Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя
(повне найменування вищого навчального закладу)
Факультет комп'ютерно-інформаційних систем і програмної інженерії
(назва факультету)
Кафедра кібербезпеки
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(освітній рівень)

на тему: "Дослідження стійкості методів обфускації JS коду"

Виконав: студент IV курсу, групи СБ-42

Спеціальності:
125 «Кібербезпека»
(шифр і назва напряму підготовки, спеціальності)
Хілініч В.Д.
підпис (прізвище та ініціали)

Керівник Козак Р. О.
підпис (прізвище та ініціали)

Нормоконтроль Дроздова Т.В.
підпис (прізвище та ініціали)

Завідувач кафедри Загородна Н.В.
підпис (прізвище та ініціали)

Рецензент
підпис (прізвище та ініціали)

м. Тернопіль – 2025

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра кібербезпеки
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Загородна Н.В.
(підпис) (прізвище та ініціали)
«__» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Бакалавр
(назва освітнього ступеня)

за спеціальністю 125 Кібербезпека
(шифр і назва спеціальності)

Студенту Хілінічу Валерію Денисовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження стійкості методів обфускації JS коду

Керівник роботи Козак Руслан Орестович, к. т. н.,
доцент кафедри КБ
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від 02.05.2025 року № 4/7-361

2. Термін подання студентом завершеної роботи 17.06.2025

3. Вихідні дані до роботи Методи та засоби обфускації JS-коду, вихідний код програмного забезпечення, який необхідно обфускувати

4. Зміст роботи (перелік питань, які потрібно розробити)
Проаналізувати актуальність обфускації JavaScript-коду для забезпечення безпеки веб-застосунків та використання заплутування коду для шкідливого ПЗ
Проаналізувати методи та підходи до обфускації JS-коду
Провести порівняльний аналіз ефективності методів обфускації JS-коду
Безпека життєдіяльності, основи охорони праці
Висновки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
Мета та завдання дослідження. Актуальність обфускації JavaScript-коду. Атаки на обфускований JS-код. Зловмисне використання обфускації. Методи обфускації JS-коду. Методи зміни структури коду. Методи перетворення змінних. Методи з використанням пунктуаційних перетворень. Використані обфускатори для аналізу стійкості методів. Критерії оцінки ефективності методів обфускації. Порівняльна таблиця використаних обфускаторів. Обрано оптимальний обфускатор — JScrambler. Результат роботи обфускатора JScrambler. Висновки

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи хорони праці	Мариненко С.Ю., к. т. н., доцент кафедри МТ		

7. Дата видачі завдання 29.01.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	29.01 – 30.01	Виконано
2.	Підбір джерел для аналізу методів обфускації JS-коду	31.01 – 05.02	Виконано
3.	Опрацювання джерел в галузі дослідження	06.02 – 20.02	Виконано
4.	Провести аналіз методів обфускації JS-коду	22.02 – 12.03	Виконано
5.	Проведення порівняльного аналізу ефективності методів обфускації JS-коду	15.03 – 25.03	Виконано
6.	Налаштування системи автоматичного блокування IP зловмисника	25.02 – 10.04	Виконано
7.	Оформлення розділу «Аналіз технічного завдання»	10.02 – 05.03	Виконано
8.	Оформлення розділу «Методи та підходи обфускації JS-коду»	26.03 – 11.04	Виконано
9.	Оформлення розділу «Порівняльний аналіз ефективності методів обфускації»	12.04 – 20.05	Виконано
10.	Виконання завдання до підрозділу «Безпека життєдіяльності, основи охорони праці»	26.05 – 01.06	Виконано
11.	Оформлення кваліфікаційної роботи	02.06 – 07.06	Виконано
12.	Нормоконтроль	10.06 – 11.06	Виконано
13.	Перевірка на плагіат	12.06 – 13.06	Виконано
14.	Попередній захист кваліфікаційної роботи	16.06 – 17.06	Виконано
15.	Захист кваліфікаційної роботи	27.06.2025	

Студент _____ Хілініч В.Д.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Козак Р.О.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дослідження стійкості методів обфускації JS коду // Кваліфікаційна робота ОР «Бакалавр» // Хілініч Валерій Денисович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра кібербезпеки, група СБ-42 // Тернопіль, 2025 // С. – 61, рис. – 23, табл. – 1 , форм. – 4 , додат. – - .

КЛЮЧОВІ СЛОВА: JavaScript, обфускація, захист коду, реверс-інжиніринг, деобфускація, кібербезпека.

Кваліфікаційну роботу присвячено аналізу та оцінці стійкості методів обфускації JavaScript-коду до атак реверс-інжинірингу. У рамках дослідження здійснено класифікацію сучасних методів обфускації, проведено порівняльний аналіз обфускаторів і розглянуто особливості реалізації кожного з них. Визначено практичну ефективність окремих методів та виявлено їхні вразливості.

У роботі проведено тестування засобів обфускації, визначено рівень захисту, який вони забезпечують, а також проаналізовано вплив обфускації на продуктивність і функціональність коду. Результати дослідження дали змогу сформулювати практичні рекомендації щодо вибору оптимальних комбінацій методів обфускації для підвищення безпеки клієнтського JavaScript.

ABSTRACT

Research on the resilience of JavaScript code obfuscation methods // Thesis of educational level "Bachelor"// Khilinich Valerii // Ternopil Ivan Puluj National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Cybersecurity, group СБ-42 // Ternopil, 2025 // P. – 61, fig. - 23, tab. – 1, formulae – 4, added. – -.

Keywords: JavaScript, obfuscation, code protection, reverse engineering, deobfuscation, cybersecurity.

The qualifying undergraduate work is devoted to the analysis and assessment of the resistance of JavaScript code obfuscation methods to reverse engineering attacks. As part of the study, a classification of modern obfuscation methods was carried out, a comparative analysis of obfuscators was conducted, and the features of the implementation of each of them were considered. The practical effectiveness of individual methods was determined, and their vulnerabilities were identified.

The qualification work tested obfuscation tools, determined the level of protection they provide, and analyzed the impact of obfuscation on code performance and functionality. The results of the study made it possible to formulate practical recommendations for choosing the optimal combinations of obfuscation methods to increase the security of client JavaScript.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

JS	—	JavaScript
IDE	—	Integrated Development Environment
ПЗ	—	Програмне забезпечення
CFF	—	Control Flow Flattening
CFG	—	Control Flow Graph

ВСТУП

Актуальність теми. У сучасному цифровому середовищі JavaScript відіграє одну з ключових ролей у розробці веб-застосунків, кросплатформених рішень, мобільних інтерфейсів та клієнтських частин систем. Оскільки JS-код виконується безпосередньо у браузері, він залишається доступним у відкритому вигляді, що створює низку серйозних ризиків: порушення авторських прав, викрадення унікальної бізнес-логіки, несанкціоноване втручання у роботу системи, зворотний інжиніринг, копіювання або компрометацію програмного забезпечення. Одним із сучасних підходів до мінімізації цих ризиків є обфускація JavaScript-коду — процес навмисного перетворення вихідного коду у вигляд, складний для сприйняття людиною, без зміни його функціональності. Такий підхід значно ускладнює реверс-інжиніринг і забезпечує захист логіки програми.

Втім, обфускація застосовується не лише з легітимною метою захисту — вона широко використовується у зловмисних цілях: для приховування шкідливих функцій, маскування вірусного або шпигунського коду, обходу антивірусного захисту та ускладнення виявлення атак. Це створює додаткові виклики для фахівців з кібербезпеки, аналітиків шкідливого ПЗ і дослідників безпечного програмування.

У зв’язку з цим, дослідження стійкості методів обфускації є актуальним — як з точки зору захисту легітимного коду, так і в контексті протидії зловживанню цими технологіями з боку кіберзлочинців.

Метою дослідження є аналіз і оцінка стійкості сучасних методів обфускації JavaScript-коду до зворотного інжинірингу, а також виявлення їх сильних і слабких сторін з точки зору захисту логіки веб-застосунків від несанкціонованого доступу та шкідливого використання.

Для досягнення поставленої мети в роботі визначено такі **завдання дослідження**:

– проаналізувати теоретичні основи обфускації JavaScript-коду, а також основні типи атак і підходи до декодування обфускованого коду;

– класифікувати основні методи обфускації, що використовуються в сучасних інструментах захисту JS;

– провести практичне тестування та порівняльний аналіз ефективності різних засобів обфускації;

– оцінити стійкість обфускованого коду до методів реверс-інжинірингу та аналізу;

– дослідити потенційні ризики зловмисного використання обфускації для приховування шкідливого функціоналу;

– сформулювати практичні рекомендації щодо вибору та застосування методів обфускації у розробці безпечного програмного забезпечення.

Об’єктом дослідження є процеси захисту програмного коду JavaScript у клієнтських веб-застосунках засобами обфускації та аналізу їх стійкості до зворотного інжинірингу.

Предметом дослідження є методи обфускації JavaScript-коду та їх стійкість до зворотного інжинірингу, деобфускації й атак реверс-інжинірингу.

Практичне значення одержаних результатів полягає у можливості використання досліджених методів обфускації JavaScript-коду для підвищення рівня захисту клієнтських веб-застосунків від зворотного інжинірингу, копіювання та несанкціонованого втручання. Результати роботи можуть бути корисними для розробників програмного забезпечення, спеціалістів із кібербезпеки та аналітиків шкідливого ПЗ при виборі або створенні ефективних механізмів заплутування коду.

РОЗДІЛ 1 АНАЛІЗ ТЕХНІЧНОГО ЗАВДАННЯ

1.1 Актуальність обфускації JavaScript-коду для забезпечення безпеки веб-застосунків

Однією з найпоширеніших мов програмування є JavaScript (JS), адже поріг входження для вивчення цієї технології достатньо невеликий [1]. У відкритих джерелах існує достатньо багато ресурсів і документації, що дозволяє легко освоїти цю мову програмування. Код зазвичай виконується безпосередньо на боці клієнта, якщо мова йде про Front-end розробку, де найбільше застосовують JavaScript. Це означає, що JS веб-додатки мають свій вихідний код, видимий і простий для читання та/або копіювання третіми сторонами.

Обфускація JavaScript є спробою вирішити цю проблему. Обфускація коду означає перетворення зрозумілого вихідного JavaScript на нечитаючу (і важку для аналізу та/або модифікації) версію скрипта, зберігаючи повну його функціональність [**Помилка! Джерело посилання не знайдено.**2]. Наведемо приклад на основі простої функції для виводу вітального повідомлення в консоль браузера, а саме: "*Hello, dear Valerii. How are you?*" (див. рисунок 1.1).

```
function sayHello(name) {
    console.log('Hello, dear ' + name + '. How are you?');
}

function _0x3f21(_0x4c9a7a) {
  this['console']['log']('Hello, dear ' + _0x4c9a7a + '. How are you?');
}
```

Рисунок 1.1 – Приклад JS коду до та після обфускації для виводу вітального повідомлення

Після обфускації імена змінних та функцій замасковані (`sayHello` перейменовано на `_0x3f21`, а `name` на `_0x4c9a7a`), загальний вигляд коду функції також змінений. . Проте логіка виконання не змінюватиметься і достатньо викликати `_0x3f21('Valerii')` і буде виведено в консоль те ж саме повідомлення.

Тому основною метою обфускації є захист коду від прямого сприйняття та аналізу, зберігаючи при цьому його функціональність. Нижче розглянуті основні причини використання обфускації в JavaScript пов’язані як із безпекою, так і з оптимізацію рішень:

Обфускація приховує алгоритми та реалізацію функцій, які можуть бути важливими вхідними даними для конкурентів, які намагаються відтворити власний процес розробки, або інженерів з поганими намірами. Наприклад, якщо ключові алгоритм реалізовано на стороні клієнта (шифрування або процедура перевірки ліцензії), обфускація коду допомагає захистити його від копіювання[2]. Також це допомагає впроваджувати ліцензійні обмеження, знижуючи можливості користувачів видалити перевірки ліцензії або обійти обмеження пробного періоду ПЗ [3].

Читання або відлагоджування добре заплутаного коду є надзвичайно складним завданням, тож хакери та аудитори безпеки під час виявлення слабких місць системи стикаються з величезними труднощами. Зловмисникам стає набагато складніше інтерпретувати логіку програми, шукати "дірки" в безпеці та розробляти вразливості [2]. Як зазначено в одному з звітів Akamai, обфускація коду на стороні клієнта не розкриває його справжньої природи до моменту виконання, потенційно приховуючи шкідливі функції від автоматизованих систем виявлення. Тобто його “справжній намір” може стати зрозумілим лише тоді, коли він буде виконуватись [4].

Багато антивірусних та захисних рішень виявляють шкідливі скрипти за сигнатурами та знайомими шаблонами коду. Обфускація може потенційно замаскувати небезпечний код, уникнувши спрацювання детекторів сигнатур [2]. Наприклад, шкідливий скрипт, який створює фальшиву форму оплати на сайті, може бути зашифрований і розпакований лише під час виконання. Таким чином, це ускладнить його розпізнання як відомий зразок шкідливого коду. Хоча сама обфускація не є "шкідливим" трюком, її здатність приховувати наміри коду робить її привабливою як для кіберзлочинців, так і для захисту легітимних додатків.

Деякі методи обфускації одночасно можуть виконувати роль мініфікатора коду – очищення коду від непотрібного об’єму коду, що не впливає на виконання інтерпретатором та не несе ніякої інформації [2]. Таким чином, розмір JS-файлів може помітно зменшитись та прискорити їх завантаження браузером і покращити метрики застосунку.

Однак варто зазначити, що обфускація має свої недоліки та ризики [2], а саме:

– надмірна обфускація може ускладнити підтримку коду, адже бувають нетривіальні випадки, де потрібно відлагоджувати код на реальному додатку;

– зменшення швидкості виконання скриптів через необхідність більших обчислюваних потужностей;

– обфускація не є гарантією безпеки коду, адже досвідчений аналітик або хакер із спеціальними інструментами спроможний розплутати навіть дуже заплутаний код.

Тому рекомендується використовувати обфускацію обережно та тільки в тих випадках, де це має сенс, доповнюючи її іншими засобами захисту, такими як: контроль доступу, шифрування даних, серверні перевірки, тощо [3].

1.2 Підходи зловмисників до аналізу обфускованого JavaScript-коду

Для хакерів або спеціалістів з інформаційної безпеки обфускований код JavaScript є додатковим рівнем захисту. Однак професійні реверс-інженери використовують різноманітні інструменти та техніки для деобфускації, щоб відновити обфускований код до його оригінального змісту. Хоча обфускація може ускладнити або затримати атаки, вона не може їх повністю зупинити [4]. Інші загрози кібербезпеки інформаційних систем розглянуто в [6, 7, 8, 9].

