Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Курсова робота

освітній ступінь – бакалавр

на тему: «**Автоматизоване виявлення викликів вразливого коду на Python шляхом порівняння версійності вихідного коду**»

Виконав: студент 3-го року навчання,
Спеціальності
122 «Комп'ютерні науки»

Кармелюк Костянтин Олександрович

Керівник Бабич Т.А.,
магістр комп'ютерних наук, асистент
«11» травня 2021 р.

Київ – 2021

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Освітній ступінь бакалавр

Спеціальність 122 «Комп'ютерні Науки»

Освітня програма бакалавр

ЗАТВЕРДЖУЮ

Завідувач кафедри інформатики

Гороховський С. С.

"10" жовтня 2020 року

ЗАВДАННЯ
ДЛЯ КУРСОВОЇ РОБОТИ СТУДЕНТУ

Кармелюк Костянтина Олександровича

1. Тема роботи **Автоматизоване виявлення викликів вразливого коду на Python шляхом порівняння версійності вихідного коду**, керівник роботи Бабич Трохим Анатолійович, магістр комп'ютерних наук, асистент

2. Строк подання студентом роботи 17 травня 2021

3. План роботи

- Анотація
- Вступ
- Розділ 1. Дослідження та аналіз предметної області
  - 1.1 Опис предметної області
  - 1.2 Python
    - 1.2.1 Синтаксис програм
    - 1.2.2 Структура програми
    - 1.2.3 Об'єкти
    - 1.2.4 Передача параметрів
    - 1.2.5 Безпека
  - 1.3 Аналіз програм
    - 1.3.1 Статичний
    - 1.3.2 Динамічний
    - 1.3.3 Аналіз помічних даних
    - 1.3.4 Види вразливостей

ГРАФІК ПІДГОТОВКИ КУРСОВОЇ РОБОТИ ДО ЗАХИСТУ

№ з/п	ПЕРЕЛІК РОБІТ	Термін виконання	Дата ознайомлення наукового керівника	Підпис наукового керівника	Примітки
1.	Вибір теми, затвердження її на засіданні кафедри та закріплення наукового керівника Узгодження календарного графіка підготовки кваліфікаційної роботи. Ознайомлення студента з критеріями оцінювання кваліфікаційної роботи (п. 8.5).	10 жовтня 2020			
2.	Вивчення джерел літератури, матеріалів архівів, періодичних видань, збір та узагальнення фактів, даних	10 жовтня 2020 – 25 листопада 2020			
3.	Складання плану каліф. роботи та узгодження з науковим керівником	27 листопада 2020			
4.	Написання розділів роботи або Постановка експерименту, аналіз отриманих результатів наукового дослідження	27 листопада 2020 – 15 березня 2021			
5.	Проміжний контроль виконання роботи	10 лютого 2021			
6.	Написання кваліфікаційної роботи в цілому, ознайомлення з її першим варіантом наукового керівника	16 січня 2021 – 30 березня			
	Розділ 1 (постановка проблеми, теоретичні основи, огляд літературних джерел)	28 січня 2021			
	Розділ 2 (аналітично-дослідницька частина) (експериментальна частина для природничих і біологічних наук)	2 березня 2021			
	Розділ 3 (проектно-рекомендаційна частина) (аналіз результатів експерименту для природничих і біологічних наук)	29 березня 2021			
7.	Повне завершення написання кваліфікаційної роботи, оформлення її згідно з вимогами й подання на відгук науковому керівнику	2 квітня 2021 – 5 травня 2021			
8.	Подання кваліфікаційної роботи для перевірки письмових робіт студентів НаУКМА на відповідність вимогам академічної доброчесності,	17 травня 2021			
9.	Публічний захист кваліфікаційної роботи перед екзаменаційною комісією	згідно з розкладом роботи ЕК			

Графік узгоджено «10» жовтня 2020 р.

Науковий керівник Бабич Трохим Анатолійович

Виконавець курсової роботи Кармелюк Костянтин Олександрович

Зміст

Анотація

У цій роботі розглядаються різні методики аналізу вразливостей програм, зокрема динамічний та статичний. Описується створення програми для пошуку вразливостей на основі статичного аналізу за допомогою методу, який базується на різності версій програми, включно з описом підходу побудови абстрактного синтаксичного дерева, графу потоку управління та їх аналізу, за допомогою мови програмування python.

Ключові слова: статичний аналіз, аналіз помічених даних, вразливість, абстрактне синтаксичне дерево, граф потоку управління, безпека, python.

Вступ

Зловмисники постійно намагаються використовувати нові вразливості, до того, як вийдуть оновлення, які їх виправлять. Кількість розголошених вразливостей, відповідно до баз даних загальновідомих вразливостей, з кожним роком не зменшується, а тільки зростає, і з 2017 року набрала різкого зростання (див. рисунок 1).

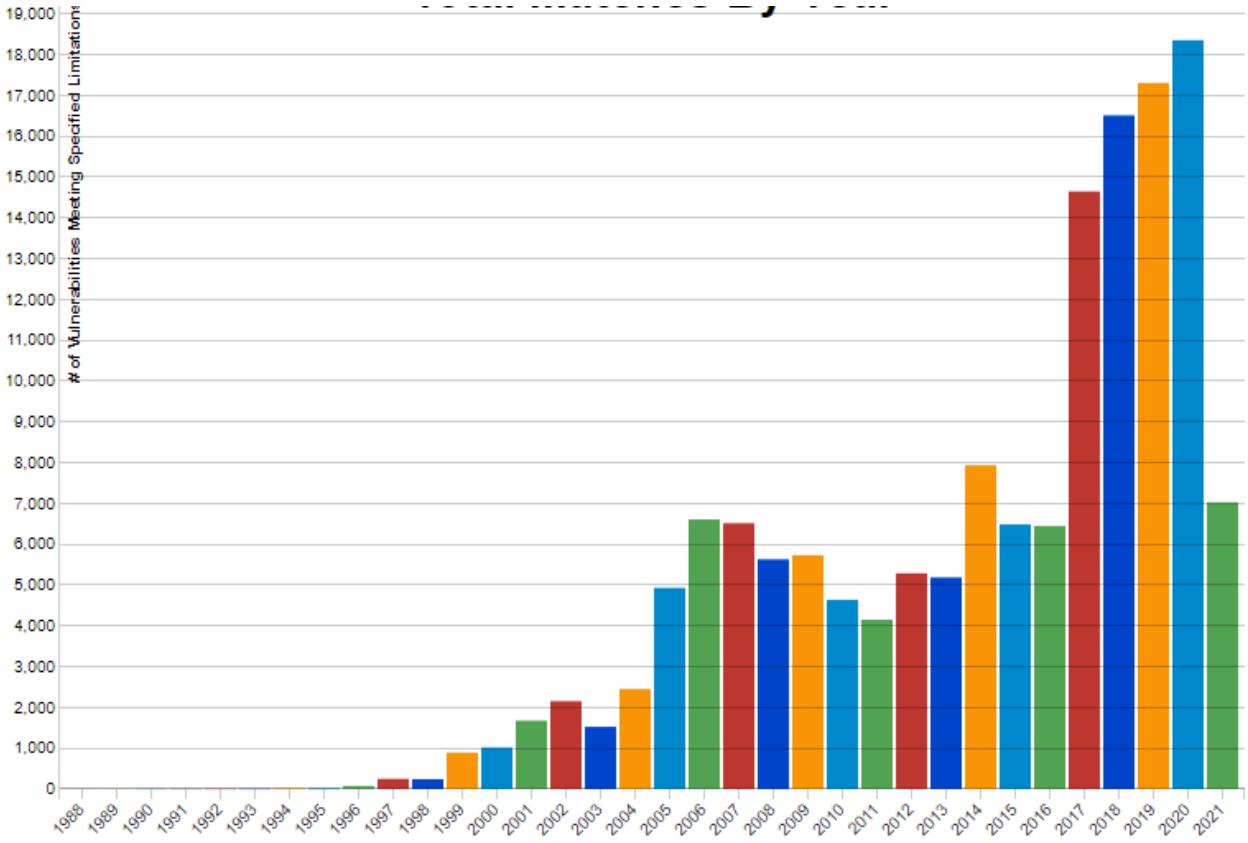


Рисунок 1 - діаграма кількості розголошених вразливостей за кожен рік, відповідно до NVD (National Vulnerability Database)

З одного погляду випуск оновлення, яке виправляє вразливість, тільки покращує безпечність програми. Проте з іншої сторони оновлення розкриває дані про вразливість старої версії, що надає зловмисникам змогу використання цієї вразливості.

Процес оновлення, що існує наразі, в якому користувачі поступово отримують оновлення, вразливий до створення експлойтів шляхом порівняння версій програми. Отримання нової версії програми надає можливість

проаналізувати, які вразливості були виправлені, що надає значну перевагу тим, хто отримав оновлення, перед тими, хто ще не оновився.

Робота розділена на два розділи.

Перший розділ містить опис мови програмування python, її особливості, і опис методик аналізу програм. В ньому також було розглянуто і проаналізовано найпоширеніші види вразливостей.

Другий розділ містить опис підходу до пошуку вразливостей, та його реалізація. Після того наведено результати роботи програми на проаналізованих вразливостях.

Метою цієї роботи було створення програми на мові python, яка за допомогою вихідних кодів старої та нової версії буде знаходити потенційні вразливості, які присутні в старій версії, що дозволить створення експлойтів в майбутньому.

