чого додано гіпотетичну вразливість у httpd на веб-сервері:

```
bugHyp(webServer, httpd, remoteExploit, privEscalation).
```

Алгоритм виявив порушення безпеки, створивши дерево атак, аналогічне попередньому, але з гіпотетичною вразливістю. Час виконання склав 0.08 секунди для одного хоста, що підтверджує здатність системи прогнозувати загрози навіть за відсутності відомих вразливостей.

Продуктивність NetScan1 оцінювалась на синтетичних мережах із різною кількістю хостів, де третина хостів були веб-серверами, третина — файловими серверами, а третина — робочими станціями. Результати показали, що сканер витрачає 236 секунд на типовий хост, тоді як логічний рушій демонструє високу ефективність: для одного хоста аналіз займає 0.08 секунди (1 шлях атаки), для 200 хостів — 0.22 секунди (135 шляхів), для 400 хостів — 0.75 секунди (269 шляхів), для 1000 хостів — 3.85 секунди (669 шляхів), а для 2000 хостів — 15.8 секунди (1335 шляхів). Для гіпотетичного аналізу продуктивність залежить від кількості

програм і вразливостей: для мережі з 1000 хостів і 20 типами програм аналіз із однією вразливістю займає 12 секунд, а з двома — 273 секунди через перевірку 190 комбінацій.

Тестування в розподілених мережах проводилося на платформі PlanetLab із 33 хостами, розташованими в різних країнах. Перший хост відповів за 1.18 хвилини, 25 хостів — за 10 хвилин, а 29 хостів — за 15 хвилин. Для локальних мереж час відгуку буде меншим, але для розподілених мереж рекомендовано асинхронне сканування з таймаутом 15 хвилин.

Переваги інженерного рішення

Реалізація NetScan1 забезпечує високу ефективність завдяки використанню Datalog для формалізації атак, що дозволяє створювати чіткі та модульні правила. Наприклад, правило для моделювання виконання коду через вразливість виглядає так:

```
execCode(Principal, Host, Perm) :-
    vulExists(Host, VulID, Program, remoteExploit, privEscalation),
    networkService(Host, Program, Protocol, Port, Perm),
    malicious(Principal).
```

Це правило описує, що зловмисник може виконати код на хості, якщо існує вразливість із віддаленою експлуатацією, а програма працює як мережева служба. Така декларативність спрощує інтеграцію з джерелами даних, такими як OVAL чи NVD, через XSLT-перетворення. Табличне обчислення в XSB забезпечує швидкий аналіз, дозволяючи обробляти мережі з 2000 хостів за 15.8 секунди. Модульна архітектура системи дозволяє легко замінювати сканери або додавати нові алгоритми, а механізм оцінки гіпотетичних вразливостей робить NetScan1 унікальним інструментом для превентивного аналізу безпеки.

Обмеження та інженерні вдосконалення

Незважаючи на ефективність, NetScan1 має обмеження, які потребують інженерних рішень. По-перше, система не може моделювати немонтонні атаки, де

компрометація одного компонента блокує подальші дії, наприклад, через спрацювання системи виявлення вторгнень. Це пов’язано з обмеженнями логічного програмування, яке не підтримує часову залежність або зміну стану. По-друге, продуктивність гіпотетичного аналізу погіршується зі збільшенням кількості вразливостей: для двох вразливостей і 20 типів програм аналіз займає 273 секунди через комбінаторну складність (202)=190\binom{20}{2} = 190(220)=190. По-третє, сканування розподілених мереж може тривати до 15 хвилин через затримки в мережі.

Для подолання цих обмежень пропонується кілька інженерних вдосконалень. Щоб прискорити гіпотетичний аналіз, можна впровадити паралельне обчислення, розподіляючи перевірку комбінацій між кількома процесами за допомогою бібліотеки MPI:

```
mpirun -np 4 xsb --hypothetical_analysis
```

Це дозволить зменшити час аналізу пропорційно кількості процесорів. Додатково, групування хостів із однаковою конфігурацією може скоротити кількість ітерацій. Наприклад, якщо 100 хостів мають ідентичну конфігурацію, їх можна аналізувати як один хост, що зменшує складність із O(m) до O(1), де m — кількість хостів.

Для моделювання немонтонних атак пропонується інтеграція з Answer Set Programming (ASP), наприклад, за допомогою інструменту Clingo. ASP дозволяє специфікувати зміну стану, додаючи правила з часовими обмеженнями:

```
attackStep(Step, Time) :- previousStep(PrevStep, PrevTime), Time>PrevTime.
```

Це дозволить моделювати атаки, де певні кроки блокуються. Для автоматизації реконфігурації мережі можна розробити модуль, який генерує скрипти для брандмауерів і конфігураційних файлів. Наприклад, скрипт для блокування NFS-доступу:

```
    echo"iptables -A INPUT -s webServer -d fileServer -p tcp --
dport 2049 -j DROP"> firewall.sh
    chmod +x firewall.sh
    ./firewall.sh
```

Для прискорення сканування пропонується інтеграція з Nmap, який швидше збирає дані про мережу:

```
nmap -sV -oX scan.xml 192.168.1.0/24
```

Цей XML-вивід можна конвертувати в Datalog через додатковий XSLT-скрипт. Асинхронне сканування з адаптивним таймаутом також зменшить затримки в розподілених мережах.