У цьому підрозділі за допомогою реальних прикладів розглянуто, як хакери аналізують і ламають обфускований код. Перш за все, хакери намагаються визначити, з чим мають справу. Коли вони стикаються з підозрілим фрагментом коду, вони визначають чи це є наслідком обфускації.

Такий код має характерні ознаки:

– маленькі функції, які самі себе генерують;

– незрозумілі імена змінних на кшталт `_0x3f21`;

– довгі рядки, заповнені символами Base64 або шістнадцятковими числами;

– вирази, що містять зашифровані дані разом з викликом eval.

Аналітик може навіть ідентифікувати використаний інструмент обфускації та розшифрувати такий зразок (наприклад, масив незрозумілих констант або функцію розшифровки), вивчивши характерні шаблони, залишені певними обфускаціями. Розуміння сервісу чи методу обфускації, що використовувався, дозволяє вибрати відповідні контрзаходи [5].

Друге — вони перетворюють вихідний код у зручну для аналізу форму. Оскільки обфускація зазвичай перетворює первинний код у один великий рядок, початковий етап зазвичай передбачає форматування (beautify) коду [2]. Це можна зробити за допомогою вбудованих у браузер інструментів для розробників (опція Pretty Print у Chrome DevTools) або сторонніх форматерів коду. Форматування не дає абсолютно зрозумілого коду, але значно полегшує подальші дії, розкриваючи структуру скрипта. Спеціалісти можуть потім завантажити ці файли в свої робочі середовища (IDE або пісочниці) для детального аналізу [5].

Третє, хакери використовують напівавтоматизовані ПЗ та скрипти для деобфускації. Вони значно зменшують трудомісткість розплутування коду. Наприклад, ці утиліти в загальному замінять безглузді фрагменти тексту на нормальні змістовні імена і розшифровують `const` чи рядкові літерал. Інтернет-сервіси, такі як JSNice, можуть аналізувати обфускований код та відображати більш кращі імена змінних на основі їх використання. Інший інструмент, de4js, автоматично обходить популярні схеми обфускації (наприклад, вставки на кшталт "`eval`" у вихідний код), замінює вбудовані приховані рядки та відновлює умови логіки [2]. Ці інструменти значно покращують ефективність реверс-інженерії — замість того, щоб вручну перейменовувати всі змінні. Варто також згадати, що навіть найпотужніші деобфускатори не можуть повністю відновити код. На практиці цей інструмент може надати лише спрощену версію коду, а решта частин все ще мають бути вручну проаналізовано шляхом ретельного вивчення логіки програми.

Четверте, застосування динамічного аналізу та налагодження коду. Якщо статичне спрощення залишає багато невідомих подробиць, хакери не мають вибору, окрім як запускати обфускований сценарій у контрольованому середовищі. Використовуючи інструменти розробника браузера або дебагери, вони можуть покроково йти через код, спостерігаючи, як змінні змінюють свої значення, слідкують за виходами функцій тощо. Один із методів — запускати скрипт з модифікованими функціями `eval`/`Function`, що перехоплюють декомпільований код, створений обфускатором, для перевірки перед виконанням [2]. Це дозволяє, наприклад, отримати розшифровані рядки або скомпільований код, що був прихований обфускатором.

У спільноті фахівців безпеки існує багато випадків, коли обфускований код вдавалося розібрати шляхом комбінування статистичного і динамічного аналізу. Зокрема, у блозі CyberSecureFox продемонстровано поетапну деобфускацію [2]: спочатку JS-код відформатували, потім відновили логіку циклів, замінили індекси масивів на безпосередні значення рядків і, вкінці, отримали зрозумілу версію скрипта, що виводив фразу *"Hello World"* замість незрозумілого набору символів. У цьому простому прикладі очевидно, що професіонал може побороти будь-яку обфускацію за допомогою відповідних інструментів (форматери, декодери, дебагери) і досвіду.

Отже, для хакерів обфускація є швидше викликом, ніж стіна захисту. Вони вдосконалюють свої методи на кожному кроці, від простих дій, таких як форматування та токенізація, аж до розробки спеціалізованих інструментів для розгортання шаблонів обфускації. Також існують безліч онлайн-інструментів з деобфускації JS, а саме плагіни для браузера, що можуть показати оригінальний код через DevTools, тощо. Все це ясно вказує на те, що через обфускацію захист коду є відносним. Вона ускладнює несанкціонований аналіз, але не гарантує повну невразливість проти досвідченого хакера.

1.3 Використання обфускації для шкідливого ПЗ

Обфускація JavaScript-коду має подвійний характер застосування у сфері безпеки. З одного боку, вона рекомендована експертами як засіб захисту коду від реверс-інжинірингу — наприклад, OWASP включає обфускацію до практик для ускладнення аналізу коду зловмисниками [10]. З іншого боку, ті ж самі техніки можуть бути використані в поганих цілях: нападники можуть заплутати шкідливий код, щоб приховати його справжню функцію та уникнути виявлення. Таким чином, було зафіксовано неетичне застосування обфускації для приховування збору даних, вбудованих бекдорів або вразливостей у програмному забезпеченні [10].

Хоча сама по собі обфускація не є шкідливою (основна проблема полягає в тому, що 0,5% популярних веб-сайтів використовують її для захисту свого клієнтського коду від зовнішнього аналізу), вона часто використовується кіберзлочинцями як зброя. Дослідження показують, що щонайменше чверть шкідливих JS-скриптів (включаючи фішинг-сторінки, dropper-скрипти, шахрайські сайти та майнери) використовують техніки обфускації, щоб уникнути виявлення антивірусами та аналізаторами [11].

Обфускація часто використовується для здійснення таких атак:

– обхід поштових фільтрів;

– маскування шкідливого вмісту на веб-сторінках та у файлах;

– зловмисні скрипти та розширення в браузерах.

Зловмисники вбудовують обфускований JavaScript-код у фішингові листи або вкладення, щоб обійти антивірусні та поштові спам-фільтри. Наприклад, одне з досліджень Forcepoint показало хвилю сертифікованих листів (PEC), які містили заплутані скрипти [12]. Завдяки довіреності каналу та шифруванню коду, їм вдалося обійти системи захисту. У такому випадку обфускація ускладнює автоматичне визначення того, чи є у файлі шкідливі payload’и, та збільшує ймовірність успішного зараження користувача.

Атаки типу drive-by download часто включають обфускований код, що приховує шкідливу логіку за, на перший погляд, безпечним вмістом. Прикладом

цього є випадок, коли банківські трояни, такі як Astaroth, поширюються шляхом spear-phishing – жертви отримували ZIP-архів, що містив HTML-документ, який використовував `mshta.exe` для виконання обфускованого JavaScript, непомітно встановлюючи шкідливе ПЗ. Так само кіберзлочинці приховують шкідливі скрипти у начебто нешкідливих файлах або на зламаних сайтах — код може бути попередньо закодований (наприклад, у Base64) і вбудований так, що його важко виявити без спеціального аналізу; лише розшифрування показує справжній зловмисний сценарій [13].

Маскування шкідливої поведінки часто використовується у розширеннях браузерів та скриптах. Компанія Google зазначила, що більше 70% шкідливих або політико-порушуючих розширень містили обфускований код. У 2018 році з цієї причини застосунки у всьому магазині додатків Chrome Web Store пішли на заборону обфускації. Наявність заплутаного коду суттєво ускладнило роботу з верифікації для доповнень, що стало і причиною цього кроку.

Команди реагування на інциденти та групи безпеки (CERT) у багатьох місцях регулярно видають такі попередження. Наприклад, CERT-UA задокументувала фішингову атаку, що видавалася за список ув'язнених. У вкладеному файлі формату CHM, що містив JavaScript, який запускало обфускований PowerShell-скрипт, а потім встановлювало шпигунське ПЗ [30]. Цей інцидент ілюструє стандартний приклад: шкідливий код, прихований у вкладенні, виконується непрямим чином. Останні подібні звіти показали, що обфускація безумовно стала інструментом для приховування шкідливого ПЗ в атаках.

Проте іноді така практика підводить зловмисників. Часто трапляється, що обфускатори залишають за собою виразні "відбитки" після себе, за якими групуються і деанонімізуються зразки шкідливого ПЗ. Наприклад, дослідники Akamai виявили, що різні частини шкідливих скриптів (фішингові сторінки, dropper-и, скримінгові коди тощо) фактично мали однакові структури обфускації. Це одразу ж вказувало на спільного джерело їх "упакування" [11]. Визначення унікальних характеристик обфускатора одразу дозволило пов'язати ці приклади разом (розшифрувати їх походження) і спростити подальше

виявлення. Крім того, аналітики можуть використовувати динамічний аналіз — це означає запуск заплутаного скрипту в безпечному обмеженому середовищі, а потім відстеження його поведінки. Там вони можуть виявити його приховані функції (наприклад, кінцеву URL-адресу фішингового сайту або небезпечні системні команди). Іншим прикладом є, замість того щоб вручну дешифрувати довгі рядки в заплутаних скриптах, фахівці можуть використовувати дебагери, щоб одразу побачити реальний вихід скрипту — наприклад, приховані рядки або адреси серверів [12].

Навіть повністю обфускований код не є ідеально прихованим місцем: зрештою він розкриває свою справжню сутність. Проте методи обфускації зловмисників продовжують вдосконалюватися, з'являються більш складні техніки. Відмінним прикладом цього стало відкриття кампанії фішингу в 2025 році, що включала JavaScript-payload, прихований за допомогою невидимих Unicode символів: код виглядав порожнім [15]. Цей новий метод заплутування був вперше публічно описаний дослідниками восени 2024 року, а його швидка поява в реальних атаках (спрямованих на афілійованих осіб комітету США) показує, як швидко нові методи можуть бути підхоплені нападниками. Цей випадок підкреслює, що фішингові сценарії можуть використовувати найновіші методи маскування, які серйозно ускладнюють їх автоматичне виявлення.

Дуже важливо як для захисту, так і для атаки: яку ступінь стійкості демонструє техніка обфускації у зворотному процесу. З одного боку, для розробників легітимного програмного забезпечення більш стійке заплутування означає, що їхній код краще захищений від несанкціонованого аналізу та крадіжки інтелектуальної власності. Однак з іншого боку, для фахівців з кібербезпеки, надмірно стійка обфускація є справжнім викликом: у результаті шкідливий код стає важчим для дешифрування та аналізу, що дає перевагу зловмисникам.

По-перше, оскільки заплутування широко використовується у поширенні шкідливого ПЗ та атаках, це вимагає більше часу ресурсів для аналізу кожного зразка шкідливого ПЗ, що збільшує ймовірність пропустити атаку [11].

По-друге, при дослідженні стійкості методів обфускації важливо збалансувати захист коду від небезпеки з можливістю ефективно виявляти приховані загрози.

З боку засобів захисту вже впроваджуються новітні підходи для боротьби з обфускованим шкідливим ПЗ. У звіті VirusTotal відзначається, що застосування AI під час аналізу скриптів дозволило на 70% покращити виявлення обфускованих шкідливих програм порівняно з традиційними методами [16]. Це підкреслює важливість подолання занадто стійкої обфускацію. Звідси забезпечення надійної обфускації легітимного коду і покращення інструментів деобфускації, здатних виявити шкідливий код – є однаково пріоритетними завданням для сучасної кібербезпеки.

РОЗДІЛ 2 МЕТОДИ ТА ПІДХОДИ ОБФУСКАЦІЇ JS-КОДУ

Проблема приховання логіки коду досить поширена та існують багато різних технік для цього. Також придумують нові авторські методи, що не описані у відкритих джерелах, адже деобфускувати невідомі шаблони досить складно і це потребує значного часу для досягнення мети – приведення заплутаного коду до початкового вигляду, придатного для подальшого аналізу логіки коду. Для досягнення бажаного результату сучасні JavaScript обфускатори використовують різні методи, комбінуючи їх між собою. Це обумовлено тим, що при застосуванні лише одного методу обфускації досить легко визначити, який саме спосіб було використано, що суттєво зменшує час, необхідний для відновлення початкової структури коду. У дані роботі було розлянуто наявні методи обфускації програмного JS-коду із прикладами їх застосування для аналізу складності та принципів роботи. Серед них можна виокремити три основні класифікації, що зображені на рисунку 2.1 [19].

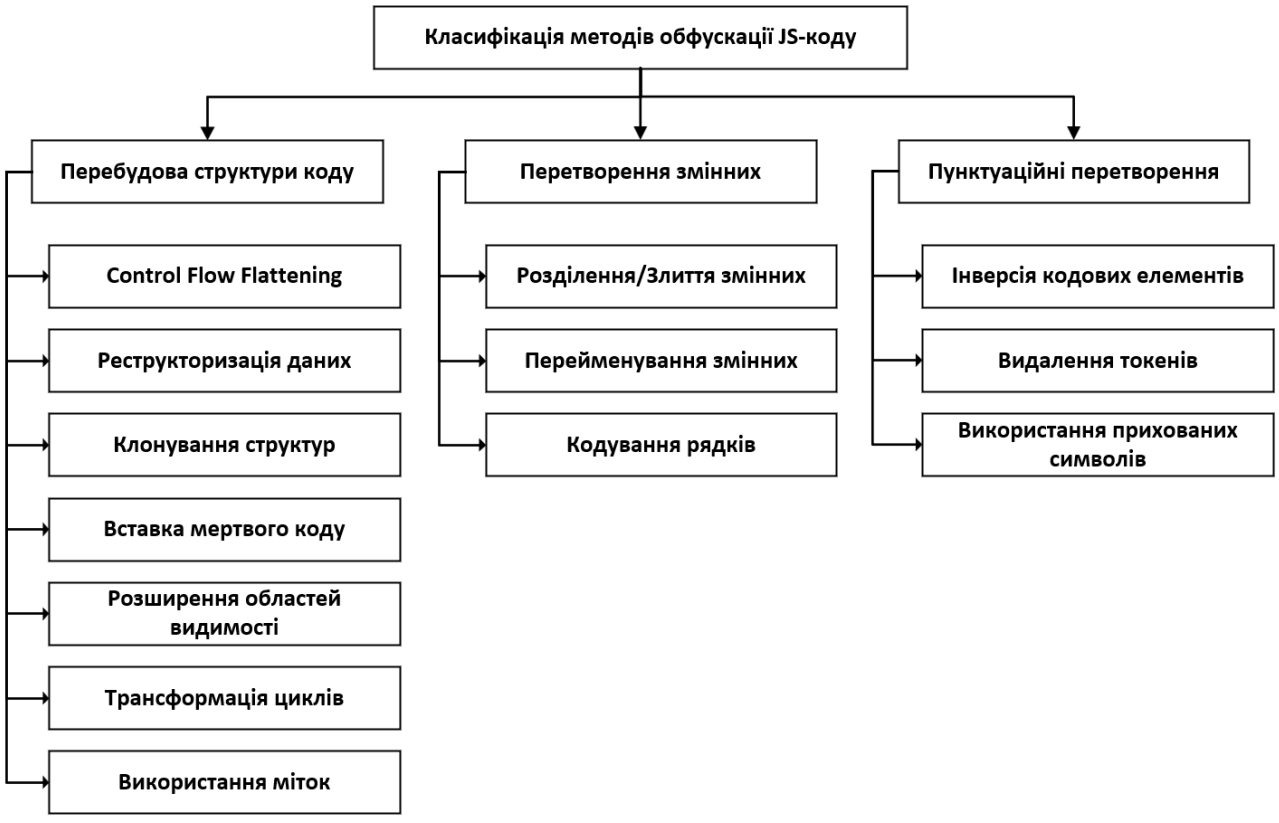


Рисунок 2.1 – Класифікація методів обфускації JavaScript-коду

2.1 Методи, що базуються на зміні структури коду

Обфускація на основі перебудови структури коду це досить обширна група методів, адже це є одним з найкращих підходів, що досить ускладнює зворотну інженерію заплутаного коду та розуміння логіки програми. Вона може змінювати розташування та форму коду без зміни його функціональності. Сюди належать достатньо чимало методів маніпуляції із циклами, функціями, умовними конструкціями, областями видимості та інші.