Було виокремлено наступні завдання:

1) Побудувати абстрактні синтаксичні дерева двох версій програми та графів потоку управління з отриманих дерев.
2) Знайти зміни між графами потоку управління
3) Вибудувати шляхи змінних, що потрапляють у «видалені» вирази нової версії програми та використати аналіз помічених змінних.
4) Проаналізувати отримані шляхи на можливу наявність вразливостей
5) Перевірити працездатність написаної програми за допомогою тестування

Для перевірки гіпотези було використано попередньо створені пари - стара і нова версія програм, в яких відповідно існувала вразливість і була виправлена.

Розділ 1. Дослідження та аналіз предметної області

1.1 Опис предметної області

Безпека програмного забезпечення напряму залежить від якості написаного коду. Навіть малі помилки можуть призвести до критичного погіршення рівня безпеки та зробити програму вразливою до атаки.

Постає потреба в аналізі програмного забезпечення як на відповідність певним конвенціям написання коду, так і на відсутність потенційних вразливостей. Тестування якості написання коду призведе до зменшення витрат на підтримку програмного забезпечення, а аналіз на наявність вразливостей може попередити можливі втрати в майбутньому.

Такі критерії важко ідентифікувати за допомогою людських ресурсів, оскільки з часом проєкти стають доволі великими. Тому рішенням може бути програмне забезпечення для статичного та динамічного аналізу коду, яке направлене на контроль якості розробки програм.

Проте пошук вразливостей також можливий і зловмисниками для використання виявлених вразливостей.

1.2 Python

Python – інтерпретована, об’єктноорієнтована, високорівнева, динамічно типізована мова програмування [1]. Основною реалізацією мови є CPython, яка написана на C і має відкритий код. Також існують інші реалізації – IronPython, написана на C#, PyPy написана на RPython, Jython написана на Java [2]. В даній роботі буде розглянуто python версії 3.8.5 звичайної реалізації.

1.2.1 Синтаксис програм

Вирази в програмах, написаних на python, відокремлюються за допомогою перенесення стрічки, проте вираз також можливо розтягнути на декілька рядків за допомогою спеціального символу “\”, якщо він стоїть в кінці рядку. Також вираз вважається одним рядком, навіть якщо він фізично розтягнутий між декількома рядками, наприклад коли вираз знаходиться в дужках. Рядки, що починаються з символу “#” трактуються як коментарі.

1.2.2 Структура програми

Для групування виразів використовується ідентація, за pep8, який визначає стандарти написання програм на python, - вона має дорівнювати чотирьом проміжкам, тобто блок визначається його ідентацією. Вирази з однаковою ідентацією належать до одного блоку, і блок переривається, якщо рядок має іншу кількість індентацій.

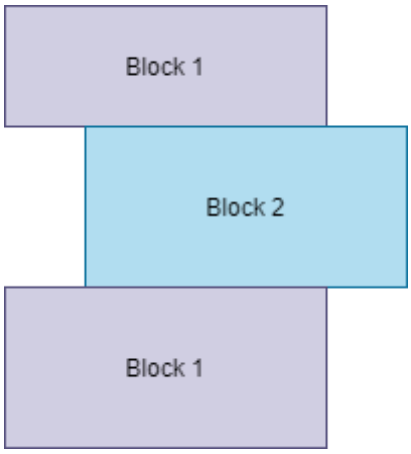


Рисунок 1.1 – блочна структура python програми

На рисунку 1.1 можна побачити, що перший блок був перерваний другим блоком, а потім перший блок знову продовжується після другого блоку.

Файл програми складається з блоків, які містить вирази, і має розширення .py. З файлів, які знаходяться в теці разом з файлом __init__.py утворюється пакет.

1.2.3 Об’єкти

Будь-яка інформація в python представляється у вигляді об’єкту або залежностей між об’єктами. Кожен об’єкт має ідентифікатор, тип і значення. Ідентифікатор об’єкту не змінюється протягом роботи програми. В реалізації CPython ідентифікатор – це адреса в пам’яті, де зберігається об’єкт.

Тип визначає які операції об’єкт підтримує і визначає можливі значення об’єкту цього типу.

Значення об’єктів можуть змінюватись. Об’єкти, значення яких можливо змінювати називаються «змінюваним», а відповідно тих, чиї значення неможливо змінити – «незмінюваними». Наприклад примітивні типи: число, рядок, тощо - є незмінюваними, а листи (масиви), словники – є змінюваними. [3]

Об’єкти визначаються в реалізації класів. Класи можуть мати атрибути і методи для підтримки і зміни стану.

На рисунку 1.2 створено клас Value, який має одну змінну об’єкту val і один метод об’єкту get_val. Метод __init__() – це конструктор об’єкту класу, який приймає параметр val. Зарезервоване слово self надає змогу звертатися до значень і методів екземпляру класу (аналог this в інших мовах програмування).

```python
class Value:
    def __init__(self, val):
        self.val = val

    def get_val(self):
        if isinstance(self.val, list):
            return self.val[1]
        return self.val

```

Рисунок 1.2 – приклад створеного класу

1.2.4 Передача параметрів

Передача параметрів у функції можлива зазвичай двома способами:

- За значенням – функція отримує копії об’єктів аргументів, переданих у виклик.
- За посиланням - функція отримує об’єкти аргументів, та значення, які ці об’єкти мають.

```python
def append_to_list(input_list: list):
    input_list.append("1")
```

Рисунок 1.3 – функція, що додає значення до переданого листа

```python
def assign(input_list: list):
    input_list = [0]
```

Рисунок 1.4 – функція, що змінює значення переданого листа

При передачі за значенням параметри, які отримує функція, мають однакове значення, у порівнянні із значенням змінних, що були передані в виклик, проте вони зберігаються в різних ділянках пам’яті. Тому функції на рисунку 1.3 та рисунку 1.4 не змінять значення об’єкту змінної, що була передана.

Передача за посиланням, в свою чергу, надає змогу змінювати змінну, яка була передана. Функції на рисунку 1.3 та рисунку 1.4 змінять значення об’єкту.

Python використовує інший підхід, так званий “передача об'єкту за посиланням”. Функція створює нову змінну, яка вказує на значення об’єкту змінної, яка була передана. В результаті дві змінні вказують на одне і те саме значення, тому функція на рисунку 1.3 змінить значення об’єкту, проте функція на рисунку 1.4 не змінить значення об’єкту, оскільки присвоєння змінить посилання локальної змінної на значення. [4]

1.2.5 Безпека

Відкритий проєкт забезпечення безпеки веб-застосунків (OWASP) – це відкрита некомерційна спільнота, яка надає інформацію у вигляді інструкцій, статей та рекомендацій, що написана для допомоги розробникам та власникам веб-застосунків в покращенні безпеки їх продуктів. Головний принцип спільноти – відкритість та безплатність інформації, що представлена на їх вебсайті.

Проєкт з безпеки python OWASP, створений для аналізу веб-застосунків, написаних на python, виділив такі три направлення:

- Безпека в python (тестування білого ящика – тестування, під час якого інформація про вихідний код доступна)
- Безпека python (тестування чорного ящика – тестування, під час якого вихідний код є відсутнім)
- Безпека за допомогою python (розробка безпечного python, який підходить для середовищ з великим риском) [5]

Оскільки python – система з відкритою екосистемою розробки, будь-хто може поширювати розроблені програми, які можливо використовувати у

інших проектах. Таке поширення створює потенціальну небезпеку, яку розробники мають усвідомлювати.

Яким чином розробники можуть підвищити безпечність програми:

- Перевіряти користувацькі дані. Деякі функції python, наприклад eval, popen, subprocess є небезпечними, оскільки можуть виконувати команди в командній стрічці або порушувати логіку програм.
- Вести облік залежностей програми. При встановленні пакетів інших розробників, вони можуть мати залежності від інших пакетів, які можуть бути небезпечними.
- Дотримуватися практик безпеки і періодично перевіряти програми за допомогою статичного та динамічного аналізу.

1.3 Аналіз програм

Аналіз програм зменшує можливість появи дефектів програми та дозволяє виявити вразливі частини коду на ранніх стадіях розробки. Для вияву вразливих частин програми використовують різні підходи. Найбільш поширені – динамічний, статичний та так званий аналіз помічених даних (taint-аналіз). У кожного є свої недоліки, тому для підвищення якості аналізу підходи можуть комбінуватися.

1.3.1 Статичний

Статичний аналіз – процес вияву помилок та вразливостей під час аналізу коду програми. Такий аналіз можливо також розглядати, як автоматичний огляд коду. Програма, яка статично аналізує, ставить за мету виділити вразливі ділянки коду без виконання самої програми, і також може використовувати методи аналізу помічених даних та аналізу потоку даних.

Статичний аналіз охоплює множину інструментів від тих, які перевіряють окремі вирази, до таких, які аналізують цілі вихідні коди програм. Інформація, отримана з аналізу, може вказувати на можливі порушення конвенцій написання коду або на можливу вразливість програми, що дозволяє нейтралізувати проблеми на ранніх стадіях розробки.

Для статичного аналізу використовують абстрактні синтаксичні дерева або графи потоку управління. Побудова абстрактного синтаксичного дерева – надалі АСД, відбувається у 3 кроки:

1. Токенізація (лексичний аналіз)

Початкова фаза побудови АСД – перетворення вхідного потоку символів, або програми, у послідовність окремих токенів, які мають вигляд

<token-name, attribute-value>

де перший параметр – умовне позначення, що використовується в синтаксичному аналізі, а другий – безпосередньо розпізнана лексема. [6, ст. 6]

Наприклад програма:

total = price + taxes + 20

буде перетворена у послідовність токенів:

<id, total> <assign,=> <id,price> <plus,+> <id,taxes> <plus,+> <number,20>

де

- id – умовне позначення ідентифікатору
- plus – умовне позначення оператору додавання
- number – умовне позначення числа

На цьому етапі лексичний аналізатор лише відокремлює відповідні лексеми, відкидаючи проміжки між ними, і не перевіряє в якому порядку дані лексеми мають бути, проте він може помітити такі помилки: неправильні токени (лексеми, які неможливо розпізнати), недопустимі символи в токени (незакінчений коментар, недопустимі символи в стрічці тощо).