Практична цінність

NetScan1 забезпечує автоматизацію аналізу безпеки, скорочуючи час із годин до секунд, прогнозує потенційні загрози через гіпотетичний аналіз, генерує конкретні рекомендації у вигляді дерев атак і скриптів для реконфігурації, а також масштабується до мереж із тисячами хостів. Це робить систему незамінним інструментом для інженерів безпеки, які прагнуть захистити складні мережеві середовища від сучасних загроз.

ВИСНОВКИ

Система NetScan1 являє собою передове інженерне рішення для автоматизованого аналізу безпеки комп'ютерних мереж, яке успішно вирішує проблему виявлення багатоступеневих атак і прогнозування потенційних загроз у складних мережевих середовищах. Її основна сила полягає в застосуванні логічного програмування на мові Datalog, що забезпечує чітку формалізацію мережевих атак і дозволяє створювати модульні, декларативні правила для моделювання сценаріїв компрометації. Використання системи XSB із табличним обчисленням гарантує високу продуктивність, дозволяючи обробляти мережі з тисячами хостів за секунди, що підтверджено експериментами на синтетичних мережах, де аналіз 2000 хостів займав 15.8 секунди. Модульна архітектура системи, яка включає сканер конфігурації, логічний рушій і інтерфейс користувача, забезпечує гнучкість, дозволяючи інтегрувати різноманітні джерела даних, такі як OVAL і NVD, через XSLT-перетворення, а також замінювати компоненти, наприклад, сканери, без зміни ядра.

Особливою цінністю NetScan1 є механізм оцінки гіпотетичних вразливостей, який дозволяє прогнозувати слабкі місця мережі до появи реальних уразливостей. Цей механізм, реалізований через ітеративний аналіз комбінацій програмного забезпечення, довів свою ефективність у тестових сценаріях, де гіпотетична вразливість у httpd призводила до компрометації даних за 0.08 секунди. Практичні тести на реальних мережах продемонстрували здатність системи виявляти складні атаки, такі як компрометація через NFS-доступ, і пропонувати конкретні інженерні рішення, включаючи скрипти для реконфігурації брандмауерів і NFS-серверів. Генерація дерев атак у форматі DOT забезпечує наочну візуалізацію шляхів компрометації, що полегшує роботу адміністраторів мережі.

Проте система має обмеження, які потребують подальшого вдосконалення. Логічне програмування не дозволяє моделювати немонтонні атаки, де

компрометація одного компонента блокує подальші дії, наприклад, через спрацювання систем виявлення вторгнень. Продуктивність оцінки гіпотетичних вразливостей знижується при аналізі кількох вразливостей через комбінаторну складність, що вимагає до 273 секунд для двох вразливостей у мережі з 20 типами програм. Сканування розподілених мереж, як показало тестування на платформі PlanetLab, може займати до 15 хвилин, що є значним обмеженням для реального часу.

Для подолання цих недоліків пропонується низка інженерних рішень. Впровадження паралельного обчислення з використанням бібліотеки MPI дозволить розподілити ітерації аналізу між процесорами, значно скоротивши час обробки великих комбінацій вразливостей. Інтеграція Answer Set Programming, наприклад, через інструмент Clingo, дасть змогу моделювати немонтонні атаки шляхом введення часових обмежень і зміни стану, що розширить можливості системи. Автоматизація реконфігурації мережі через генерацію скриптів для брандмауерів (наприклад, iptables) і конфігураційних файлів (наприклад, /etc/exports) підвищить практичну цінність NetScan1, дозволяючи не лише виявляти загрози, а й автоматично їх усувати. Прискорення сканування за рахунок інтеграції з Nmap і впровадження асинхронного сканування з адаптивним таймаутом зробить систему ефективнішою в розподілених середовищах.

Практична цінність NetScan1 полягає в її здатності скорочувати час аналізу безпеки з годин до секунд, забезпечувати превентивний аналіз через прогнозування загроз, генерувати чіткі інженерні рекомендації у вигляді скриптів і дерев атак, а також масштабуватися до великих мереж. Експериментальні результати підтвердили її ефективність у виявленні реальних і гіпотетичних загроз, а модульна архітектура гарантує адаптивність до нових викликів. Перспективи розвитку системи включають створення комплексної платформи для управління безпекою, яка поєднує аналіз, реагування та моніторинг у реальному часі. NetScan1 має потенціал стати стандартом для інженерів безпеки, забезпечуючи надійний захист складних мережевих середовищ від сучасних і майбутніх кіберзагроз.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Alur R., Henzinger T.A., Mang F.Y.C., Qadeer S., Rajamani S.K., Tasiran S. Mocha: Modularity in model checking . Proceedings of the Tenth International Conference on Computer-aided Verification (CAV 1998), Lecture Notes in Computer Science 1427, 1998. С. 521–525. URL: https://link.springer.com/ chapter/ 10.1007/ BFb0028774 (дата звернення: травень 2025).