2.1.1 Вирівнювання контролю потоку

Метод Control Flow Flattening (CFF) є одним із найефективніших, оскільки трансформує усі умовні конструкції, цикли, функції, створюючи один нескінченний цикл `while`, який містить оператор `switch` і виконує роль диспетчера програми, перенаправляючи в потрібну гілку `case` залежно від стану спеціально задекларованого прапорця [17]. Це значно ускладнює аналіз та читання потоку виконання програми, адже складно відстежити усі природні умови конструкцій.

JavaScript виконується послідовно від початку файлу, до його кінця. Тобто кожному виклику функції чи циклу передує певна інша конструкція, і виконання коду виконується поетапно. На рисунку 2.2 подано схематичний приклад роботи цього методу. На лівій стороні показано Control Flow Graph (CFG) до обфускації, де легко зрозуміти умовні оператори та послідовність виконання програми. На правій стороні показано обфускований код, де всі конструкції знаходяться на одному рівні вкладеності та переходять між собою через повернення до початкової вершини й зміну значення прапорця [18].

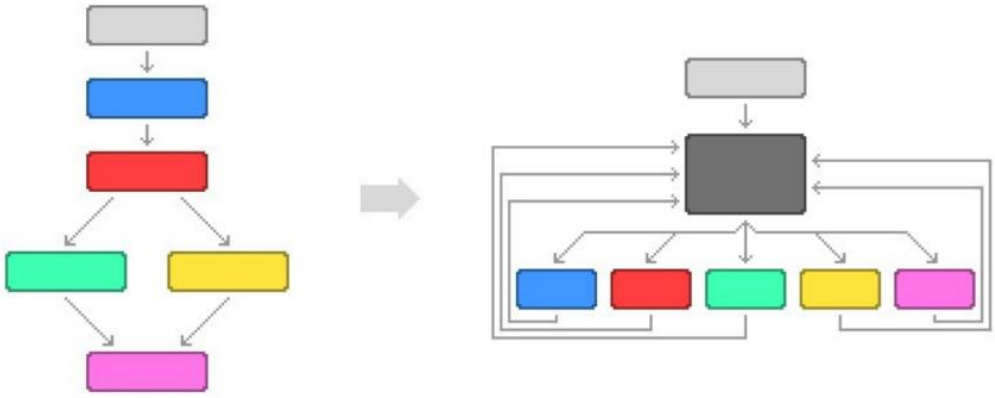


Рисунок 2.2 – Абстрактний приклад трансформації за допомогою Control Flow Flattening

На рисунку 2.3 представлено практичне застосування методу на прикладі простої перевірки двох чисел та визначення меншого з них. Після обфускації було створено одновимірну анонімну функцію, що одразу викликається після запуску коду. У ній задекларовано спеціальний прапорець `state`, від значення якого залежить, у який `case` перейде інтерпретатор для виконання потрібної логіки. До того ж, обов'язково має бути передбачений механізм виходу з циклу для уникнення зациклення, що зробить код нефункціональним.

```
const x = 5,
    y = 7;

if (x < y) {
    console.log("X is lower");
} else {
    console.log("Y is lower");
}
```

```
(function(){
    const x = 5, y = 7;
    var state = 0;

    while (true) {
        switch (state) {
            case 0:
                if (x < y) {
                    state = 1;
                } else {
                    state = 2;
                }
                break;
            case 1:
                console["log"]("X is lower");
                state = null;
                break;
            case 2:
                console["log"]("Y is lower");
                state = null;
                break;
            default:
                return;
        }
    }
})();
```

Рисунок 2.3 – Застосування Control Flow Flattening

Метод Control Flow Flattening забезпечення параметризації, що значно ускладнює процес зворотного інжинірингу (reverse-engineering) та підвищує стійкість до статичного аналізу коду [18].

2.1.2 Реструктуризація даних

У більшості програм використовуються для реалізації специфічної логіки масиви. Це зручна структура даних, яка дозволяє зберігати подібні значення та об’єднувати їх для спільної обробки. Метод реструктуризації даних полягає у розбитті одновимірних масивів на вкладені підмасиви або, навпаки, у зворотному об’єднанні. У результаті зміниться лише форма даних, однак обчислення програми залишається незмінним за умови правильного застосування методу [19].

На рисунку 2.4 показано практичне застосування цього підходу для ітерації по кожному елементу масиву з виведенням значення у консоль. У вхідному коді була одна конструкція `for` та одновимірний масив `arr`. Після застосування методу в обфускованому коді з’явилися вкладені підмасиви та додатковий підцикл. При цьому логіка коду зовсім не змінилась, однак читабельність значно знизилась.

```
var arr = [1, 2, 3];
for (var i = 0; i < arr.length; i++) {
    console.log(arr[i]);
}
```

```
1  var arr = [[1, 2], [3]];
2  for (var i = 0; i < arr.length; i++) {
3      for (var j = 0; j < arr[i].length; j++) {
4          console.log(arr[i][j]);
5      }
6  }
```

Рисунок 2.4 – Застосування методу реструктуризації даних

Цей метод доволі простий в реалізації і є нестійким до деобфускації. Проте його можна удосконалити – ускладнити масив даних шляхом додавання скалярних даних та масивів одночасно. Це змінить логіку циклів, але результат не зміниться.

2.1.3 Клонування структур

Суть цього методу полягає в клонуванні існуючих структур даних, таких як функцій, масивів, тощо. В базових варіаціях цього методу застосовується просте копіювання з додаванням нових ідентифікаторів. Далі логіку коду видозмінюють, використовуючи випадковим чином або оригінальну структуру, або копію [18]. Приклад застосування цього методу наведено на рисунку 2.5.

```
function calcSquare(x) {
    console.log(Math.pow(x, 2));
}

calcSquare(5);
```

```
1  function a(x) {
2      console.log(Math.pow(x, 2));
3  }
4
5  function b(x) {
6      console.log(Math.sqrt(x ** 2 * x * x));
7  }
8
9  (Math.random() < 0.5 ? a : b)(5);
```

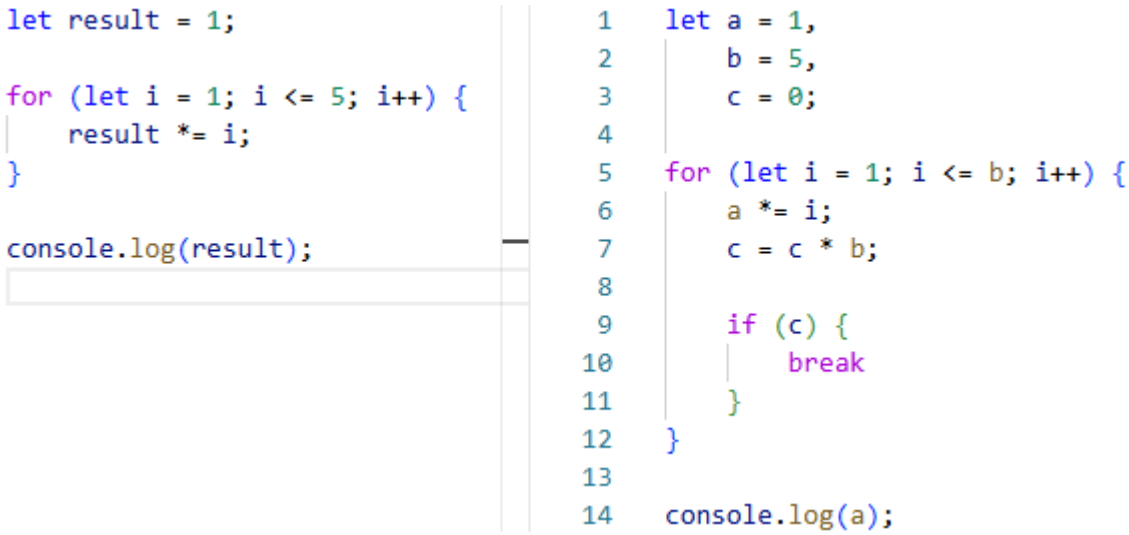
Рисунок 2.5 – Застосування методу клонування структур

У даному прикладі було застосовано клонування функції `calcSquare`. Результат виконання функції-клону ніяк не відрізняється, однак додавання складнішої логіки для досягнення такого ж результату та виконання випадкової функції ускладнює аналіз логіки коду. У результаті у зловмисника може скластися враження, що ці функції різні та виконують різні операції, хоча насправді це не має значення – яка з них буде обрана [19].

2.1.4 Вставка мертвого коду

Цей метод полягає у вставці фрагментів коду (Dead code insertion), які не мають жодного впливу на кінцевий результат виконання програми. Це можуть бути умовні конструкції з гілками, цикли з невикористовуваним лічильником, зайві функції, що ніколи не викликаються, або непотрібні змінні тощо. Збільшуючи обсяг коду без зміни його функціональної поведінки, вставка "мертвого" коду значно ускладнює зловмиснику розпізнавання справжньої логіки та послідовності виконання програми. Ця техніка ефективно підвищує

складність та час, необхідний для аналізу, тим самим покращуючи захищеність програмного забезпечення [20]. На рисунку 2.6 наведено приклад застосування цього методу для обфускації коду.



```
let result = 1;

for (let i = 1; i <= 5; i++) {
    result *= i;
}

console.log(result);
```

```
let a = 1,
    b = 5,
    c = 0;

for (let i = 1; i <= b; i++) {
    a *= i;
    c = c * b;

    if (c) {
        break
    }
}

console.log(a);
```

Рисунок 2.6 – Приклад методу обфускації мертвим кодом

У вихідному коді було додано дві додаткові змінні: одна — це лічильник, інша — так звана *"мертва"* змінна [19]. Згідно з логікою початкового коду, можна припустити, що додана конструкція здатна зупинити цикл, тобто певним чином вплинути на результат обчислення суми геометричної прогресії. Однак, якщо уважно проаналізувати значення змінної c, можна помітити, що воно дорівнює 0. При множенні на 0 результат завжди буде 0, отже, умова завершення циклу ніколи не виконається, і цикл завершуватиметься завжди, незалежно від значення змінної c. Таким чином, ця вставка може ввести в оману під час аналізу коду програми.

2.1.5 Розширення областей видимості

Цей метод розширення областей видимості (Scope obfuscation) може значно ускладнити розуміння коду шляхом перевизначення змінних всередині різних контекстів функцій. Суть підходу полягає у декларуванні додаткових областей видимості, що не матимуть ніякого сенсу, однак при аналізі можуть ввести в

оману. Особливо при великій кількості вкладених функцій повторне використання однакових імен змінних значно ускладнює сприйняття логіки програми. На рисунку 2.7 показано приклад обфускації коду зі створенням нових областей видимості та зміною значень змінних.

```
let a = 5;

function pow(x) {
    return x ** 2;
}

console.log(pow(a));
```

```
let a = 7;

(function(){
    let a = 10;
    {
        let a = 5;

        function b(a) {
            return a ** 2;
        }

        console.log(b(a));

    }
})();
```

Рисунок 2.7 – Приклад застосування scope obfuscation

У цьому прикладі в обфускованому коді змінній `a` декілька раз призначається нове значення. Завдяки особливостям інтерпретатора JavaScript, кожне значення застосовується лише в межах відповідної області видимості. Крім того, використання однакової назви змінної як у внутрішній функції, так і в зовнішньому контексті може спричинити плутанину під час аналізу. Цей метод працює у випадку використання оператора `let`, адже він забезпечує блочну область видимості. Натомість при використанні `var` значення буде встановлюватися в глобальну або функціональну область, що зменшує ефективність техніки.

Ця техніка на перший погляд може здаватися спрощеною версією Dead Code Insertion, однак має ключову відмінність: вона вводить кілька областей видимості з однаковими ідентифікаторами. У нашому тривіальному прикладі це легко зрозуміти, оскільки жодна з функцій у внутрішній області видимості не викликається. Проте в реальних умовах це може створити значні труднощі при аналізі поведінки програми [21].