2. Парсер (синтаксичний аналіз)

Друга фаза – з послідовності токенів, отриманих в першій фазі, утворити деревоподібне представлення, що зображує синтаксичну структуру програми, відповідно до заданої граматичної структури. Таке представлення називається дерево розбору або конкретним синтаксичним деревом. Типовим представленням дереву є знаходження нетермінальних конструкцій мови у внутрішніх вершинах, а аргументів – у дочірніх вершинах.[6, ст. 45]

З минулої програми отримали дерево розбору (рис 1.5). Замість «…» мають бути проміжні вершини.

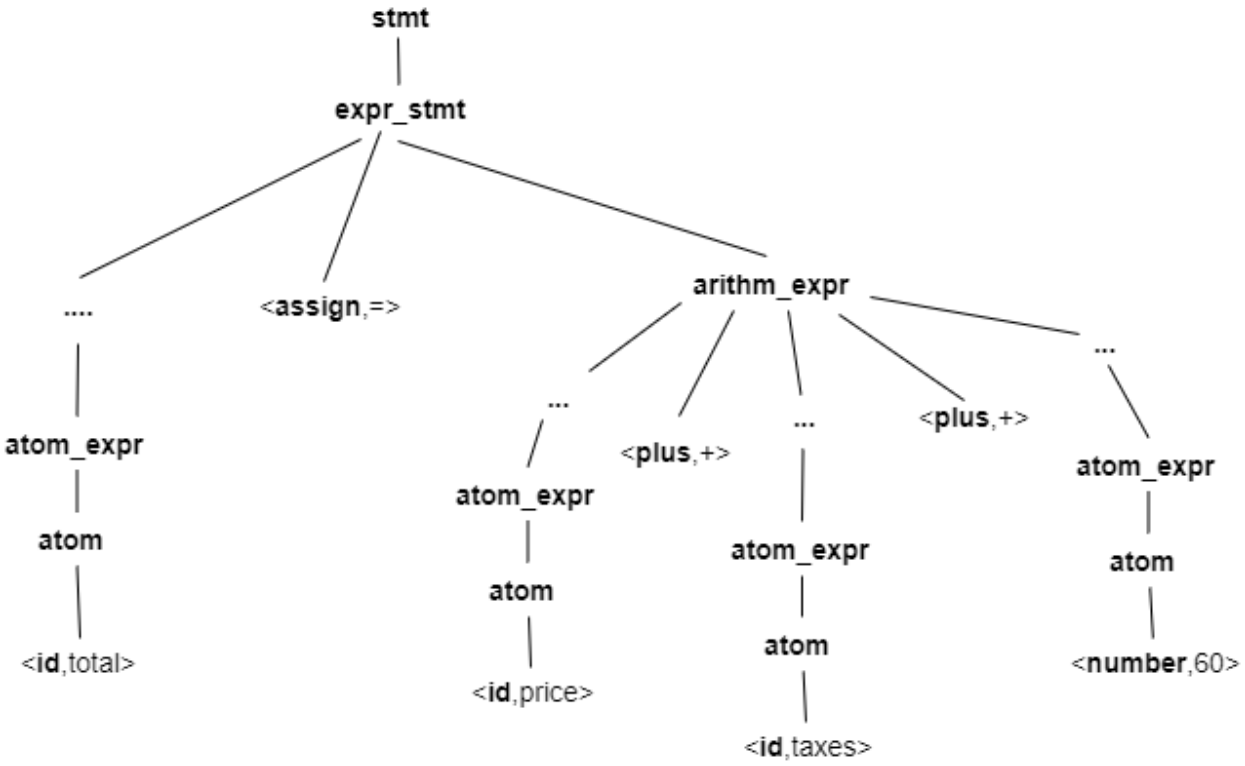


Рисунок 1.5 – дерево розбору

3. Перетворення дерева розбору

Дерево розбору може містити «допоміжні» нетермінальні конструкції у внутрішніх вершинах, які не впливають на семантику програми (наприклад дужки, вершини, які утворюються заданою граматикою для нівелювання невизначеності тощо), які в результаті спрощуються (видаляються). Остання фаза побудови – перетворення дерева розбору в абстрактне синтаксичне дерево [7].

Отримано абстрактне синтаксичне дерево (рис 1.6) з дерева розбору (рис 1.5). В результаті спрощення було видалено вершини проміжні вершини, які не впливали на семантику програми, і дерево стало більш абстрактним, незалежним від граматики.

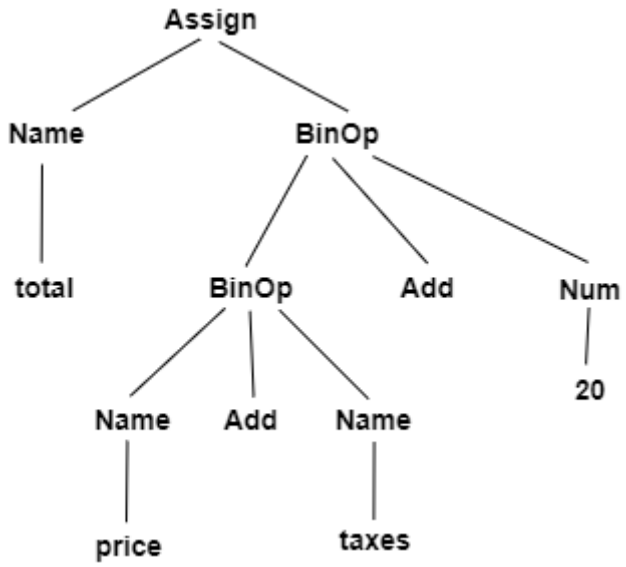


Рисунок 1.6 – абстрактне синтаксичне дерево, отримане з дерева робзбору

За допомогою абстрактного синтаксичного дерева вже можливо проводити:

- синтаксичний аналіз – чи була зміна оголошена до використання, чи стоїть конструкція переривання (break, continue тощо) у дозволених місцях тощо.
- Перевірка типу – чи кожен оператор має дозволені операнди. Наприклад чи була функція викликана з необхідною кількістю параметрів. [6, ст. 99]

Оскільки статичний аналіз проводиться без виконання програми, неможливо визначити реальний результат виконання, тому статичний аналіз лише вказує можливі порушення та вразливості, що призводить до появи неправильно позитивних відповідей (не усі виявлені вразливості є насправді вразливостями). Проте статичний підхід з великою впевненістю може виявити переповнення буферу, або SQL та інші види ін'єкцій.

Технологія швидко розвивається, але наразі вона не може виділити багато типів вразливостей, наприклад вияв проблем автентифікації або небезпечного використання криптографії, оскільки такі вразливості помітні лише під час виконання програми [8].

1.3.2 Динамічний

Аналізу коду програми недостатньо для вияву усіх вразливостей та проблем зі швидкодією програми. Динамічний аналіз використовує підхід, в основі котрого, на відміну від статичного, - аналіз програми під час її виконання. Такий аналіз не може довести, що програма відповідає заданим властивостям, як конвенціям написання коду, проте за його допомогою можливо виявити порушення та надати корисну інформацію про виконання програми. [9]

Можна виділити таку корисну інформацію, яка отримується за допомогою динамічного аналізу:

- Використання ресурсів – час виконання програми, кількість запитів до баз даних тощо.
- Програмні помилки – проблеми роботи з пам'яттю, умови перегонів тощо.

Оскільки тестування програми відбувається під час її виконання, то можливість отримання неправильно позитивних відповідей відсутня, оскільки вияв проблеми базується не на аналізі, а на прецеденті.

Головний недолік такого підходу - для ефективного вияву вразливостей необхідно виконати програму з достатньою кількістю тестів. Це означає, що наявна вразливість може залишитись непоміченою. Більш того, для правильного аналізу, таке тестування не має впливати на швидкодію програми, що призводить до необхідності обмеження впливу програми, що аналізує на систему.

1.3.3 Аналіз помічних даних

Аналіз помічених даних базується на принципі, що жодній інформації, яка надходить від користувача не можна довіряти. Такий аналіз можна поділити

на два типи: динамічний та статичний. Динамічний аналізує програму на рівні машини, тому для нього немає потреби доступності вихідного коду, на відміну від статичного, який конвертує вихідний код в абстрактне синтаксичне дерево для подальшого аналізу [10].

Вираз, який містить дані користувача називають джерелом, а конструкцію, яка може бути вразливою – злив. Між джерелом і зливом може існувати нейтралізатор – вираз, який нейтралізує дані, які приходять від користувача, і, зазвичай, нейтралізатор замінює символи, які мають спеціальне значення в контексті злива.

Такий аналіз використовують для відстеження користувацьких даних між джерелами та зливами. Виділення великої кількості даних як небезпечних може призвести до втрати точності і появи неправильно позитивних результатів. Проте виділення малої кількості може не виявити наявні вразливості, тому аналіз помічених даних вважають точним, якщо він виділяє не забагато і не недостатню кількість даних.

Перший етап аналізу – позначення даних, що приходять від користувача, як небезпечних. Такий етап також називають представлення помічених даних.