2. Ammann P., Wijesekera D., Kaushik S. Scalable, graph-based network vulnerability analysisю. Proceedings of 9th ACM Conference on Computer and Communications Security, 2002. URL:https://dl.acm.org/doi/10.1145/586110.586112 (дата звернення: травень 2025).

3. Arbaugh W.A., Fithen W.L., McHugh J. Windows of vulnerability: A case study analysis. IEEE Computer, 2000, С. 52–59. URL:https://ieeexplore.ieee.org/ document/895190 (дата звернення: травень 2025).

4. Baldwin R. Rule based analysis of computer security. Technical Report TR-401, MIT LCS Lab, 1988. URL: https://dspace.mit.edu/handle/1721.1/6835 (дата звернення: травень 2025).

5. Bartal Y., Mayer A.J., Nissim K., Wool A. Firmato: A novel firewall management toolkit. IEEE Symposium on Security and Privacy, 1999, С. 17–31. URL: https://ieeexplore.ieee.org/document/766909 (дата звернення: травень 2025).

6. Beale J., Meer H., Temmingh R., Van Der Walt C., Deraison R. Nessus Network Auditing. Syngress Publishing, 1998, Chapter 11. URL:https://www.oreilly.com/library/view/nessus-network-auditing/9781931836081/ (дата звернення: травень 2025).

7. Bhatt S., Konstantinou A.V., Rajagopalan S.R., Yemini Y. Managing security in dynamic networks. 13th USENIX Systems Administration Conference (LISA’99), 1999. URL: https://www.usenix.org/legacy/publications/library/proceedings/lisa99/ bhatt.html (дата звернення: травень 2025).

8. Blaze M., Feigenbaum J., Ioannidis J., Keromytis A.D. The KeyNote Trust-Management System, Version 2. Request For Comments (RFC) 2704, 1999. URL: https://datatracker.ietf.org/doc/html/rfc2704 (дата звернення: травень 2025).

9. Blaze M., Feigenbaum J., Lacy J. Decentralized trust management. Proceedings of the 17th IEEE Symposium on Security and Privacy, 1996, С. 164–173. URL: https://ieeexplore.ieee.org/document/502679 (дата звернення: травень 2025).

10. Burns J., Cheng A., Gurung P., Martin D., Rajagopalan S.R., Rao P., Surendran A.V. Automatic management of network security policy. DARPA Information Survivability Conference and Exposition (DISCEX II'01), 2001, Vol. 2. URL: https://ieeexplore.ieee.org/document/932234 (дата звернення: травень 2025).

11. Ceri S., Gottlob G., Tanca L. What you always wanted to know about datalog (and never dared to ask). IEEE Transactions on Knowledge and Data Engineering, 1989, С. 146–166. URL: https://ieeexplore.ieee.org/document/43410 (дата звернення: травень 2025).

12. Clocksin W.F., Mellish C.S. Programming in Prolog. Springer-Verlag New York, Inc., 1987. URL:https://link.springer.com/book/10.1007/978-3-642-96873-0 (дата звернення: травень 2025).

13. Damianou N., Dulay N., Lupu E., Sloman M. Ponder: A language for specifying security and management policies for distributed systems. Technical report, Imperial College, 2000. URL: https://www.doc.ic.ac.uk/~mss/Papers/Ponder-TR.pdf (дата звернення: травень 2025).

14. Dantsin E., Eiter T., Gottlob G., Voronkov A. Complexity and expressive power of logic programming. Computing Surveys, 2001, Vol. 33, No. 3, С. 374–425 URL:https://dl.acm.org/doi/10.1145/502807.502810 (дата звернення: травень 2025).

15. DeTreville J. Binder, a logic-based security language. Proceedings of the 2002 IEEE Symposium on Security and Privacy, 2002, С. 105. URL: https://ieeexplore.ieee.org/document/1004367 (дата звернення: травень 2025).

16. Van Emden M.H., Kowalski R.A. The semantics of predicate logic as a programming language. Journal of the ACM, 1976, Vol. 23, No. 4, С. 733–742. URL:https://dl.acm.org/doi/10.1145/321978.321991 (дата звернення: травень 2025).

17. Farmer D., Spafford E.H. The cops security checker system. Technical Report CSD-TR-993, Purdue University, 1991. URL:https://docs.lib.purdue.edu/cstech/794/ (дата звернення: травень 2025).

18. Fithen W.L., Hernan S.V., O'Rourke P.F., Shinberg D.A. Formal modeling of vulnerabilities. Bell Labs Technical Journal, 2004, Vol. 8, No. 4, С. 173–186. URL:https://ieeexplore.ieee.org/document/5289635 (дата звернення: травень 2025).

19. Flanagan C., Godefroid P. Dynamic partial-order reduction for model checking software. Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2005, С. 110–121. URL:https://dl.acm.org/doi/10.1145/1040305.1040315 (дата звернення: травень 2025).

20. Van Gelder A., Ross K., Schlipf J.S. Unfounded sets and well-founded semantics for general logic programs. Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1988, С. 221–230. URL:https://dl.acm.org/doi/10.1145/308386.308444 (дата звернення: травень 2025).