2.1.6 Метод трансформації циклів

Трансформація циклів (Loop transformation) змінює структуру циклічних конструкцій у коді з метою ускладнення розуміння їхньої мети та функціональності. Це може включати такі методи, як розгортання циклів (loop unrolling), коли тіло циклу перетворюється на повторювані послідовності інструкцій, або зміну умов виконання циклу та керуючих змінних нетривіальним способом. Такі трансформації роблять потік керування менш передбачуваним і порушують типові шаблони, на які зазвичай орієнтуються реверс-інженери для розуміння логіки коду. Застосування цього методу дозволяє суттєво підвищити складність аналізу, посилюючи стійкість програми до атак [20].

```
const x = 5
let sum = 0;
for (let i = 1; i <= x; i++) {
    sum += i;
}

console.log(sum)
```

```
1   const x = 5
2   let sum = 0,
3       i = 1;
4   if (i <= x && (i + sum * x) >= i ) { sum += i; i++; }
5   if (i <= x && (i + sum * x) >= i ) { sum += i; i++; }
6   if (i <= x && (i + sum * x) >= i ) { sum += i; i++; }
7   if (i <= x && (i + sum * x) >= i ) { sum += i; i++; }
8   if (i <= x && (i + sum * x) >= i ) { sum += i; i++; }
9
10
11  console.log(sum)
```

Рисунок 2.8 – Приклад обфускації за допомогою Loop transformation

У вихідному коді цикл було розгорнуто до п’яти умовних перевірок, а також додано зайву перевірку, яка завжди буде істинною, оскільки безпосередньо залежить від значення змінної `i`. Незалежно від значень змінних `sum` та `x`, змінна `i` завжди буде більшою або рівною сама собі [19]. Таким чином, модифікація умови жодним чином не впливає на кінцевий результат обчислень. Як видно з прикладу, код стає значно менш зрозумілим і потребує детального аналізу для відокремлення реальної логіки від зайвих, заплутуваних конструкцій.

2.1.7 Використання міток

У написанні JavaScript-застосунків мітки використовуються вкрай рідко, проте їх активно застосовують під час обфускації програмного коду. Велика кількість міток може ввести в оману зловмисника та значно ускладнити аналіз послідовності виконання, зробивши код менш читабельним. Цей метод є досить ефективним проти реверс-інжинірингу, оскільки за допомогою міток звична лінійна логіка виконання змінюється на неочікувані переходи (стрибки), що ускладнює налагодження коду в процесі дебагу [19].

На рисунку 2.9 наведено простий приклад застосування міток для умовних операторів, але в більш масштабних сценаріях їх використання значно ускладнює розуміння коду.

```
const x = prompt('Enter x:')

if (x > 0) {
    console.log("Positive");
} else {
    console.log("Non-positive");
}
```

```
1   lbl: {
2       const x = prompt('Enter x:')
3
4       lb2: {
5           if (x > 0) {
6               console.log("Positive");
7               break lbl;
8           }
9
10          console.log("Non-positive");
11      }
12  }
```

Рисунок 2.9 – Приклад застосування міток

У цьому прикладі звичайний код трансформовано у позначений блок `lbl` з вкладеним блоком `lb2`. Використання конструкції break `lbl` дозволяє вийти з усієї програми при виконанні певної умови. Обфускований код дає такий самий результат, однак структура коду істотно змінюється: умовний оператор розбивається на два позначені блоки, що є нетиповим і неочікуваним для аналітика.

2.2 Засоби обфускації, що використовують перетворення змінних

В основі цієї групи методів обфускації лежить зміна форми представлення змінних, потоку даних та трансформація типів цих даних, що ускладнює зворотний аналіз. Вихідний код перетворюється так, щоб приховати логічні зв'язки між змінними – кожен оригінальний об'єкт даних розбивається або маскується низкою додаткових дій і допоміжних змінних.

2.2.1 Розділення і злиття змінних

Суть цього методу полягає у виявленні змінних та їх подальшому розбитті на підзмінні. Таким чином, для отримання початкового значення необхідно виконати додаткові операції, наприклад: обчислення математичного виразу для отримання числового значення або конкатенацію рядків для збирання повного текстового значення [19]. Цей підхід також може застосовуватись у зворотному порядку — розділені змінні об'єднуються, після чого за допомогою побітових операцій, таких як зсув, відбувається розгортання об'єднаних значень. Хоча реалізація такого обфускаційного скрипта є досить складним завданням, результат виходить надзвичайно заплутаним. Зловмиснику буде значно важче одразу виявити, що об'єднана змінна насправді містить два або більше окремих значень [22].

На рисунку 2.10 продемонстровано злиття змінних та їх подальше розділення із використанням складніших арифметичних та логічних операцій.

```
let x = 42;
    y = 19;

console.log("Result x - y: " + (x - y))
```

```
1   let x = 42;
2   let y = 19;
3   let key = 77;
4
5   let merged = ((x ^ key) << 8) | (y ^ key);
6
7   let decodedX = ((merged >> 8) ^ key);
8   let decodedY = ((merged & 0xFF) ^ key);
9
10  console.log("Result x - y: " + (decodedX - decodedY))
```

Рисунок 2.10 – Приклад злиття/розділення змінних для обфускації JS-коду

Спочатку значення змінних `x` та `y` було об'єднано в одну змінну `merged`, після чого за допомогою логічних і арифметичних операцій їх було відновлено до десяткового вигляду. У реальному застосуванні початкові значення змінних можна видалити, залишивши в коді лише результат злиття. Це дозволяє приховати оригінальні дані та ускладнити їх відновлення. Проте в цьому прикладі, з метою демонстрації роботи методу, початкові значення залишено.

Загалом, злиття та розділення змінних ускладнює статичний аналіз коду: для відновлення початкових значень необхідно простежити проміжні змінні та операції, що суттєво збільшує час аналізу та ускладнює реверс-інжиніринг.

2.2.2 Перейменування змінних

Цей метод полягає у заміні зрозумілих імен змінних, функцій, класів тощо на беззмістовні, випадкові або односимвольні ідентифікатори. Функціональність програми при цьому не змінюється, проте втрачається семантична інформація про призначення змінних [23]. Перейменування є одним із найпоширеніших прийомів обфускації. Цей метод зазвичай застосовується у поєднанні з іншими техніками для досягнення бажаного рівня захисту.

```
function calculateArea(width, height) {
    let area = width * height;
    return area;
}

let rectWidth = 10;
let rectHeight = 5;
let result = calculateArea(rectWidth, rectHeight);
console.log("Area:", result);
```

```
1  let e = 10;
2  let f = 5;
3  let g = (function (b, c) {
4      let d = b * c;
5      return d;
6  })(e, f)
7
8  console.log("Area:", g);
9
```

Рисунок 2.11 – Обфускація імен змінних на практиці

На рисунку 2.11 продемонстровано використання цього методу. Усі зрозумілі імена було замінено на прості літери англійського алфавіту, через що визначити призначення змінної за її іменем стало неможливо. Такий підхід значно ускладнює читання та аналіз коду людиною, адже доводиться вручну встановлювати, які змінні до чого належать. Для протидії цьому методу деякі

деобфускатори намагаються автоматично відновити імена змінних, наприклад, за допомогою методів машинного навчання. Однак гарантувати точне відновлення початкових назв неможливо, оскільки інформація про оригінальні імена безповоротно втрачається [24].

2.2.3 Кодування рядків

Цей метод полягає в приховуванні текстових або числових констант шляхом їх кодування або шифрування, що значно ускладнює їх розпізнавання під час статичного аналізу коду та реверс-інженерії. Замість звичних літералів використовуються, наприклад, шістнадцяткові значення, Base64 або інші формати, які розкодовуються безпосередньо під час виконання програми. У складніших випадках застосовується шифрування рядків, коли їх справжнє значення зберігається в зашифрованому вигляді та відновлюється лише в момент потреби. Хоча динамічне декодування або розшифрування може дещо впливати на продуктивність, зазвичай цей вплив є незначним, тоді як рівень захищеності коду суттєво підвищується [20].

```
console.log("Test message for demonstration");

// Шістнадцяткове кодування
console.log("\x54\x65\x73\x74\x20\x6D\x65\x73\x73\x61\x67\x65\x20\x66"
    + "\x6F\x72\x20\x64\x65\x6D\x6F\x6E\x73\x74\x72\x61\x74\x69\x6F\x6E");

// Unicode символи
console.log("\u0054\u0065\u0073\u0074\u0020\u006D\u0065\u0073\u0073\u0061\u0067"
    + "\u0065\u0020\u0066\u006F\u0072\u0020\u0064\u0065\u006D"
    + "\u006F\u006E\u0073\u0074\u0072\u0061\u0074\u0069\u006F\u006E");

// Base64 представлення
console.log(atob("VGVzdCBtZXNzYWdlIGZvciBkZW1vbnN0cmF0aW9u"));
```

Рисунок 2.12 – Приклад кодування повідомлення для приховування його змісту

На рисунку 2.12 показано приклад, у якому текст *"Test message for demonstration"* було закодовано трьома різними способами. Для зовнішнього спостерігача такий код виглядає нечитабельним, проте для інтерпретатора це не

створює жодних труднощів — результат виводу залишиться ідентичним. Деякі сучасні обфускатори додатково виносять усі літерали (рядки та числа) в окремі масиви та замінюють їх на виклики функцій розшифрування, що ще більше ускладнює аналіз коду [25].

2.3 Підходи обфускації з використанням пунктуаційних перетворень

Один із напрямів обфускації JavaScript-коду полягає у використанні пунктуаційних перетворень, суть яких полягає в зміні синтаксичного оформлення коду без зміни його функціональності. Такі методи передбачають видалення або навмисне дублювання розділових символів, форматування інструкцій у нестандартний спосіб, застосування тернарних виразів, вставку невидимих символів тощо. Метою таких трансформацій є ускладнення аналізу коду.

2.3.1 Інверсія кодових елементів

Суть цього методу обфускації полягає у трансформації початкового зрозумілого коду в інверсований або дезорганізований. Основна ідея — змінити порядок викликів функцій, розділити складні обчислення на кілька етапів, а потім об’єднати їх для отримання того самого результату. Порядок оголошення та використання змінних і функцій у JavaScript дозволяє значні маніпуляції, які застосовуються для заплутування логіки виконання. Така «перестановка» блоків коду не змінює функціональність програми, але значно ускладнює розуміння її роботи та статичний аналіз.

JavaScript підтримує подібні трансформації завдяки механізму "*hoisting*" — підняття оголошень змінних, здійснених через `var`, на початок області видимості [26]. Це дозволяє використовувати змінні до їх фактичного оголошення в коді. Водночас при використанні `let` і `const` потрібно враховувати наявність тимчасової мертвої зони, яка ускладнює такі маніпуляції. Загалом перестановка змінних, функцій та логічних блоків є ефективним засобом обфускації коду, що

підвищує його стійкість до реверс-інженерії — аналогічно до вирівнювання контрольного потоку [19].

На рисунку 2.13 наведено застосування інверсії коду до обфускації простої функції автентифікації. Хоча в реальних умовах імена користувачів та паролі не зберігаються у відкритому вигляді в коді, для наочності використано саме такий приклад.

```
function authenticate(user, password) {
    if (user === "admin" && password === "1234") {
        return "Access granted";
    } else {
        return "Access denied";
    }
}

console.log(authenticate(
    prompt("Username"),
    prompt("Password")
));
```

```
const secrets = ["nimda", "4321"];
const msg = ["Access granted", "Access denied"];

function check(p, u) {
    let x = u.split("").reverse().join("");
    let y = p.split("").reverse().join("");

    let result = (x === secrets[1].split("").reverse().join("") &&
                  y === secrets[0].split("").reverse().join("")) ? msg[0] : msg[1];
    return result;
}

let log = check(prompt("Username").split("").reverse().join(""),
                prompt("Password").split("").reverse().join(""));

console["log"](log);
```

Рисунок 2.13 – Приклад застосування інверсії коду

У вихідному коді функція authenticate приймає два рядки та безпосередньо порівнює їх із константами `"admin"` і `"1234"`. Після обфускації ці значення зберігаються у масиві `secrets`, але у зворотному порядку — `"nimda"` та `"4321"` замість оригінальних значень. Далі введені користувачем значення обертаються за допомогою `split("").reverse().join("")`, щоб відповідати зворотному формату, після чого виконується порівняння. Це дозволяє приховати прямий зміст даних у коді та заплутати логіку перевірки. Додатково виведення результату реалізовано через `console["log"]` замість звичного `console.log`, що додає ще один рівень непрозорості.

2.3.2 Видалення токенів

Для обфускації JavaScript зазвичай використовують мініфікацію — процес видалення всіх неважливих для виконання символів, зокрема пробілів, розривів рядків, табуляцій і коментарів. Хоча цей метод не впливає на функціональність скрипта, він суттєво ускладнює його читання людиною. Відсутність відступів та

коментарів робить складнішим швидке сприйняття логіки програми, особливо у великих проектах. Попри те, що базову структуру коду можна частково відновити за допомогою спеціальних форматерів, мініфікація залишається невід'ємною частиною майже всіх сучасних обфускаторів. Вона виконує подвійну функцію: знижує читабельність коду для сторонніх осіб і водночас зменшує його обсяг, що позитивно впливає на швидкість завантаження веб-сторінок [27]. Завдяки простоті впровадження, універсальності та ефективності, цей метод вважається стандартним кроком у процесі захисту скриптів.

```
function process(data) {
    if (data.length > 0) {
        for (let i = 0; i < data.length; i++) {
            console.log("Item:", data[i]);
        }
    } else {
        console.log("No data");
    }
}

process(["a", "b", "c"]);

// Мініфікований результат
function process(d){if(d.length>0){for(let i=0;i<d.length;i++)
console.log("Item:",d[i])}else console.log("No data")}process(["a","b","c"])
```

Рисунок 2.14 – Приклад мініфікації коду

На рисунку 2.14 показано приклад застосування мініфікації: добре відформатований код перетворено на щільну однорядкову "кашу", яку надзвичайно важко читати без попереднього форматування.

2.3.3 Використання прихованих символів

Одним із найбільш витончених методів обфускації JavaScript-коду є використання невидимих символів Unicode, таких як `\u3164` (Hangul Filler), які не відображаються в редакторах коду або браузері, але технічно присутні в коді. Цей підхід дозволяє приховати логіку виконання за рахунок того, що частини шкідливого або конфіденційного коду кодуються через довжини властивостей

або символи, а потім динамічно відновлюються та виконуються за допомогою конструкцій `Proxy` і `eval()`. Таким чином, зовні код може здаватися порожнім або безпечним, але насправді містить логіку, що активується лише під час виконання. Ця техніка є особливо небезпечною через здатність обходити засоби статичного аналізу коду [28].