Наступний етап – поширення даних. На даному етапі прослідковується поширення помічених даних. Будь-які дані, що взаємодіяли з помічними даними вважаються також поміченими, а дані, які попали в нейтралізатор, стають нейтралізованими.

Останній етап – вияв вразливостей за допомогою перевірки попадань помічених даних у зливи.

Розглянемо головну проблему статичного аналізу помічених даних – складно визначити усі нейтралізатори для заданих джерел. Проте мови програмування зазвичай мають власні реалізації функцій нейтралізаторів. Наприклад функція «escape» в модулі «cgi» мови програмування Python конвертує спеціальні символи HTML, що містяться в рядку у відповідні

аналоги (наприклад символ «<» конвертується у «&lt»), що робить отримані дані можна безпечними для інтерпретування у HTML.

Подібні функції набагато легше виявляти, аніж рукописний розробником нейтралізатор, який замінював би спеціальні символи. Попри працездатність обох підходів, статичні аналізатори, скоріше за все, повідомлять, що вираз, який нейтралізується рукописним нейтралізатором, є небезпечним.

Можливим розв’язанням проблеми може бути задання аналізатору правила, яке буде визначати будь-яку взаємодію з заміною символів, або іншою нейтралізацією, як таку, що нейтралізує дані. Проте це не гарантує покращення роботи аналізатора, оскільки розробник може помилитися під час написання нейтралізатора. Тому такі проблеми вирішуються поєднанням динамічного та статичного підходів.

1.3.4 Види вразливостей

Найбільш поширене джерело вразливостей – відсутність перевірки даних, що приходять від користувача. За документом «OWASP Top 10 2017», в якому містяться найбільш поширені джерела вразливостей, відсутність перевірки вхідних даних (можливість ін’єкції) знаходиться на першому місці, XXE та небезпечна десеріалізація знаходяться на 4 і 8 місцях відповідно. Розглянемо приклади таких вразливостей у мові python.

1.3.4.1 OS ін'єкція

OS ін’єкція – атака, головна ціль якої виконання команд на комп’ютері хоста, а тобто системі, на якій виконується програма. Така атака можлива лише за умови доступу користувацьких даних до командної стрічки і за недостатньої перевірки вхідних даних.

```
def ping(ip):
    address = ip
    cmd = "ping -c 1 %s" % address
    os.popen(cmd)
```

Рисунок 1.7 – приклад програми, вразливої до OS ін’єкції

На рисунку 1.7 можна побачити вразливу функцію, яка приймає користувацькі дані і виконує команду «ping» з заданою IP-адресою. «Popen» – метод вбудованого модуля python «os», створює файл, в якій зберігаються результати виконання команди, яка передана. Якщо користувач передасть дані у вигляді «127.0.0.1 && rm –rf .», то програма виконає команду «ping», а після виконає команду «rm –rf .», що видалить усі файли та теки в наявній директорії, якщо програма працює на юнікс-подібній системі і має необхідні для цього привілеї.

Для попередження такої атаки необхідно використовувати метод нейтралізації вхідних даних за допомогою вбудованого методу «quote» в модуль «shlex», який повертає пристосований до використання в командній стрічці рядок.

1.3.4.2 SQL ін'єкція

SQL ін’єкція – атака, що полягає у ін’єкції SQL запиту через користувацькі дані у вже наявний запит на стороні програми, для зміни поведінки цього запиту. Успішна атака надає змогу отримати чутливі дані з бази даних, виконувати операції над базою (Додавання, оновлення, видалення) тощо. SQL ін’єкція описана як найбільш поширена вразливість веб застосунків [11].

```
def hello(request):
    id = request.GET.get("id", "")
    cursor = connection.cursor()
    cursor.execute("SELECT username FROM auth_user WHERE id=%s" % id)
    row = cursor.fetchone()
    return HttpResponse("Hello %s" % row[0])
```

Рисунок 1.8 – приклад програми, вразливої до SQL ін’єкції

На рисунку 1.8 можна побачити вразливу функцію, яка виконує запит до бази даних з ідентифікатором користувача, за допомогою форматування рядків. Якщо користувач передасть в якості параметра «id» рядок «33 OR 1=1», то запит поверне усі рядки, що знаходяться в таблиці «auth_user». Або більш небезпечний рядок id = «33; DROP TABLE auth_user», що видалить таблицю «auth_user» з бази даних.

Для попередження SQL ін’єкції є декілька можливих рішень. Використання підставляння параметрів замість форматування рядків може нейтралізувати вразливість. Наприклад, в python вони мають вигляд [12]

«SELECT username FROM auth_user WHERE id=:id", {"id": id}»

Або

«SELECT username FROM auth_user WHERE id=%s ", ("id")»

Також гарною практикою є надання найменш необхідних привілеїв для роботи з базою даних.

1.3.4.3 XXE

XXE (XML External Entity) – атака, спрямована на отримання конфіденційної інформації, відказу роботи, сканування пристрою, на якому запущена програма. Така атака можлива у програмах з неправильно налаштованим оброблювачем, який оброблює вхідні дані у вигляді XML документів і виникає, коли такий документ містить посилання на зовнішню сутність.

Зовнішня сутність може викликати читання парсером даних, які знаходяться за даним посиланням. Початково такий функціонал використовувався для змоги роботи XML з зовнішніми файлами, так само як і з мережними ресурсами. Посилання на джерело великого об’єму може спричинити відказ роботи програми, а посилання на вразливі файли може дістати їх вміст.

```
parser = etree.XMLParser(resolve_entities=True)
tree1 = etree.parse('ressources/xxe.xml', parser)
root1 = tree1.getroot()
```

Рисунок 1.9 – приклад програми, вразливої до XXE атаки

На рисунку 1.9 неправильно налаштовано парсер, оскільки він буде оброблювати посилання на зовнішні сутності. Якщо на вхід було подано XML файл, в якому міститься сутність «<!ENTITY xxe SYSTEM "file:///etc/passwd" >», яка вказує на файл, в якому на юнікс-подібних операційних системах зберігається інформація про ім’я користувачів та їх паролів, то зловмисник отримає чутливі дані.

Для попередження XXE необхідно налаштовувати парсер таким чином, щоб він не оброблював зовнішні сутності. В python це робиться таким чином:

«etree.XMLParser(resolve_entities=False, no_network=True)»

1.3.4.4 Десереалізація

Десеріалізація може бути вразливою, якщо десереалізуються дані, яким, можливо, не треба довіряти, і, зазвичай, правдивість джерела переданих даних неможливо підтвердити. Атака десереалізацією може призвести до віддаленого виконання команд.

В Python для роботи з сереалізацією використовується влаштований модуль pickle, що виконує команду «dumps» для сереалізації і команду «loads» для десереалізації. Серіалізований об’єкт в python виглядає як пара, де перший елемент – це метод, другий – параметри, які необхідно передати в цей метод для десереалізації. Метод «__reduce__» визначає, яким чином будь-який об'єкт в python має сереліазуватися.

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen()
connection = s.accept()
received_data = connection[0].recv(1024)
data = pickle.loads(received_data)
```

Рисунок 1.10 – приклад програми, вразливої до атаки десереліазацією

На рисунку 1.10 можна побачити вразливий серверний додаток, який приймає запит на підключення і десереалізує дані, що були передані. Зловмисник може перевизначити метод «__reduce__» так, щоб при десереалізації було виконано команду «os.system(rm –rf .)», яка видалить всі файли в поточному каталозі програми, якщо програма має необхідні для цього привілеї.

Для запобігання вразливості можна створювати новий об'єкт і наповнювати його значеннями, замість виконання десереалізації. Також можливо використовувати засоби забезпечення «чистоти» даних, наприклад використання HMAC для підтвердження, що дані не були змінені під час передачі. Також необхідно надавати програми найменш необхідні привілеї.

1.3.4.5 Віддалене виконання команд

Віддалене виконання команд – атака (різновид ін’єкції), яка передбачає ін’єкцію коду, що виконається програмою. Відрізняється від OS ін’єкції тим, що дозволяє виконувати код, в той час, як OS ін'єкція виконує системну команду. Така вразливість існує в мовах програмування, які мають можливість виконувати рядок як вираз під час роботи програми.