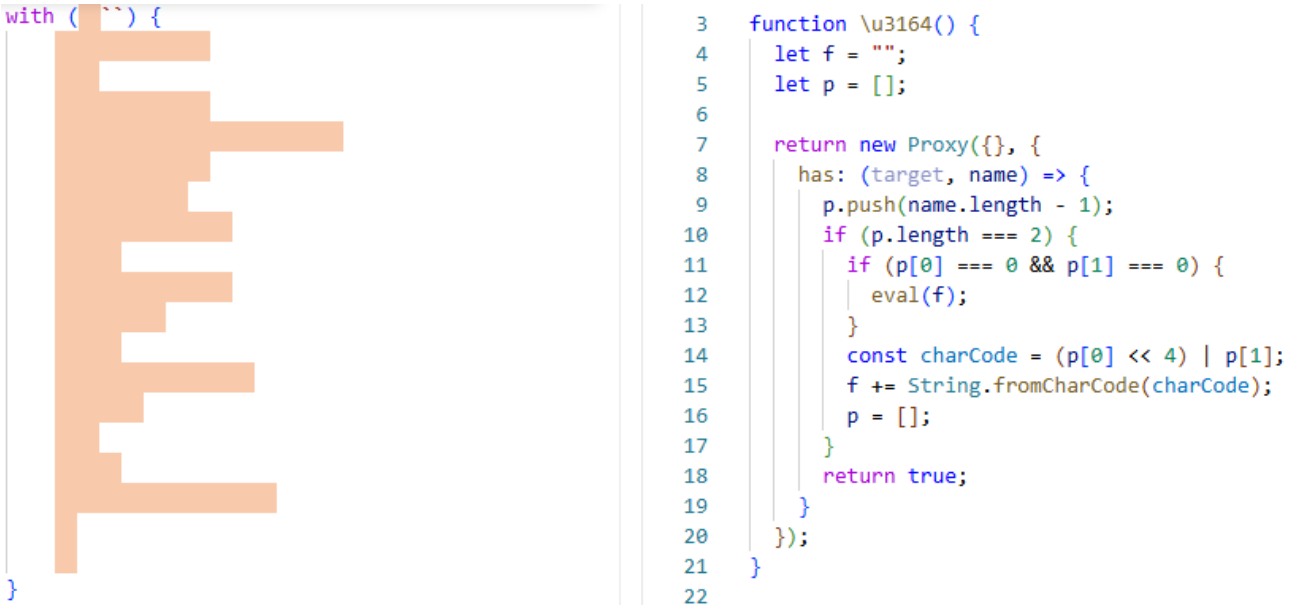


Рисунок 2.15 – Застосування прихованих символів як обфускація виклику `alert(1)`

На рисунку 2.15 зліва наведено приклад використання конструкції `with`, в якій у якості об'єкта передається результат виклику функції з назвою `\u3164`. Хоча ім'я функції на перший погляд здається порожнім, JavaScript-інтерпретатор правильно розпізнає символ `\u3164` як дійсний Unicode-ідентифікатор (символ "порожнього пробілу") і шукає функцію з таким ім’ям. Визначена функція `\u3164` повертає об'єкт `Proxy`, який перехоплює усі звернення до властивостей усередині блоку `with`. Кожен візуально порожній рядок у тілі `with` фактично містить різну кількість невидимих символів і при спробі звернутися до них викликається перевірка `name in proxy`, що активує перехоплювач `has` у `Proxy`. Ця дія виконується для кожного рядка окремо. При кожному такому виклику з імені властивості (яке насправді є рядком з певною кількістю пробілів) визначається його довжина, і до масиву `p` додається значення `name.length - 1`,

що дає 0 або 1. Таким чином, кожні два виклики накопичують біти одного символу: перше значення зсувається на 4 біти вліво (тобто множиться на 16), друге — додається до нього побітовим OR, утворюючи значення від 0 до 255. Цей код символу додається до результуючого рядка через `String.fromCharCode`. Коли обидва елементи `p` дорівнюють 0 (тобто два однобайтові порожні рядки, що означають завершення), виконується декодування — зібраний рядок передається у `eval`, що запускає обчислення закодованої логіки. Таким чином, за рахунок викликів has та контролю структури з допомогою Unicode-пробілів, відбувається приховане поступове збирання та виконання обфускованого коду .

Цей метод є прикладом складної та підступної обфускації, що активно використовується для приховування справжньої логіки виконання коду. Він ускладнює як ручний аналіз, так і автоматизовану перевірку, оскільки код стає повністю непрозорим без попереднього декодування. Подібні техніки нерідко застосовуються зловмисниками для маскування шкідливих скриптів.

РОЗДІЛ 3 ПОРІВНЯЛЬНИЙ АНАЛІЗ ЕФЕКТИВНОСТІ МЕТОДІВ ОБФУСКАЦІЇ

3.1 Програмне забезпечення для обфускації JavaScript-коду

Проблема обфускації саме JavaScript є доволі актуальною, адже додатки, написані цією мовою програмування, здебільшого виконуються на стороні клієнта при завантаженні веб-сторінки. Через те, що JavaScript-код доступний у відкритому вигляді, зловмисники можуть легко аналізувати його, копіювати бізнес-логіку, алгоритми або чутливі частини функціоналу [17]. З метою ускладнення такого аналізу існує чимало як комерційних, так і open-source рішень для обфускації, які дозволяють приховати логіку коду від конкурентів або зловмисників, що прагнуть завдати шкоди власникам веб-ресурсів. У цьому підрозділі буде розглянуто п’ять популярних обфускаторів JavaScript-коду та проаналізовано їхні можливості щодо заплутування коду без порушення його функціональності.

3.1.1 JScrambler

JScrambler — це комерційний сервіс (SaaS) для обфускації та захисту JavaScript- і HTML5-коду. Він надає широкий набір інструментів, що дозволяють ускладнити аналіз і модифікацію клієнтської частини програм, забезпечуючи захист бізнес-логіки, чутливих даних та інтелектуальної власності. Платформа пропонує декілька тарифних планів (Starter, Enterprise тощо) з можливістю безкоштовного пробного періоду. Клієнтські утиліти, такі як CLI та Web API — доступні за відкритими ліцензіями, у той час як основний функціонал доступний за підпискою. JScrambler реалізує розширений набір трансформацій JavaScript-коду, серед яких:

– control flow flattening;

– розділення/злиття змінних;

– мініфікація коду (видалення токенів);

– вставка мертвого коду;

– трансформація циклів;

– перейменування змінних;

– інверсію кодових елементів;

– реструктуризація даних, тощо.

Крім цього, сервіс підтримує поліморфну обфускацію, коли кожна нова збірка коду генерує унікальний результат, що суттєво ускладнює повторне застосування зловмисних технік зворотного інжинірингу [17]. Серед новітніх функцій — VM-Based Obfuscation, яка компілює JavaScript у двійкову проміжну мову з випадково згенерованими інструкціями. Отриманий захищений код містить як інтерпретатор JavaScript, так і навантаження у вигляді двійкового коду, який щоразу формується за новою схемою.

Окрім трансформації та маскування коду, JScrambler забезпечує функції самозахисту. Оброблені додатки здатні виявляти спроби втручання — наприклад, запуск дебагера або модифікацію логіки — і відповідним чином реагувати. Це включає блокування несанкціонованого доступу до функцій, приховування літералів і виявлення дебагерів для ініціації активного захисту (наприклад, завершення виконання або запуск контрзаходів).

Важливою перевагою JScrambler є інтеграція в CI/CD-процеси — захист коду може автоматично застосовуватись під час кожного деплою. Це можливо завдяки повнофункціональній CLI-версії, що забезпечує ті самі можливості, що й веб-інтерфейс платформи. У веб-версії користувач може вручну завантажити необхідні файли та налаштувати конфігурацію обфускації відповідно до бажаного балансу між продуктивністю й рівнем захисту. На рисунку 3.1 показано вигляд веб-інтерфейсу JScrambler.

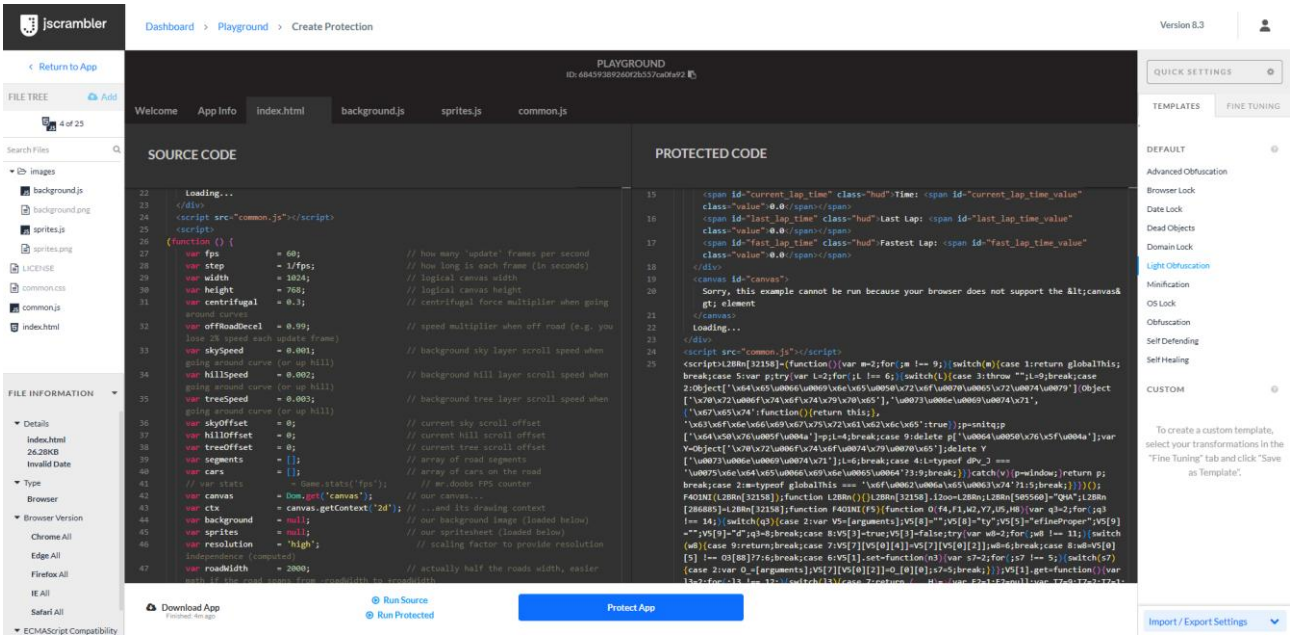


Рисунок 3.1 – Вигляд веб-додатку JScrambler

JScrambler оптимізовано для сучасних веб-додатків: він працює з JavaScript-кодом у форматах ES5, ES6/ES2015 і навіть ES8. Сервіс підтримує інтеграцію з найпопулярнішими JS фреймворками та бібліотеками такими, як React, Angular, тощо. Для зручності користувачів JScrambler надає гайдлайни та плагіни для інтеграції захисту в збирачі проєктів, зокрема Webpack і Metro, а також підтримує середовища мобільної збірки. Це робить його придатним як для фронтенд, так і для кросплатформених рішень.

Завдяки потужним засобам обфускації та активного захисту, JScrambler широко використовується, включно з компаніями зі списку Fortune 500. Комерційна підтримка сервісу гарантує постійне оновлення інструментів відповідно до актуальних загроз. Додаткові функції моніторингу та профілювання дозволяють налаштовувати захист, враховуючи продуктивність додатка і поведінкові дані з реального середовища.

3.2.2 JShaman (JS–Obfuscator)

JShaman — це професійний онлайн-сервіс (SaaS) для обфускації та шифрування JavaScript-коду, розроблений китайською компанією *Shanxi JShaman Tech*. Платформа є однією з провідних у Китаї та використовується

тисячами компаній і розробників. JShaman дозволяє ефективно захистити бізнес-логіку та інтелектуальну власність: обфускований код стає практично нерозбірним і не піддається зворотному відновленню. Сервіс працює без обов'язкової реєстрації — достатньо вказати секретний ключ або VIP-код. Залежно від обраного тарифу, користувач має змогу скористатися безкоштовною базовою версією (з обмеженим функціоналом) або розширеним захистом на основі VIP-підписки [29].

JShaman реалізує десятки технологій обфускації JavaScript-коду для максимального ускладнення його аналізу. Серед ключових методів, що підтримуються платформою є:

– control flow flattening;

– вставка мертвого коду;

– перейменування змінних;

– кодування рядків;

– розділення/злиття змінних;

– мініфікація коду (видалення токенів);

– шифрування JSON та регулярних виразів, тощо.

Сервіс також підтримує поліморфну обфускацію: кожна нова обфускація одного і того ж вихідного коду призводить до створення унікального захищеного файлу. Це унеможливлює відтворення шаблонів захисту та протидіє автоматизованому злому [29]. На відміну від таких рішень, як Jscrambler, JShaman не реалізує активних функцій самозахисту, зокрема виявлення дебагерів чи реагування на модифікацію коду під час виконання. Основу захисту тут становить висока складність та нестандартність синтаксичних конструкцій у результаті статичних перетворень.

JShaman доступний через веб-інтерфейс (див. рисунок 3.2), однак також надає розширені можливості інтеграції в процес розробки. Зокрема, сервіс пропонує Web API, що дозволяє надсилати код (або архіви з файлами) на обфускацію безпосередньо з програмного середовища або CI/CD-пайплайну. Окрім хмарної версії, існує локальна інсталяція платформи, що встановлюється на власний сервер користувача. Локальний варіант має ідентичний функціонал,

але працює в повністю автономному режимі без підключення до Інтернету, що забезпечує максимальну швидкість обробки та відсутність обмежень на обсяг чи частоту запитів [29].