```
def compute(request):
    comp = request.GET.get("comp")
    print("Result = ", eval(comp))
```

Рисунок 1.11 – приклад програми, вразливої до віддаленого виконання команд

На рисунку 1.11 функція виконує вираз, який приходить у вигляді рядку в запиті. Якщо користувач передасть арифметичну операцію, наприклад «5 + 10», то python підрахує результат такої операції і поверне його. Проте якщо користувач передасть вираз «__import__(‘os’).system(‘rm –rf .)’», то буде такий самий результат, як у прикладі з OS ін’єкцією. Вираз «__import__» динамічно імпортує влаштований модуль «os» і викликає функцію «system», яка виконує задану команду в командній стрічці.

Можливим рішенням буде перевірка наявності заборонених символів в переданих параметрах, наприклад подвійного підкреслення, або наявність оберненої скісної риски перед лапками тощо. Влаштовані модулі python не мають нейтралізаторів такої вразливості.

Розділ 2. Опис підходу та реалізація

В основі пошуку вразливостей покладено ідею різності коду нової та старої версій програми. Існує два підходи знаходження вразливостей в старих версіях за допомогою різності версій:

- Знаходження нейтралізаторів, які з'явилися лише в новій версії, і генерування вхідних даних, на які ці нейтралізатори впливають. Такий підхід більш динамічно орієнтований і створений для генерації експлойтів, які в результаті виявляють і також експлуатують вразливості. Він описаний у роботі Девіда Брумлі [13].
- Знаходження частин, які відсутні в новій версії, і використання аналізу помічених даних, початково позначивши змінні, що входять у видалені вирази. Такий підхід статично орієнтований і створений для вияву вразливостей, з можливістю генерації експлойту в майбутньому.

В даній роботі було використано другий підхід. Якщо нова версія – латка старої, тобто в ній виправлено вразливість, тоді відсутність частин старої версії в новій, можливо, означає що ці частини були вразливими. Для аналізу необхідно побудувати графи потоку управління старої та нової версії, які в свою чергу будуються з абстрактного синтаксичного дерева, і проаналізувати шляхи змінних.

2.1 Побудова абстрактного синтаксичного дерева

Для побудови було використано вбудований модуль «ast», який використовується компілятором cpython. Вибір влаштованого модулю, замість парсера написаного за допомогою, наприклад, ANTLR, пояснюється надійністю та зручністю використання. Більш того такий вибір підтримує аналіз програм на python, що будуть написані в майбутньому без необхідності підтримання парсеру.

Деякі компілятори об’єднують токенізацію і парсинг, якщо вони використовують парсери, які можуть самостійно формувати токени. Проте cpython розділяє два процеси для спрощення токенізатора і задання незвичної поведінки токенізатора для спрощення процесу парсингу.

Під час токенізації відділяються так звані токени «відступу», оскільки python використовує їх для відокремлення блоків програми (див розділ 1.2.2). Таке рішення призводить до полегшення роботи парсеру, оскільки в звичайному токенізатору зазвичай видаляються символи проміжків. Для позначення проміжків токенізатор імітує токени «відступу».

Також для токенізації спеціальних слів «async» та «await», які позначають асинхронність в python. Оскільки в попередніх версіях, до введення асинхронності, змінні чи інші конструкції могли мати назву зарезервованих слів, які позначають асинхронність в нових версіях. Поява такої проблеми також породила необхідність відокремлення токенізації та парсингу. Для її вирішення токенізатор виділяє слова «await» лише перед функціями та класами, а «async» лише в асинхронних конструкціях, як зарезервовані.

Для парсингу cpython використовує LL(1) граматику, що означає, що отримані токени аналізуються зліва-направо, будується лівосторонній вивід з переглядом вперед на 1 токен. В результаті отримується представлення у вигляді дерева розбору.

Останній крок – перетворення дерева розбору в абстрактне синтаксичне дерево. Представлення кожної вершини АСД в модулі «ast» визначені за допомогою абстрактного синтаксичного задання мови Zephyr (ASDL). На рисунку 2.1 зображено запис за допомогою такого задання, який описує два види виразів: оголошення функції і оператор повернення результату. Обидва вирази мають тип «stmt», що зображує вертикальна лінія, яка розділяє вирази. Позначки поруч з визначенням атрибутів позначають необхідність і кількість таких атрибутів: «*» позначає, що атрибут має з'явитися 0 або багато разів, «?» позначає, що атрибут необов'язковий. Відсутність позначки означає, що

атрибут обов'язковий і має бути один раз. З такого задання генеруються класи, що типізують вершини дерева.

```
stmt = FunctionDef(identifier name, arguments args,
                   stmt* body, expr* decorator_list, expr? returns,
                   string? type_comment)
      | Return(expr? value)
```

Рисунок 2.1 – приклад задання за допомогою абстрактного синтаксичного задання мови Zephyr

Перетворення відбувається рекурсивно, спускаючись по дереву розбору, його вершини перетворюються у відповідні типізовані вершини абстрактного синтаксичного дерева. Наприклад отримана вершина «comparison» з дерева розбору викликає функцію «ast_for_comp», яка побудує представлення порівняння в АСД.

Увесь процес токенізації, парсингу і побудови АСД модуль ast виконує за допомогою команди «parse», яка на вхід приймає текст програми.

2.2Граф потоку управління

Для аналізу програми недостатньо лише АСД, оскільки є потреба аналізу потоку програми – яким чином частини коду взаємодіють між собою, що дозволить аналізувати не лише синтаксичний аналіз, а й аналіз вразливостей програми, тому після побудови АСД для аналізу необхідно побудувати граф потоку управління [14].

2.2.1 Структура

Граф потоку управління (ГПУ) – представлення потоку програми у вигляді направленого графу, де вершини графу – блоки, які містять частини програми, які виконуються починаючи з першої інструкції і закінчуючи останньою без можливості пропуску проміжних інструкцій, а ребра – можливі переходи потоку між блоками [15]. Два блоки (А і Б) пов’язані лише тоді, коли можливе виконання блоку Б безпосередньо після блоку А.

Було задано таку структуру ГПУ:

1. Блок – вершина графу (рис 2.2), в якому:

- statements – частини коду, які виконуються в даному блоці
- func_calls – виклики функцій в даному блоці
- predecessors, exits – відповідно множини вхідних та вихідних блоків

```
class Block:
    def __init__(self, block_id: int) -> None:
        self.id = block_id
        self.statements = []
        self.func_calls = {}
        self.predecessors = []
        self.exits = []
```

Рисунок 2.2 – структура вершини графу потоку управління

2. Ребро (рис 2.3), в якому:

- source – блок, з якого виходить ребро
- target – блок, з яким пов’язаний source
- exitcase – умова переходу (для умовних конструкцій)

```
class Link:
    def __init__(
            self,
            source: Block,
            target: Block,
            exitcase=None,
    ) -> None:
        self.source = source
        self.target = target
        self.exitcase = exitcase
```

Рисунок 2.3 – структура ребра між вершинами графу

3. Граф потоку управління (рис 2.4), в якому:

- entryblock – вхідний блок, який не має попередників.
- functioncfg – відокремлені ГПУ для функцій.

```
class CFG:
    def __init__(
            self, name: str
    ):
        self.name = name
        self.entryblock = None
        self.functioncfgs = {}
        self._nodesByClass = {}
```

Рисунок 2.4 – структура графу потоку управління

Така структура надає змогу легко використовувати граф в подальшому аналізі та виявлені вразливостей.

2.2.2 Побудова

Для перетворення АСД у ГПУ можна використовувати два підходи:

- Відвідувач – для обробки дочірніх елементів необхідно викликати обхід кожного вручну. Якщо не викликати для деяких нащадків обробку – вони не будуть оброблені.
- Слухач – обробка дочірніх елементів виконується автоматично. Слухач отримує інформацію про входження та вихід з вершини дерева.

Для побудови ГПУ було розроблено клас CFGBuilder, який використовує паттерн відвідувач і використовує клас NodeVisitor модулю ast.

```
def visit(self, node):
    """Visit a node."""
    method = 'visit_' + node.__class__.__name__
    visitor = getattr(self, method, self.generic_visit)
    return visitor(node)
```

Рисунок 2.5 – Метод обходу вершин

На рисунку 2.5 визначено метод обробки вершини заданого типу. Якщо тип обробки такої вершини не заданий, то викликається метод self.generic_visit, який викликає обхід усіх нащадків вершини.

На рисунку 2.6 реалізовано метод, який викликається при обробці вершини з типом присвоєння. Він додає вираз присвоєння в поточний блок і викликає обробку для усіх нащадків поточної вершини.

```
def visit_Assign(self, node: Assign):
    self.current_block.add_statement(node)
    self.generic_visit(node)
    self.add_node_to_class(node)
```

Рисунок 2.6 – метод обробки вершини, з типом присвоєння

Вершини з умовними операторами потребують складнішої обробки, оскільки вона відбувається рекурсивно. На прикладі обробки умовного оператора «if» (див додаток А) спочатку оброблюється умовна частина, а потім створюється два нових блоки – частина, що йде після умовного оператора, і тіло умовного оператора – вирази, що виконується, якщо умова справджується. Також необхідно оброблювати умовний оператор «else», який слідує після тіла, але перед частиною, що йде після умовної конструкції.

В результаті отримано представлення коду у вигляді графу потоку управління.

2.3 Пошук та відображення вразливостей

Наступний етап – аналіз отриманих графів потоку управління.

2.3.1 Знаходження змін між графами

Для виявлення вразливостей необхідно в першу чергу порівняти вихідні ГПУ старої та нової версії. Підхід також оснований на припущенні, що функції зберігають свою назву в обидвох версіях програми.

Для виявлення змін було створено клас CFGComparator, який порівнює дві версії коду. Було розроблено метод, що порівнює на повну відповідність вирази, які належать до одного типу (присвоєння, умовна конструкція тощо). На вхід подається вираз старої версії та усі вирази такого типу нової версії.

Повернення негативної відповіді (false) означає, що вираз, який наявний в старій версії, відсутній в новій, і відповідно має бути проаналізований (див додаток Б).

Такий аналіз відбувається в два кроки – спочатку порівнюються вирази, наявні в блоках ГПУ, а після порівнюються функції, що мають однакове ім'я (див додаток В). Задана структура ГПУ надає можливість зручного порівняння функцій. В результаті буде отримана різниця коду старої та нової версії, а тобто вирази, які відсутні в новій версії.

2.3.2 Вибудова шляхів змінних

Наступний крок аналізу - вибудова шляху використання змінних, що потрапляють у потенційно вразливі вирази.

Для розв’язання цієї задачі було створено додаткові класи:

- змінна (Variable) – містить усі значення та використання змінної.
- значення (Value) – містить значення змінної. Створення такого класу обумовлюється передачею параметрів в python (Розділ 1.2.4).

Така задача вирішується таким алгоритмом:

1. Ітеративно перевіряється кожен вираз старої версії.
2. Якщо вираз потенційно вразливий і не був ще розглянутий – змінні, які використовуються у виразі дістаються і помічаються, як потенційно вразливі.
3. Тепер в кожному наступному виразі перевіряється, чи використовується в ньому вразлива змінна. Якщо так – тоді перевіряється, чи це присвоєння, чи виклик функції, інші вирази відкидаються, і цей вираз додається до «використання» цієї змінної з заданим значенням. Якщо це присвоєння нового значення – додається нове значення до змінної. Також змінні, які були залучені у виразах разом із поміченими, стають поміченими.

В результаті отримаємо усі шляхи вразливих змінних - їх використання зі значеннями у виразах.

2.3.3 Пошук вразливостей

Останній і головний етап аналізу – пошук вразливостей за допомогою аналізу шляхів змінних.

З отриманих шляхів легко визначити чи існує дійсно вразлива конструкція. Щоб перевірити чи є конструкція вразливою був заданий перелік вразливостей у вигляді словника, який є легко розширюваним (рис 2.7).