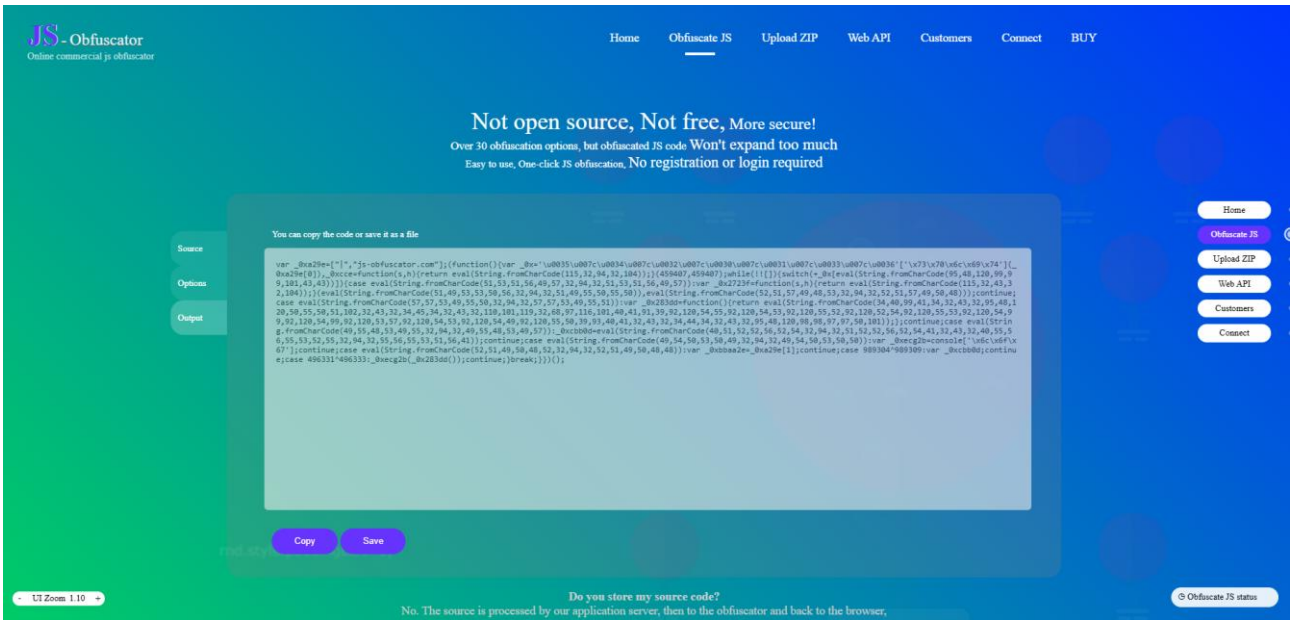


Рисунок 3.2 – Вигляд веб-сайту JS-obfuscator

Комерційна модель JShaman передбачає регулярні оновлення та підтримку: команда розробників оперативно реагує на появу нових загроз, впроваджує покращення захисту та оптимізації коду. Додатково сервіс надає можливість моніторингу результатів обфускації — перегляд статистики операцій, часу виконання, кількості трансформацій тощо. Завдяки широкому набору обфускаційних функцій, простоті використання та високій гнучкості, JShaman є ефективним рішенням як для невеликих веб-проектів, так і для складних комерційних систем зі значною кодовою базою.

3.1.3 JavaScript Obfuscator (Obfuscator.io)

Цей інструмент є повністю безкоштовним і з відкритим вихідним кодом. Його важливою перевагою перед іншими обфускаторами є активна підтримка спільноти, а також можливість удосконалення власноруч, оскільки він є open-source-рішенням із публічним репозиторієм на GitHub. JavaScript Obfuscator

підтримує сучасні стандарти ECMAScript (включно до ES2022) і реалізований мовою TypeScript [30]. Його основні функціональні можливості включають:

– перейменування змінних;

– кодування рядків;

– вставку мертвого коду;

– control flow flattening;

– реструктуризація даних;

– перейменування змінних, тощо;

Крім того, обфускатор реалізує механізм винесення рядкових значень до окремого масиву з подальшим їх випадковим кодуванням одним із доступних методів, зокрема `base64` або `RC4`. Опція *Self-Defending* дозволяє зробити захищений код чутливим до змін: у випадку будь-якої спроби форматування або спрощення структури скрипта він припиняє виконання. Це значно підвищує стійкість обфускованого коду до реверс-інжинірингу та автоматизованого аналізу [14].

Інструмент має зручний веб-інтерфейс, який показано на рисунку 3.3. Через нього користувач може обрати всі необхідні параметри та сформувати конфігурацію, яка забезпечує бажаний баланс між рівнем захисту та розміром коду [30]. Підтримується можливість вставлення окремих фрагментів коду або повних JavaScript-файлів, вибір одного з трьох пресетів складності, активація генерації source map, а також перегляд прикладів коду до та після обфускації, що дозволяє підібрати оптимальні налаштування.

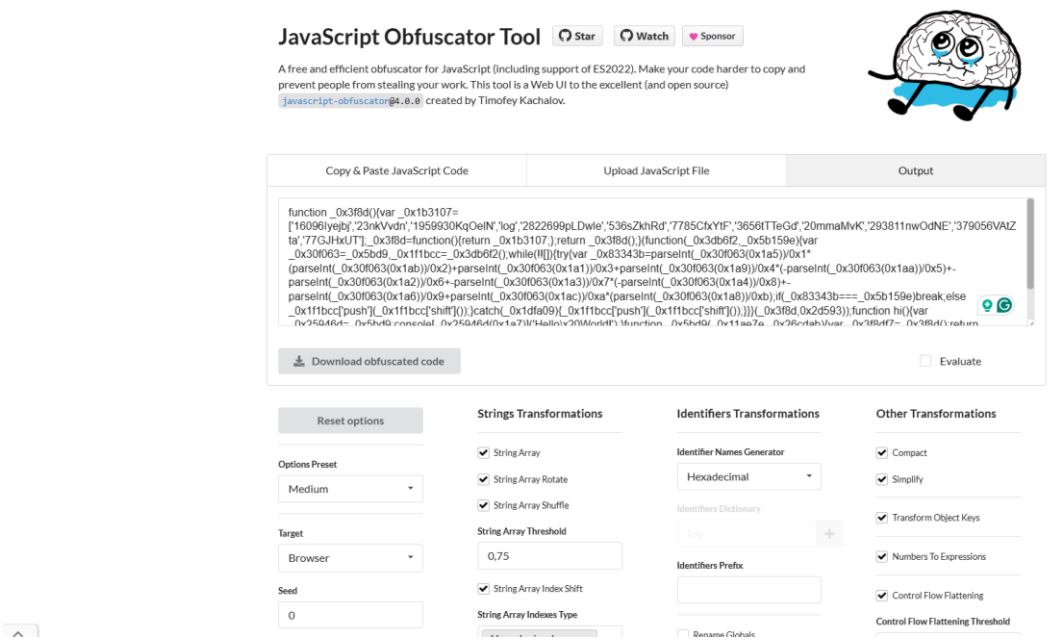


Рисунок 3.3 – Налаштування опцій obfuscator.io на веб-сайті

3.2 Критерії ефективності методів обфускації

Оцінювання ефективності методів обфускації коду є ключовим етапом при розробці засобів захисту програмного забезпечення від реверс-інжинірингу. Сучасні підходи до захисту програм у контексті моделі загроз «man-at-the-end» (MATE) базуються на застосуванні обфускації як одного з методів ускладнення статичного та динамічного аналізу. Проте без чітко визначених критеріїв складно об'єктивно оцінити, наскільки застосована обфускація є ефективною. У науковій літературі традиційно виокремлюють чотири базові характеристики обфускаційних перетворень: «потужність» (potency), «стійкість» (resilience), «маскувальність» (stealth) та «вартість» (cost) їх реалізації. Ефективна обфускація, як правило, істотно підвищує складність аналізу коду (potency), мінімально впливає на продуктивність системи (cost) та ускладнює автоматизоване відновлення логіки програми (resilience) [31]. Визначення та застосування формалізованих критеріїв дозволяє здійснювати порівняння між різними методами обфускації, знаходити оптимальний баланс між безпекою й продуктивністю, а також оцінювати стійкість до новітніх засобів деобфускації.

Рівень ускладнення коду — показує, наскільки збільшується складність програмного коду після застосування обфускації. Цей показник ґрунтується на

співвідношенні певної обраної метрики складності між обфускованим і вихідним варіантами програми. Як зазначають Collberg та співавтори, такий підхід дозволяє оцінити «потужність» (potency) трансформації [31]. У якості метрик можуть використовуватись кількість операторів, складність розгалуження, глибина вкладеності, тощо. Для формалізації критерію застосовується така залежність, де C_{before} – значення метрики складності до обфускації, а C_{after} - після:

$$\mathrm{C} = \frac{\mathrm{C}_{after}}{C_{before}} \tag{3.1}$$

Якщо отримане значення $C > 1$, це свідчить про зростання складності програми, а отже — про вищу ефективність методу. Чим більший цей коефіцієнт, тим більш вираженим є ефект заплутування коду.

Стійкість до реверс-інжинірингу відображає здатність обфускованого коду протистояти спробам автоматичного або ручного відновлення його первісної структури. Цей критерій зазвичай асоціюється з поняттям *resilience* — наскільки ефективно алгоритм обфускації витримує атаки з боку деобфускаторів [31]. Інакше кажучи, йдеться про міру зусиль, необхідних для того, щоб повернути читабельну версію коду або частково реконструювати його логіку. Оцінювання цього критерію здійснюється емпірично шляхом запуску спеціалізованих інструментів аналізу — наприклад, деобфускаторів, дизасемблерів або декомпіляторів. Для формалізації доцільно використати співвідношення кількості успішних спроб відновлення до загальної кількості аналізів, де K — кількість успішних спроб аналізу або часткового відновлення логіки коду, а M — загальна кількість спроб:

$$I = 1 - \frac{K}{M} \tag{3.2}$$

Значення критерію *I*, що наближається до 1, свідчить про високу стійкість методу обфускації — тобто відновити початкову структуру коду практично неможливо навіть із застосуванням спеціалізованих засобів [32].

Вплив на швидкодію характеризує зміну продуктивності програмного коду внаслідок застосування обфускації. Йдеться, зокрема, про так звану «вартість» (cost) обфускації — тобто про те, наскільки виконання обфускованого коду є повільнішим порівняно з оригінальним. Такий вплив зазвичай проявляється у вигляді збільшення часу виконання, споживання пам'яті або розміру коду. Оцінювання цього критерію проводиться експериментально — шляхом порівняння часу виконання одного й того самого сценарію до та після обфускації. У своєму звіті Antoine Vastel зазначає, що середнє збільшення часу виконання після обфускації становить близько 15–20 % [33].

Для кількісної оцінки даного параметра застосовується формула, яка виражає відносне зростання часу виконання. Нехай T_{before} — середній час виконання оригінального коду, а T_{after} — відповідне значення для обфускованого:

$$P = \frac{T_{after} - T_{before}}{T_{before}} \tag{3.3}$$

Отриманий коефіцієнт *P* відображає відносне зростання часу виконання: чим більше його значення, тим більшим є вплив обфускації на швидкодію. У випадках, коли продуктивність є критичною, значення *P* слід мінімізувати для збереження прийнятного рівня ефективності.

Збереження функціональності є фундаментальним критерієм ефективності будь-якого методу обфускації. Він визначає, наскільки точно обфускований код зберігає функціональність програми. Обфускація не повинна змінювати поведінку системи — результат виконання має залишатися ідентичним оригіналу при однакових вхідних даних [32]. В іншому випадку обфускований код втрачає прикладну цінність і не може бути використаний на практиці. Для формалізації цього критерію доцільно використати співвідношення між кількістю успішно виконаних функцій після обфускації та загальною кількістю

функціональних одиниць у вихідному коді. Нехай N — кількість тестів (або поведінкових сценаріїв), які успішно завершуються після обфускації, а M — загальна кількість функцій, що були протестовані у вихідному варіанті:

$$F = \frac{N}{M} \tag{3.4}$$

Значення F, яке наближається до 1, вказує на повне збереження функціональності, тобто обфускація не вплинула на результат виконання програми. Зменшення цього коефіцієнта свідчить про наявність помилок, некоректної роботи або втрати частини логіки, що є критичним недоліком для реального застосування методу.

3.3 Результати порівняльного аналізу методів обфускації

Для проведення аналізу ефективності методів обфускації було обрано один із проєктів власної розробки — платформу для спілкування користувачів із можливістю публікації постів, їх коментування та взаємодії з контентом через реакції. Платформа реалізована на PHP із серверним рендерингом, однак для забезпечення інтерактивності використовується JavaScript — зокрема для надсилання асинхронних запитів без перезавантаження сторінки, додавання візуальних ефектів і динамічного оновлення елементів інтерфейсу. У системі також реалізовано адміністративну панель для керування користувачами, публікаціями та правами адміністраторів із гнучкою системою доступу. Це дає змогу делегувати окремі повноваження, наприклад — дозволити редагування лише постів без доступу до інших функцій.

Основною частиною JavaScript-логіки цього проекту, що підлягає обфускації, є live-чат, реалізований на React із використанням класових компонентів. Саме ця частина містить унікальну бізнес-логіку: обробку повідомлень, їх відображення, синхронізацію з сервером у режимі реального часу, що робить її вразливою до реверс-інжинірингу та потребує захисту від

несанкціонованого аналізу або копіювання. Саме ця логіка й стала об’єктом експериментального дослідження.

Для порівняння ефективності методів обфускації та вибору оптимального інструменту, відповідно до описаних раніше критеріїв, було використано три раніше розглянуті обфускатори. Усі вони підтримують інтеграцію через npm-пакети або Web API, проте в межах експерименту застосовувалися їхні веб-інтерфейси для обфускації коду live-чату.

На рисунку 3.4 показано результат використання шаблону Advanced Obfuscation у сервісі JScrambler. Оскільки пробна версія не дозволяє повної кастомізації параметрів, було обрано найсильніший доступний пресет із ввімкненою мініфікацією коду. Отриманий результат практично неможливо аналізувати вручну, оскільки застосовано значну кількість технік та трансформацій. Для розбору такого коду за допомогою базових JavaScript-деобфускаторів необхідне глибоке знання реверс-інжинірингу, а також поетапне відновлення логіки.