```
"XXE attack": {
    "sanitizers": [["resolve_entities=False", "no_network=True"]],
    "sink": ["etree.parse("],
    "level": "Moderate"
},
```

Рисунок 2.7 – приклад задання вразливостей у програмі

На рисунку 2.7 задано такі позначення:

- Ключ – назва вразливості
- Sanitizers – так звані нейтралізатори вразливості. Можуть бути задані у різних конструкціях. В прикладі наведено масив – означає, що усі дані нейтралізатори мають бути наявні в шляху даної змінної.
- Source – джерело вразливості, зазвичай – метод, який є небезпечним.
- Level – рівень небезпеки даної вразливості.

Алгоритм вияву вразливостей (див додаток Г):

1. Для кожної змінної перевіряємо її використання. Якщо вираз містить джерело вразливості, тоді програма робить припущення, що цей вираз є вразливим.
2. При вияву вразливості йде перевірка шляху змінних, що потрапляють у вразливий вираз, на наявність нейтралізатора у шляху до моменту

даного виразу. За вияву нейтралізатора програма помічає задану вразливість як нейтралізовану.

В результаті буде отримано перелік усіх можливих вразливостей в програмі, але робота такої програми може видавати хибно позитивні відповіді, оскільки програма перевіряє всі можливі шляхи програми, які під час роботи програми можуть бути недосяжними.

2.3.4 Графічне відображення

Для графічного відображення було використано застосування graphviz [16]. Graphviz – програмне застосування з відкритим кодом, призначене для графічного відображення графів. За його допомогою можливо зберігати відображення у різних форматах, а також налаштовувати параметри відображення – колір тексту, шрифт, табуляція тощо.

Запустивши на перевірку спеціально сформованих пар (стара та нова версія) програм, програма виділила вразливі місця відповідно до створеної бази вразливостей (див додаток Ґ), описаних у розділі 1.3.4 і створила графічне відображення.

Результати роботи програми відповідно бази вразливостей:

1) OS ін’єкція (див додаток Д)

Для заданої вразливості:

```
"OS injection": {
        "sanitizers": ["shlex.quote("],
        "source": ["os.popen(", "subprocess.run(", "os.spawnl("],
        "level": "Critical"
    },
```

Вразливим є виклик команд os.popen, subprocess.run, os.spawnl зі змінною, яка не був нейтралізований за допомогою shlex.quote.

Було побудовано відображення (рис 2.8):

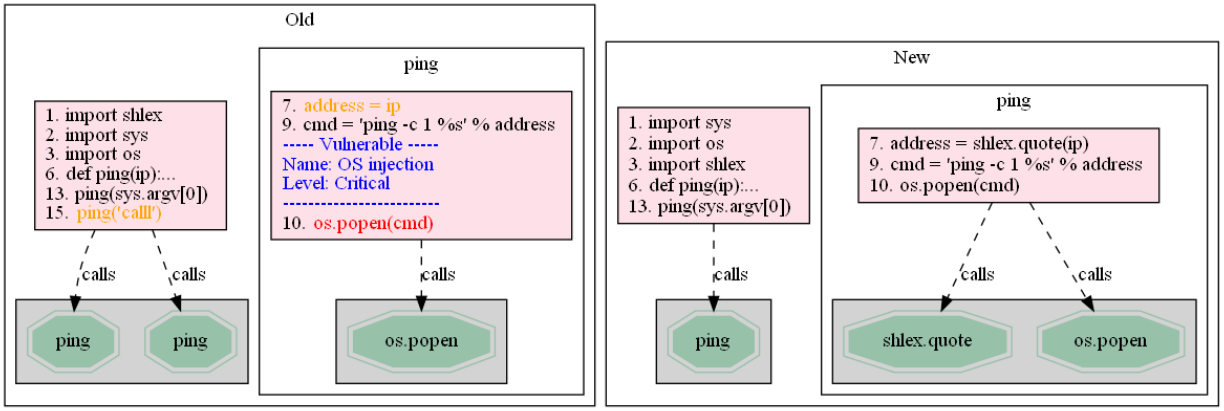


Рисунок 2.8 – Приклад роботи програми, на якому виявлено OS ін’єкцію

2) SQL ін’єкція (див додаток Е)

Для заданої вразливості:

```
"SQL Injection": {
        "sanitizers": [],
        "source": [["select", "%s", "%"], ["from","%s","%"]],
        "level": "Critical"
    },
```

Вразливим є підстановка значень безпосередньо у SQL запит за допомогою форматування рядків.

В результаті аналізу вразливості було побудовано відображення (див додаток Ї, Й)

3) XXE (див додаток Є)

Для заданої вразливості:

```
"XXE attack": {
        "sanitizers": [["resolve_entities=False", "no_network=True"]],
        "source": ["etree.parse("],
        "level": "Moderate"
    },
```

Вразливим є створення неправильно налаштованого парсеру XML документів.

Було побудовано відображення (рис 2.9):

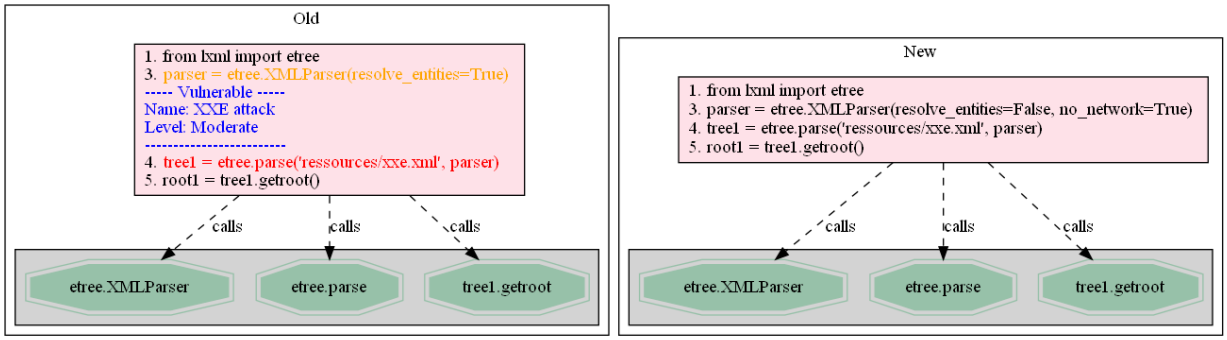


Рисунок 2.9 – приклад роботи програми, на якій виявлено можливу XXE вразливість

4) Десереалізація (див додаток Ж)

Для заданої вразливості:

```
"Deserialization attack": {
    "sanitizers": [],
    "source": ["pickle.loads("],
    "level": "Critical"
},
```

Десереалізація за допомогою «pickle.loads» завжди є вразливою і не може бути нейтралізована.

Було побудовано відображення (див додаток И, І)

5) Віддалене виконання команд (див додаток З)

Для заданої вразливості:

```
"Remote Code Execution": {
   "sanitizers": [],
   "source": ["eval("],
   "level": "Critical"
}
```

Виклик методу eval завжди є вразливим і не може бути нейтралізованим.

Було побудовано відображення (рис 2.10):

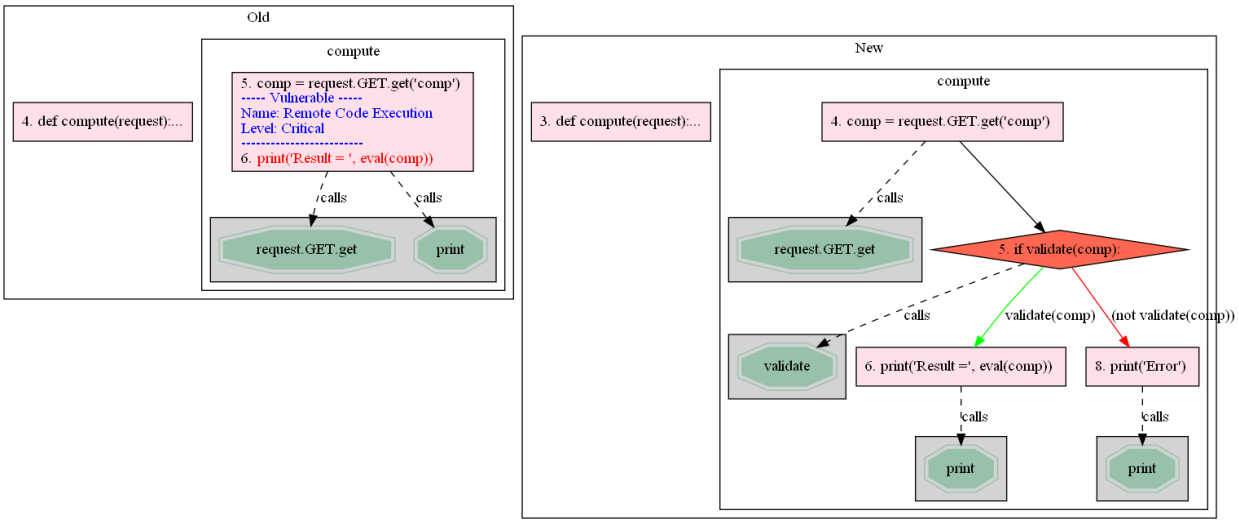


Рисунок 2.10 – приклад роботи програми, на якому виявлено вразливість до віддаленого виконання команд

Висновки

Розроблений додаток підтверджує припущення, що за допомогою порівняння вихідних кодів нової та старої версій програми можливо отримати потенційні вразливості, які наявні в старій версії, що також надає змогу використання отриманої інформації для генерування експлойтів в майбутньому. Було розглянуто одні з найпоширеніших вразливостей – XXE, небезпечну десереалізацію та різні види ін’єкцій, а саме SQL, OS та ін’єкція команд, і продемонстровано їх виявлення та відображення за допомогою реалізованої програми.

Для досягнення цілі програма будувала абстрактні синтаксичні дерева, графи потоку управління двох версій програми, знаходила їх різницю, та вибудовувала шляхи змінних, в яких знаходила вразливості за допомогою аналізу помічених даних відповідно до створеної бази вразливостей.

Роботу програми можливо покращити за допомогою підтримки ООП під час аналізу помічених даних та адаптування підтримки поширених фреймворків, наприклад flask або django. Таке рішення також можливо переписати для аналізу інших мов програмування, окрім python, змінивши процес утворення абстрактного синтаксичного дерева.

Список використаної літератури

[1] What is Python? [Електронний ресурс] – Режим доступу до ресурсу: https://www.python.org/doc/essays/blurb/.

[2] Alternative Python Implementations [Електронний ресурс] – Режим доступу до ресурсу: https://www.python.org/download/alternatives/.

[3] Python data model [Електронний ресурс] – Режим доступу до ресурсу: https://docs.python.org/3/reference/datamodel.html#the-standard-type-hierarchy.

[4] Robert H. Is Python pass-by-reference or pass-by-value? [Електронний ресурс] / Heaton Robert. – 2014. – Режим доступу до ресурсу: https://robertheaton.com/2014/02/09/pythons-pass-by-object-reference-as-explained-by-philip-k-dick/.

[5] OWASP Python Security wiki [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/ebranca/owasp-pysec/wiki.

[6] Compilers, principles, techniques, and tools / Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, 2007. 2nd ed.

[7] P.D. Terry. Compilers and Compiler Generators – 1996. С. 23.

[8] Static Code Analysis [Електронний ресурс] – Режим доступу до ресурсу: https://owasp.org/www-community/controls/Static_Code_Analysis.

[9] Ball T. The Concept of Dynamic Analysis / Thomas Ball. – С. 1–3.

[10] Heribertus Y. Web Application Vulnerability Detection Using Taint Analysis and Blackbox Testing / Yulianton Heribertus. – 2020. – С. 3–4.

[11] AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks / William G.J. Halfond, Alessandro Orso – 2005. С. 1-2.

[12] Psycopg documentation [Електронний ресурс] – Режим доступу до ресурсу: https://www.psycopg.org/docs/cursor.html.

[13] Automatic Patch-Based Exploit Generation is Possible: Techniques and Implication / D.Brumley, P. Poosankam, D. Song, J. Zheng. – 2008.

[14] Designing and Implementing Control Flow Graph for Magic 4th Generation Language / Richárd Dévai , Judit Jász , Csaba Nagy , Rudolf Ferenc. – 2014. C. 4-5.

[15] Frances E. Allen / Control Flow Analysis – 1970. C. 1-3.

[16] Grapviz library [Електронний ресурс] – Режим доступу до ресурсу:

Додатки

Додаток А

Приклад обробки складених конструкцій під час побудови ГПУ

```
def visit_If(self, node: If) -> None:
    self.add_node_to_class(node)
    if self.current_block.statements:
        cond_block = self.new_block()
        cond_block.add_statement(node)
        self.current_block.add_exit(cond_block)
        self.current_block = cond_block
    else:
        self.current_block.add_statement(node)
    if isinstance(node.test, ast.Compare) or isinstance(node.test, ast.Call):
        self.visit(node.test)
    # Create a new block for the body of if.
    if_block = self.new_block()
    self.current_block.add_exit(if_block, node.test)
    # Create a block for the code after if.
    after_block = self.new_block()
    if node.orelse:
        else_block = self.new_block()
        self.current_block.add_exit(else_block)
        self.current_block = else_block
        # Visit the children of the else
        for child in node.orelse:
            self.visit(child)
        self.current_block.add_exit(after_block)
    else:
        self.current_block.add_exit(after_block)
    self.current_block = if_block
    for child in node.body:
        self.visit(child)
    self.current_block.add_exit(after_block)
    self.current_block = after_block
```

Додаток Б

Код функцій, що перевіряють наявність старого виразу в новій версії

```
def node_exists(self, node: ast.AST, node_list: [ast.AST]):
        for listed_node in node_list:
            if self.nodes_are_equal(node, listed_node):
                break
        else:
            if node not in self.not_existing_nodes:
                from .CFGFinder import CFGFinder
                self.not_existing_nodes.append((node,
CFGFinder.extract_variable(node, True)))

def nodes_are_equal(self, node1, node2):
    if issubclass(type(node1), ast.AST) or isinstance(node1, ast.AST):
        if type(node1) == type(node2):
            for key, value in vars(node1).items():
                if key in ["lineno", "col_offset", "end_lineno", "end_col_offset", 'ctx']:
                    continue
                if not self.nodes_are_equal(value, getattr(node2, key, None)):
                    return False
            return True
        else:
            return False
    elif isinstance(node1, list) and isinstance(node2, list):
        return all([self.nodes_are_equal(n1, n2) for n1, n2 in zip(node1, node2)])
    return node1 == node2
```

Додаток В

Код функції, що будує різність функцій старої та нової версії програми.

```
def build_difference_functions(self):
    for function_name in self.patched_cfg.functioncfgs:
        try:
            function_comparator =
self.compare_functions(self.vuln_cfg.functioncfgs[function_name],
                                            self.patched_cfg.functioncfgs[function_name])
            if len(function_comparator.not_existing_nodes) == 0:
                print("Functions are the same", function_name)
            else:
                self.not_existing_functions.append((function_name,
function_comparator))
        # This function does not exist in vuln cfg
        except KeyError:
            print("Function was deleted")
```

Додаток Г

Код функції, що перевіряє наявність вразливості у виразі

```
def expression_contains_vulnerability(self, variables, usage_node):
    expression_method: str = astor.to_source(usage_node[0]).lower()
    try:
        real_value = astor.to_source(usage_node[1].get_last_statement()).lower()
    except AttributeError:
        real_value = usage_node[1].get_last_statement()
    for vulnerability_name in vulnerabilities_with_sanitizers:
        for vulnerability_method in
vulnerabilities_with_sanitizers[vulnerability_name]["source"]:
            vulnerable = False
            statement = None
            if isinstance(vulnerability_method, list):
                if CFGFinder.find_with_split(expression_method, vulnerability_method):
                    statement = usage_node[0]
                    vulnerable = True
                elif CFGFinder.find_with_split(real_value, vulnerability_method):
                    statement = usage_node[1].get_last_statement()
                    vulnerable = True
            else:
                if vulnerability_method in expression_method:
                    statement = usage_node[0]
                    vulnerable = True
                elif vulnerability_method in real_value:
                    statement = usage_node[1].get_last_statement()
                    vulnerable = True
            if vulnerable:
                # this statement is vulnerable
                existing_vulnerability =
Vulnerability(trigger_method=vulnerability_method,
                                              expression=statement,
                                              level=

vulnerabilities_with_sanitizers[vulnerability_name][
                                                  "level"],
                                              name=vulnerability_name)
                # find any sanitizers, if they exist
                for sanitizer in
vulnerabilities_with_sanitizers[vulnerability_name]["sanitizers"]:
                    if isinstance(sanitizer, list):
                        all_sanitizers = list(
```

```
                    map(lambda x: x in real_value or x in expression_method,
sanitizer))
                if len(all_sanitizers) > 0 and all(all_sanitizers):
                    existing_vulnerability.vulnerable =
VulnerabilityType.SANITIZED
                    existing_vulnerability.sanitizers.append(sanitizer)
                    break
                elif sanitizer in real_value or sanitizer in expression_method:
                    existing_vulnerability.vulnerable =
VulnerabilityType.SANITIZED
                    existing_vulnerability.sanitizers.append(sanitizer)
                    break
            self.vulnerabilities.append(existing_vulnerability)
```

Додаток Ґ

База вразливостей, що була перевірена.