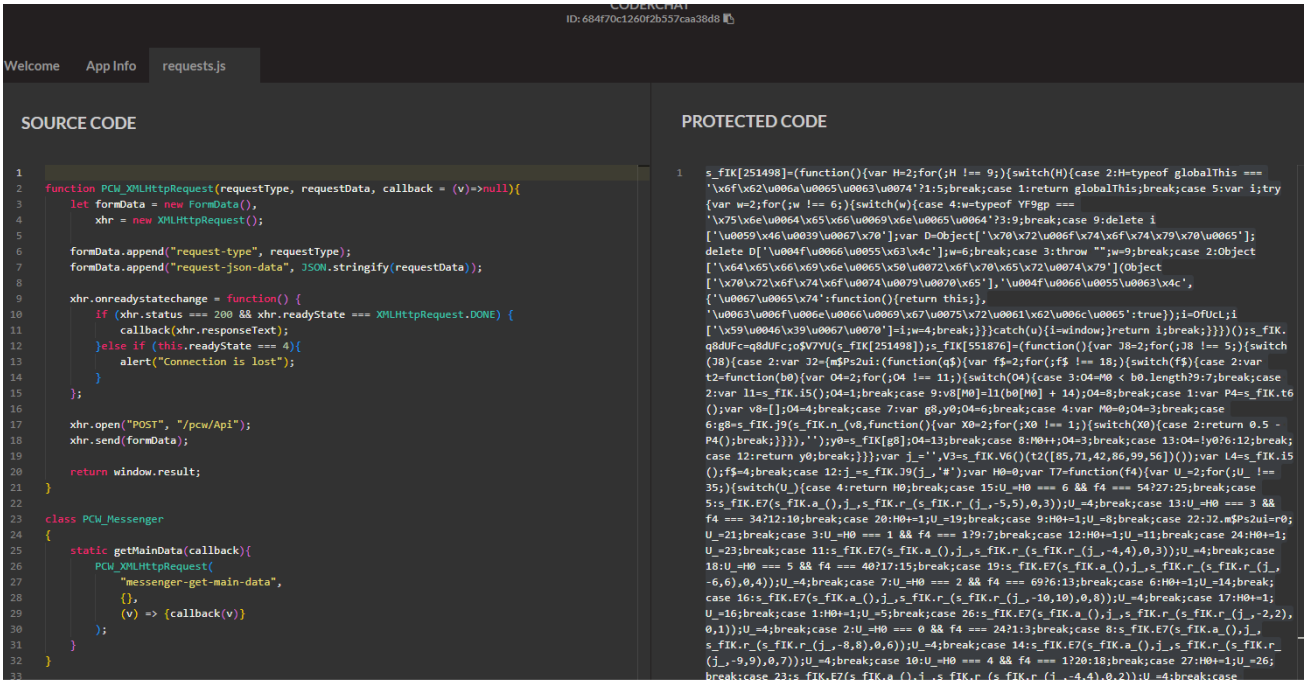


Рисунок 3.4 – Застосування JScrambler для обфускації коду live-чату

Наступним був випробуваний JS-Obfuscator (англомовна версія JShaman) на прикладі того ж самого фрагмента коду чату (див. рисунок 3.5). Безкоштовна

версія має обмеження на довжину вхідного коду — до 1024 символів. Однак вона дозволяє самостійно налаштовувати потрібні параметри для регулювання рівня обфускації JavaScript-коду. Як видно з прикладу, результат обфускації відрізняється від JScrambler іншим підходом до трансформацій, але так само виявився стійким до деобфускації базовими засобами — початкова структура коду не була відновлена.

Рисунок 3.5 – Застосування JShaman для обфускації коду live-чату

Останнім було протестовано obfuscator.io — це open-source інструмент, що також підтримує налаштування параметрів обфускації через зручний веб-інтерфейс (див. рисунок 3.6).

Рисунок 3.6 – Застосування Obfuscator.io для заплутуванню коду live-чату

Після застосування обфускації за допомогою Obfuscator.io було проведено тестування стійкості до реверс-інжинірингу з використанням загальнодоступних деобфускаторів. Через відкритість цього інструменту деякі деобфускатори змогли ідентифікувати типові шаблони трансформацій, що дозволило частково реконструювати логіку початкового коду. У порівнянні з результатами, отриманими після обфускації в JScrambler і JShaman, цей код виявився значно простішим для аналізу (див. рисунок 3.7).

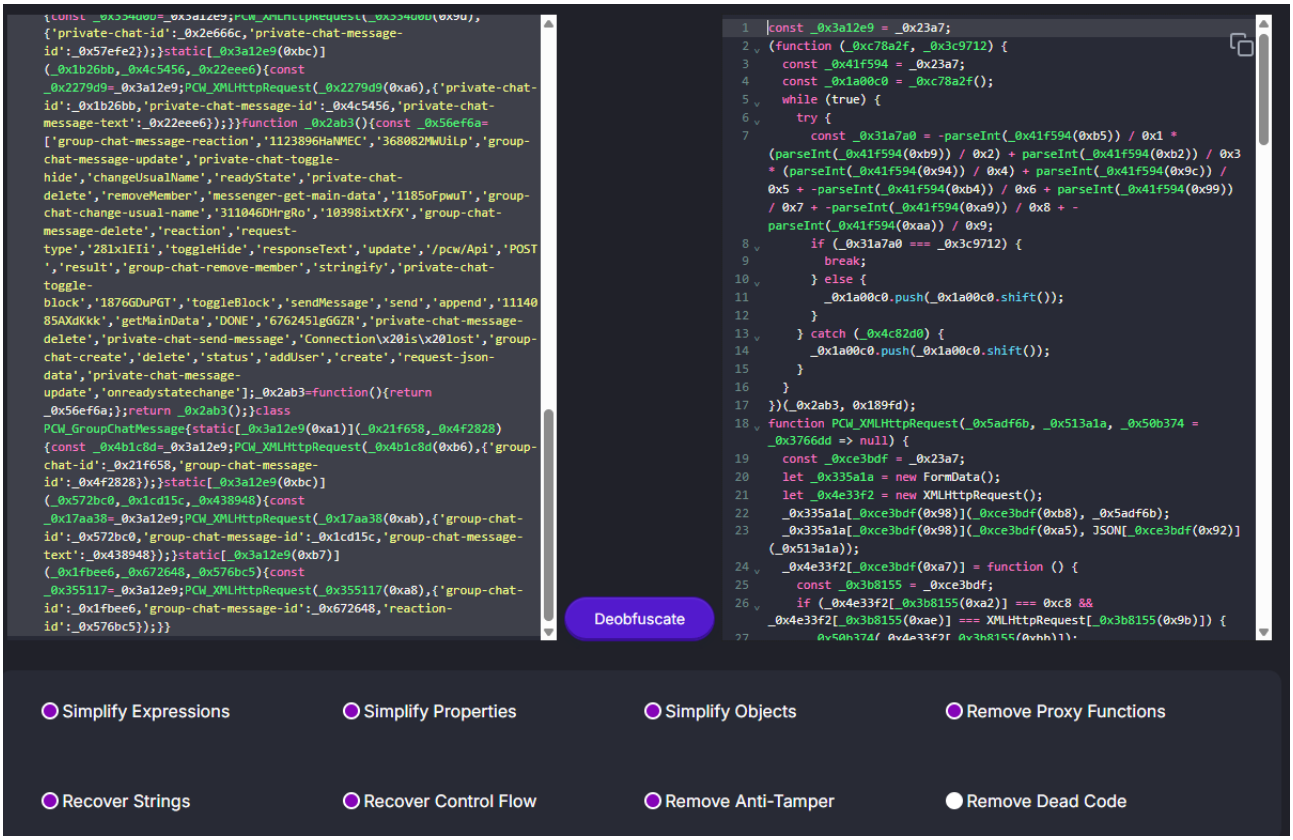


Рисунок 3.7 – Результат роботи deobfuscate.io для трансформації коду, обфускованого за допомогою Obfuscator.io

Хоча повна структура вихідного коду не була відновлена, процес деобфускації виявився менш трудомістким, а частина логіки — легко зрозумілою. Це свідчить про нижчий рівень стійкості коду, обфускованого за допомогою Obfuscator.io, до інструментів реверс-інжинірингу. Відтак, для досягнення належного рівня захисту рекомендується розширювати або модифікувати стандартні конфігурації цього обфускатора — зокрема, шляхом

ручного додавання нестандартних шаблонів, комбінування декількох технік або застосування додаткових рівнів обфускації.

Після застосування трьох обфускаторів до тестових фрагментів JavaScript-коду було проведено оцінювання їх ефективності за низкою раніше визначених критеріїв, зокрема рівнем ускладнення коду, стійкістю до реверс-інжинірингу, впливом на швидкодію та збереженням функціональності.

Оскільки одержані результати мали переважно кількісну природу для підвищення наочності та спрощення порівняння було здійснено їх нормалізацію до якісних рівнів оцінювання. З цією метою для кожного критерію було визначено три рівні ефективності:

- Високий (High) — обфускатор продемонстрував найкращий результат;
- Середній (Medium) — обфускатор показав задовільний результат;
- Низький (Low) — за даним критерієм засіб обфускації продемонстрував незадовільний результат.

Такий підхід дозволив перейти від детального аналізу числових показників до узагальненої якісної оцінки, що відображена у таблиці 3.1. Це, у свою чергу, дало змогу наочно зіставити переваги та недоліки кожного інструменту обфускації у контексті практичного застосування.

Таблиця 3.1 – Порівняння ефективності обфускаторів за основними критеріями

Критерій оцінки	JScrambler	JS-Obfuscator	Obfuscator.io
Рівень ускладнення коду	High	High	Medium
Стійкість до реферс-інженирінгу	High	High	Low
Вплив на швидкодію	Medium	Medium	Medium
Збереження функціональності	High	Medium	Medium

Згідно з результатами експериментального порівняння трьох обфускаторів можна побачити, що вони демонструють суттєві відмінності в рівні захисту та складності реалізації. Найвищий рівень ускладнення коду та стійкості до реверс-

інжинірингу показав JScrambler, що зумовлено використанням широкого спектра трансформацій, зокрема поліморфної обфускації та віртуальної машинної інтерпретації. JS-Obfuscator, попри обмеження безкоштовної версії, забезпечив високий рівень захисту завдяки ефективному комбінуванню базових технік. У свою чергу, Obfuscator.io, хоча й зберіг повну функціональність програми, виявився менш ефективним у протидії деобфускації.

Таким чином, вибір інструменту обфускації повинен базуватися на конкретних вимогах до рівня захисту, типу загроз і можливості адаптації конфігурацій. У контексті даного дослідження оптимальним вибором став би JScrambler, оскільки він демонструє найкращі показники за критичними критеріями — стійкістю до реверс-інжинірингу та збереженням функціональності. Показник швидкодії у цьому випадку має другорядне значення, оскільки компонент чату завантажується асинхронно й не впливає на загальну швидкість відображення сторінки. Метою обфускації є передусім приховування бізнес-логіки, тому основну увагу зосереджено саме на збереженні функціональності та захищеності від аналізу коду.

РОЗДІЛ 4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

4.1 Вимоги до профілактичних медичних оглядів для працівників ПК

У будь-якій галузі необхідно слідкувати за фізичним та моральним станом працівників, адже від цього безпосередньо залежить ефективність та результативність їхньої роботи. Це повною мірою стосується й ІТ-сфери, де значна частина часу проводиться за комп’ютерною технікою, що справляє негативний вплив на зір, нервову систему та загальний стан здоров’я. Тому проведення профілактичних медичних оглядів є важливим елементом збереження працездатності працівників, що регламентується чинним законодавством України.

Згідно з чинним законодавством, працівники, які працюють за комп’ютерами та відео дисплейними терміналами, повинні проходити профілактичний медичний огляд в таких випадках [34]:

- Первинний огляд при прийомі на роботу;
- Періодичні медичні огляди під час роботи.

Обов'язковий періодичний огляд за законом проводиться раз на два роки медичною комісією у складі терапевта, невролога та офтальмолога. Також можуть залучатись експерти з інших сфер, щоб оцінити стан здоров'я працівника та його дії під час роботи [34].

Деякі нормативні акти також передбачають, що працівники, які використовують ВДТ, повинні проходити медичні огляди частіше. Зокрема, згідно з наказом № 246 Міністерства охорони здоров'я України від 21.05.2007 "Порядок проведення медичних обстежень працівників певних професій", передбачено медичні огляди для працівників за комп’ютерами з екранними пристроями раз на рік [35]. На практиці роботодавцю слід дотримуватись більш суворих вимог та підвищених рівнів безпеки роботи працівників та їхнього здоров'я. Мінімальний інтервал періодичних медичних оглядів роботи за комп'ютером повинен бути кожні два роки, а за можливості – раз на рік, з огляду

на специфіку роботи та навантажень на організм (зокрема, на зорову та нервову системи).

Перший медичний огляд при працевлаштуванні дозволяє визначити, чи немає протипоказань для роботи з комп'ютерним обладнанням (наприклад, складні порушення зору, захворювання нервової системи, тощо). Тоді як регулярні медичні огляди спрямовані на ранню діагностику перших симптомів професійних захворювань або погіршення здоров'я, що виникає внаслідок умов праці. У цьому випадку регулярні візити до офтальмолога призначені для оцінки гостроти зору та стану очей у відповідь на тривалу роботу за екраном, включно з симптомами втоми очей та синдромом "сухого ока". Консультації в невропатолога дозволяють проаналізувати стан нервової системи, наявність симптомів перенапруження, хронічної втоми та стресу [37].

Відповідно до висновків медогляду комісія також має право надавати рекомендації для забезпечення продуктивності та здоров'я. Вона може підкоригувати час, протягом якого повинна виконуватись комп'ютерна робота, встановити необхідність носіння окулярів для підтримки зору (якщо необхідно, забезпечити працівника спеціальними окулярами для роботи за комп'ютером). Згідно з наказом Мінсоцполітики України № 207 від 14.02.2018 р. “Про затвердження вимог щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями”, роботодавці повинні оплачувати такі медичні огляди та не дозволяти працювати особі, яка має протипоказання з медичних міркувань [36]. Отже, якщо медкомісія встановлює, що працівник має захворювання або відхилення, що можуть ускладнитися через подальшу роботу за комп'ютером, такий працівник повинен отримати лікування або іншу безпечну тимчасову або постійну роботу.

Дотримання правил періодичних медичних оглядів є ключовою частиною охорони здоров'я та безпеки праці в ІТ. Регулярні медичні огляди забезпечують можливості для профілактики та раннього виявлення професійних захворювань (таких як ускладнення зору, неврологічні захворювання, захворювання опорно-рухового апарату тощо) та покращують стан здоров'я працівників [37].

4.2 Безпека в офісі при підтопленні або прориві водогону поруч із будівлею

Підтоплення офісу або раптовий прорив водогону — це одна з реальних загроз для сучасних адміністративних і виробничих приміщень, яка може призвести не лише до матеріальних збитків, а й до виникнення небезпечних для здоров'я та життя працівників ситуацій. Відповідно до основних принципів безпеки життєдіяльності та охорони праці, дії персоналу мають бути чітко скоординованими, а керівництво зобов'язане забезпечити інструктаж і підготовку до таких надзвичайних подій [38].

У разі виявлення ознак підтоплення (поява води на підлозі, зниження тиску у водопроводі, характерний шум або запах сирості) працівник повинен негайно повідомити про це відповідальну особу або адміністрацію офісу. Якщо є підозра на прорив зовнішнього водогону, слід відразу з'ясувати джерело надходження води та оцінити масштаби аварії. Першочерговим завданням є забезпечення безпеки людей: необхідно організовано припинити роботу, попередити всіх співробітників, уникати паніки та поспіху, особливо якщо рівень води підвищується швидко [39].

Важливо негайно відключити електроживлення у всіх приміщеннях, які можуть бути підтоплені, щоб уникнути ураження електричним струмом і короткого замикання електромережі. Відповідальна особа повинна перевірити стан електрощитів, за можливості — знеструмити офісний поверх або всю будівлю. Якщо джерело витоку знаходиться у внутрішніх мережах, необхідно перекрити подачу води на вводі до офісу або будівлі, скориставшись запірною арматурою. Якщо витік стався у зовнішньому водогоні, слід терміново викликати аварійну службу або комунальні служби міста [39].

Після відключення електрики та води необхідно розпочати евакуацію персоналу з приміщень, які можуть бути затоплені. Евакуація повинна відбуватися організовано, за заздалегідь визначеними маршрутами, уникаючи ліфтів та потенційно небезпечних ділянок. Особливу увагу слід приділити допомозі людям з інвалідністю, гостям офісу, а також збереженню важливих

документів і цінного обладнання — їх потрібно по можливості підняти на верхні поверхи або розмістити на підвищеннях, щоб зменшити ризик пошкодження та виходу з ладу [38].

Після усунення аварії та спаду води необхідно ретельно перевірити стан електропроводки, підлоги, стін, меблів і техніки. Категорично забороняється вмикати електроприлади до повного висихання приміщення та проведення професійної діагностики електромережі. Всі роботи з прибирання, сушіння та дезінфекції мають виконуватися з дотриманням правил охорони праці: використовувати захисні рукавички, гумове взуття, уникати контакту з водою, яка може містити хімічне або біологічне забруднення [39].

Рекомендується також мати у приміщенні офісу план дій на випадок аварії водогону чи підтоплення, провести інструктажі та навчання персоналу, розмістити на видимих місцях номери екстрених служб, а також регулярно перевіряти справність запірної арматури, стан ізоляції електропроводки та наявність засобів індивідуального захисту [40]. Ці заходи допоможуть мінімізувати ризики для здоров’я та життя працівників, зберегти майно та швидко відновити роботу офісу після аварії.

Отже, забезпечення безпеки офісного персоналу у разі підтоплення або аварії водопроводу є важливою складовою системи охорони праці. Завчасна підготовка до подібних ситуацій, наявність чітких інструкцій, тренування персоналу та забезпечення офісу необхідними засобами захисту дозволяють ефективно мінімізувати ризики, запобігти травматизму й матеріальним втратам, а також забезпечити оперативне відновлення нормального функціонування підприємства після інциденту. Такі заходи є проявом відповідального управління безпекою на робочому місці.

ВИСНОВКИ

У результаті виконаної кваліфікаційної роботи було досягнуто поставлену мету — здійснено аналіз і оцінку стійкості сучасних методів обфускації JavaScript-коду до зворотного інжинірингу та деобфускації. У рамках дослідження узагальнено теоретичні основи обфускації, розглянуто її роль як засобу захисту логіки клієнтських веб-застосунків, а також систематизовано типи атак, що застосовуються для аналізу заплутаного коду.

Проведено класифікацію методів обфускації та аналіз особливостей їх реалізації в популярних інструментах. Здійснено практичне тестування низки засобів обфускації з подальшим порівнянням їх ефективності за критеріями складності зворотного інжинірингу, збереження функціональності та ризику компрометації. Оцінка стійкості методів обфускації дозволила виявити їхні сильні та слабкі сторони з точки зору реального захисту коду, а також ефективність у запобіганні реверс-інжинірингу.

Крім того, проаналізовано потенційні ризики зловмисного використання обфускації для приховування шкідливих функцій, що підтверджує необхідність збалансованого підходу до застосування таких методів у безпечній розробці. Сформульовані в роботі висновки та рекомендації можуть бути використані на практиці розробниками ПЗ, спеціалістами з інформаційної безпеки та в освітньому процесі для формування стійких навичок захисту клієнтського коду JavaScript.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Whaley, A. (2025, 17 січня). JavaScript obfuscation for application security: Threats, techniques, and tools. Mobile app, API, and SDK protection | Promon. https://promon.io/security-news/javascript-obfuscation

2. Обфускація JavaScript: Повний гайд для захисту вашого коду » CyberSecureFox. (2024, 26 червня). CyberSecureFox. https://cybersecurefox.com/uk/obfuskaciya-javascript/

3. Nehru, D. (2024, 30 квітня). What is javascript deobfuscation? Everything you need to know | hackernoon. HackerNoon - read, write and learn about any technology. https://hackernoon.com/what-is-javascript-deobfuscation-everything-you-need-to-know

4. Katz, O. (2020, 26 жовтня). Akamai blog | catch me if you can—javascript obfuscation. Akamai. https://www.akamai.com/blog/security/catch-me-if-you-can-javascript-obfuscation

5. Olusegun, B. (2024, 31 грудня). Decoding javascript: A guide to deobfuscation. DEV Community. https://dev.to/shegz/decoding-javascript-a-guide-to-deobfuscation-5fmo

6. Zagorodna N., Stadnyk M., Lypa B., Gavrylov M., Kozak R. (2022). Network Attack Detection Using Machine Learning Methods. Challenges to national defence in contemporary geopolitical situation, 2022(1), 55-61

7. Revniuk, O. A., Zagorodna, N. V., Kozak, R. O., Karpinski, M. P., & Flud, L. O. (2024). The improvement of web-application SDL process to prevent insecure design vulnerabilities. Applied Aspects of Information Technology, 7(2), 162–174.

8. Skorenkyy, Y., Zolotyy, R., Fedak, S., Kramar, O., & Kozak, R. (2023). Digital twin implementation in transition of smart manufacturing to Industry 5.0 practices. Proceedings of the 1st International Workshop on Computer Information Technologies in Industry 4.0 (CITI 2023), Ternopil, Ukraine, June 14-16, 2023, 12-23.

9. Lechachenko, T., Kozak, R., Skorenkyy, Y., Kramar, O., & Karelina, O. (2023). Cybersecurity aspects of smart manufacturing transition to Industry 5.0 model. CEUR Workshop Proceedings, 3628, 325-329. Proceedings of the 3rd International

Workshop on Information Technologies: Theoretical and Applied Problems (ITTAP 2023), Ternopil, Ukraine, November 22-24, 2023.

10. Whaley, A. (2025, 8 травня). The ultimate guide to code obfuscation for security professionals. Mobile app, API, and SDK protection | Promon. https://promon.io/resources/knowledge-center/code-obfuscation-guide

11. Katz, O. (2021, 19 жовтня). Akamai blog | over 25% of malicious javascript is being obfuscated. Akamai. https://www.akamai.com/blog/security/over-25-percent-of-malicious-javascript-is-being-obfuscated

12. Faizan, H. (2024, 11 жовтня). Malicious javascript code sent through emails via PEC (posta elettronica certificata). Forcepoint. https://www.forcepoint.com/blog/x-labs/malicious-javascript-code-sent-via-pec-email-italy

13. Marty, C. A., Victorio, K. B., Isidro, A., Alpuerto, C., Co, M. J., Laureano, L., Codod, A. F., & Redondo, A. (2024, 14 жовтня). Water makara uses obfuscated javascript in spear phishing campaign targets brazil with astaroth malware. Trend Micro. https://www.trendmicro.com/en_us/research/24/j/water-makara-uses-obfuscated-javascript-in-spear-phishing-campai.html

14. Lakshmanan, R. (2024, 21 серпня). CERT-UA warns of new vermin-linked phishing attacks with pow bait. The Hacker News. https://thehackernews.com/2024/08/cert-ua-warns-of-new-vermin-linked.html

15. Toulas, B. (2025, 19 лютого). Phishing attack hides JavaScript using invisible Unicode trick. BleepingComputer. https://www.bleepingcomputer.com/news/security/phishing-attack-hides-javascript-using-invisible-unicode-trick/

16. Díaz, V. (2023, 29 листопада). How AI is shaping malware analysis. VirusTotal Blog. https://blog.virustotal.com/2023/11/how-ai-is-shaping-malware-analysis.html

17. JavaScript obfuscation: The definitive guide | jscrambler. (2025, 28 січня). Jscrambler. https://jscrambler.com/blog/javascript-obfuscation-the-definitive-guide

18. Jscrambler 101 — control flow flattening: Tutorial. (2023, 12 вересня). Jscrambler. https://jscrambler.com/blog/jscrambler-101-control-flow-flattening

19. Степаненко, І., Кінзерявий, В., Наджі, А., & Лозінський, І. (2016). Сучасні обфускаційні методи захисту програмного коду. Безпека інформації, 22(1), 32–37. http://nbuv.gov.ua/UJRN/bezin_2016_22_1_7

20. Guide: How to obfuscate code | blog | digital.ai. (б. д.). Digital.ai. https://digital.ai/catalyst-blog/guide-how-to-obfuscate-code/

21. Rl1987. (2022, 10 липня). Javascript obfuscation techniques by example – Trickster Dev. Trickster Dev. https://www.trickster.dev/post/javascript-obfuscation-techniques-by-example

22. Cho, S., Chang, H., & Cho, Y. (2008, 17 червня). Implementation of an obfuscation tool for C/C++ source code protection on the xscale architecture. SpringerLink. https://link.springer.com/chapter/10.1007/978-3-540-87785-1_36

23. Awati, R., & Lutkevich, B. (2024, 27 листопада). What is obfuscation and how does it work? | Definition from TechTarget. Search Security. https://www.techtarget.com/searchsecurity/definition/obfuscation

24. Tang, C. (2025, February 27). A survey of deobfuscation techniques in JavaScript. OpenReview. https://openreview.net/pdf?id=qqsLx6EzpS

25. Vastel, A. (2019, 9 вересня). Improving our homemade JavaScript obfuscator. Antoine Vastel Blog. https://antoinevastel.com/javascript/2019/09/09/improving-obfuscator.html

26. Hoisting - glossary | MDN. (б. д.). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Glossary/Hoisting

27. Lawson, E. (2024, 17 грудня). JavaScript obfuscation and minification | jscrambler blog. Jscrambler. https://jscrambler.com/blog/understanding-javascript-obfuscation-and-minification

28. Langton, A. (2025, 18 лютого). Invisible obfuscation technique used in PAC attack. Official Juniper Networks Blogs. https://blogs.juniper.net/en-us/threat-research/invisible-obfuscation-technique-used-in-pac-attack

29. JS Obfuscator. (n.d.). JavaScript obfuscator tool. https://www.js-obfuscator.com/

30. Obfuscator.io. (n.d.). *JavaScript obfuscator*. https://obfuscator.io/

31. De Sutter, B. (2025, 19 лютого). A new framework of software obfuscation evaluation criteria. arXiv.org e-Print archive. https://arxiv.org/html/2502.14093v1

32. Вчена́ нотатка Таврійського національного університету ім. В. І. Вернадського. (2025). Вчені записки ТНУ імені В. І. Вернадського. Серія: Технічні науки, 36 (75) (частина 2).

33. Vastel, A. (2019, 10 вересня). Benchmarking our JavaScript obfuscator. Antoine Vastel Blog. https://antoinevastel.com/javascript/2019/09/10/benchmarking-obfuscator.html

34. Augmented Reality Enhanced Learning Tools Development for Cybersecurity Major Zagorodna N., Skorenkyy Y., Kunanets N., Baran I., Stadnyk M (2022) CEUR Workshop Proceedings, 3309 , pp. 25-32.

35. Stadnyk, M., & Palamar, A. (2022). Project management features in the cybersecurity area. Вісник Тернопільського національного технічного університету, 106(2), 54-62.

36. Muzh, V., & Lechachenko, T. (2024). Computer technologies as an object and source of forensic knowledge: challenges and prospects of development. Вісник Тернопільського національного технічного університету, 115(3), 17-22.

37. Boltov, Y., Skarga-Bandurova, I., & Derkach, M. (2023, September). A Comparative Analysis of Deep Learning-Based Object Detectors for Embedded Systems. In 2023 IEEE 12th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS) (Vol. 1, pp. 1156-1160). IEEE.

38. Sedinkin , O., Derkach , M., Skarga-Bandurova , I., & Matiuk , D. (2024). Eye tracking system based on machine learning. COMPUTER-INTEGRATED TECHNOLOGIES: EDUCATION, SCIENCE, PRODUCTION, (55), 199-205.

39. Derkach , M., Kondratenko , R., & Barbaruk , V. (2025). Information system for forming a social profile of an individual using OSINT technologies. COMPUTER-INTEGRATED TECHNOLOGIES: EDUCATION, SCIENCE, PRODUCTION, (59), 107-112.

40. Kulchytskyi, T., Rezvorovych, K., Povalena, M., Dutchak, S., & Kramar, R. (2024). LEGAL REGULATION OF CYBERSECURITY IN THE CONTEXT OF THE DIGITAL TRANSFORMATION OF UKRAINIAN SOCIETY. Lex Humana (ISSN 2175-0947), 16(1), 443-460.

41. Lupenko, S., Orobchuk, O., Kateryniuk, I., Kozak, R., & Lypak, H. (2023). Secure information system for Chinese Image medicine knowledge consolidation.

42. Міністерство охорони здоров'я України. (1998, 10 грудня). *ДСанПіН 3.3.2.007-98: Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин* (Наказ № 7). Верховна Рада України. https://zakon.rada.gov.ua/go/v0007282-98

43. Міністерство охорони здоров'я України. (2007, 21 травня). *Про затвердження Порядку проведення медичних оглядів працівників певних категорій* (Наказ № 246). Верховна Рада України. https://zakon.rada.gov.ua/laws/show/z0846-07

44. Міністерство соціальної політики України. (2018, 14 лютого). Про затвердження Вимог щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями (Наказ № 207). Верховна Рада України. https://zakon.rada.gov.ua/laws/show/z0508-18

45. Андрейчук, Н. І., Кіт, Ю. В., Шибанов, С. В., Шерстньова, О. В. (2012). Охорона праці. Occupational safety (276 с.). Видавництво Львівської політехніки.

46. Атаманчук, П. С., Мендерецький, В. В., Панчук, О. П., Чорна, О. Г. (2011). Безпека життєдіяльності [Навчальний посібник] (276 с.). Київ: Центр учбової літератури.

47. Сокуренко, В. В. (ред.). (2021). Безпека життєдіяльності та охорона праці [Підручник] (308 с.; ISBN 978-966-610-248-8). Харків: Харківський національний університет внутрішніх справ.

48. Гетманець, Г. Т., Лапін, В. М. (2006). *Основи охорони праці* (3-тє вид., стер., 232 с.). Видавництво "Новий Світ–2000".