```
vulnerabilities_with_sanitizers = {
   "OS injection": {
      "sanitizers": ["shlex.quote("],
      "source": ["os.popen(", "subprocess.run(", "os.spawnl("],
      "level": "Critical"
   },
   "Deserialization attack": {
      "sanitizers": [],
      "source": ["pickle.loads("],
      "level": "Critical"
   },
   "XXE attack": {
      "sanitizers": [["resolve_entities=False", "no_network=True"]],
      "source": ["etree.parse("],
      "level": "Moderate"
   },
   "Temporary file creation": {
      "sanitizers": [],
      "source": ["mktemp("],
      "level": "Moderate"
   },
   "SQL Injection": {
      "sanitizers": [],
      "source": [["select", "%s", "%"], ["from","%s","%"]],
      "level": "Critical"
   },
   "Remote Code Execution": {
      "sanitizers": [],
      "source": ["eval("],
      "level": "Critical"
   }
}
```

Додаток Д

Приклад старої версії, вразливої до OS ін’єкції, і нової, в якій вразливість була нейтралізована.

Стара:

```
import sys
import os
def ping(ip):
    address = ip
    cmd = "ping -c 1 %s" % address
    os.popen(cmd)
```

Нова:

```
import sys
import os
import shlex
def ping(ip):
    address = shlex.quote(ip)
    cmd = "ping -c 1 %s" % address
    os.popen(cmd)
```

Додаток Е

Приклад старої версії, вразливої до SQL ін’єкції, і нової, в якій вразливість була нейтралізована.

Стара:

```
def hello(request):
    id = request.GET.get("id", "")
    cursor = connection.cursor()
    cursor.execute("SELECT username FROM auth_user WHERE id=%s" % id)
    row = cursor.fetchone()
    return HttpResponse("Hello %s" % row[0])
```

Нова:

```
def hello(request):
    id = request.GET.get("id", "")
    cursor = connection.cursor()
    cursor.execute("SELECT username FROM auth_user WHERE id=:id", {"id": id})
    row = cursor.fetchone()
    return HttpResponse("Hello %s" % row[0])
```

Додаток Є

Приклад старої версії, вразливої до XXE, і нової, в якій вразливість була нейтралізована.

Стара:

```
parser = etree.XMLParser(resolve_entities=True)
tree1 = etree.parse('ressources/xxe.xml', parser)
root1 = tree1.getroot()
```

Нова:

```
parser = etree.XMLParser(resolve_entities=False, no_network=True)
tree1 = etree.parse('ressources/xxe.xml', parser)
root1 = tree1.getroot()
```

Додаток Ж

Приклад старої версії, вразливої до небезпечної десереалізації, і нової, в якій вразливість була частково нейтралізована.

Стара:

```
HOST = "0.0.0.0"
PORT = 7575

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen()
connection = s.accept()
received_data = connection[0].recv(1024)
data = pickle.loads(received_data)
```

Нова:

```
HOST = "0.0.0.0"
PORT = 7575

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen()
connection = s.accept()
received_data = connection[0].recv(1024)
data_splitted = received_data.split(' ')
byte_key = binascii.unhexlify("Public-key")
new_digest = hmac.new(byte_key, data_splitted[1], hashlib.sha1).hexdigest()
if data_splitted[0] != new_digest:
    print('Integrity check failed')
else:
    unpickled_data = pickle.loads(data_splitted[1])
```

Додаток З

Приклад старої версії, вразливої до віддаленого виконання команд, і нової, в якій вразливість була нейтралізована.

Стара:

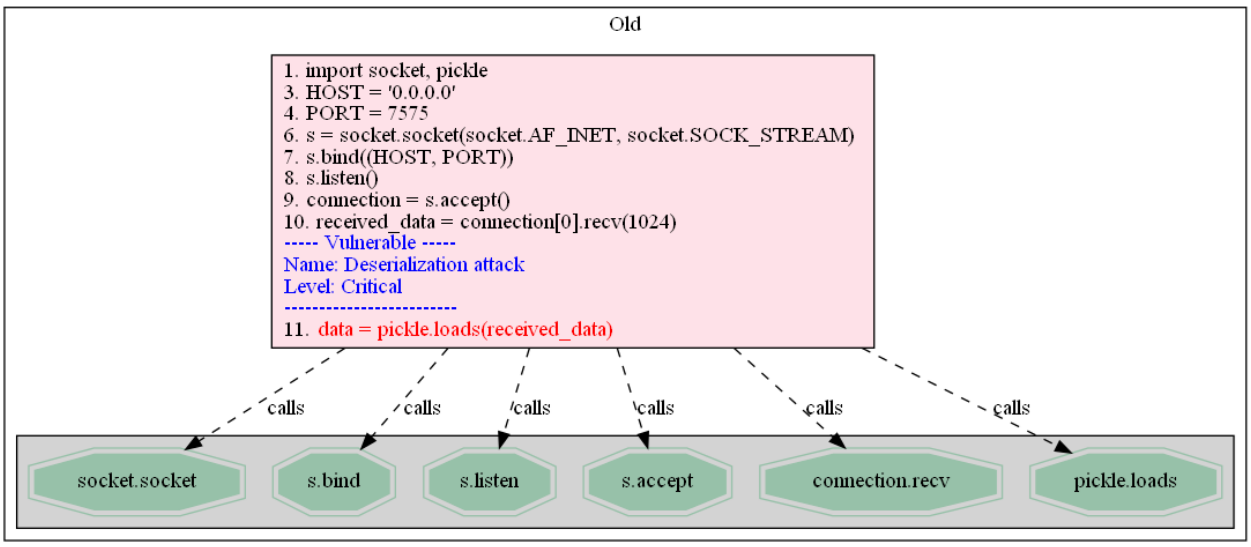
```
def compute(request):
    comp = request.GET.get("comp")
    print("Result = ", eval(comp))
```

Нова:

```
def compute(request):
    comp = request.GET.get("comp")
    if validate(comp):
        print("Result =", eval(comp))
    else:
        print("Error")
```

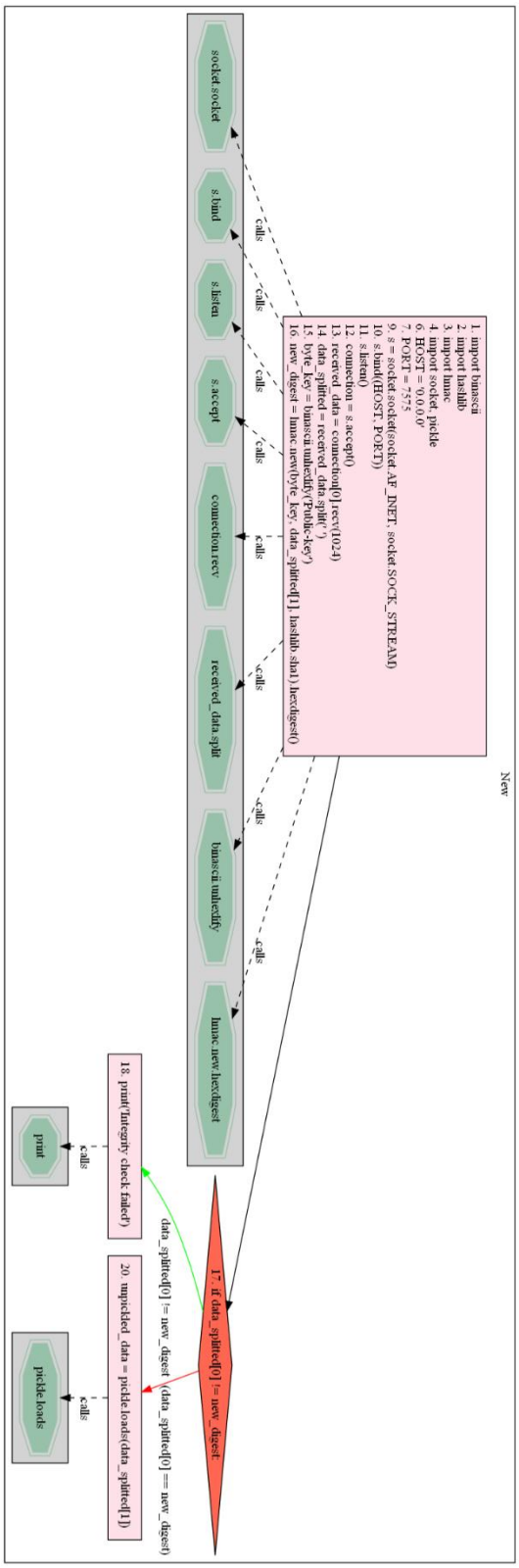
Додаток И

Відображення старої версії програми, яка вразлива до атаки десереалізацією (код з додатку Ж)



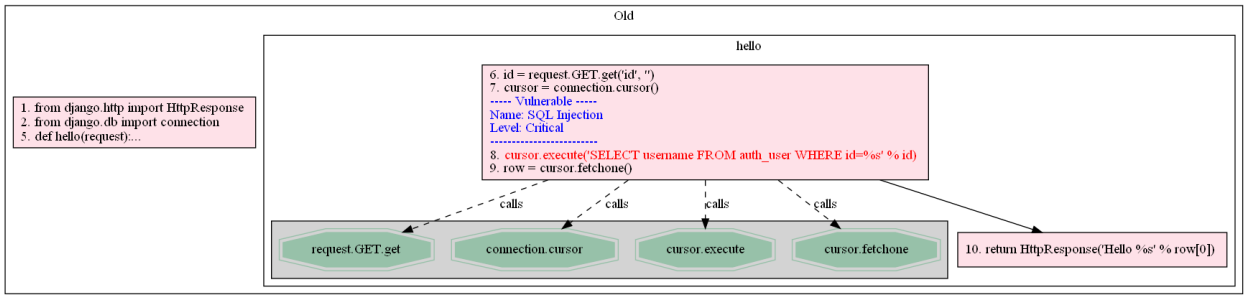
Додаток І

Відображення нової версії, в якій частково виправлено вразливість до десереалізації (код з додатку Ж)



Додаток Ї

Відображення старої версії програми, яка вразлива до атаки SQL ін'єкції (код з додатку Е)



Додаток Й

Відображення нової версії програми, в якій нейтралізовано можливість SQL ін'єкції (код з додатку Е)