МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

ВИПУСКНА РОБОТА

на тему:

«Алгоритм обфускації JavaScript коду з використанням стандарту кодування base64»

Завідувач кафедри	**Довбиш А.С.**
Керівник роботи	**Проценко О.Б.**
Студент гр. ІН-72	**Маландій А.Є.**

СУМИ 2021

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Кафедра комп’ютерних наук

Затверджую___________

Зав. кафедри Довбиш А.С.

“_____” ____________2021 р.

ЗАВДАННЯ
до випускної роботи

Студента четвертого курсу, групи ІН-72 спеціальності “Інформатика” денної форми навчання Маландія Артема Євгеновича.

Тема: “**Алгоритм обфускації JavaScript коду з використанням стандарту кодування base64** ”

Затверджена наказом по СумДУ

№________ від ___________2021 р.

Зміст пояснювальної записки: 1) аналітичний огляд методів обфускації; 2) постановка завдання й формування завдань; 3) опис основних положень, концепція методу обфускації; 5) розробка алгоритму обфускації; 6) аналіз отриманих результатів.

Дата видачі завдання “_____”____________2021 р.

Керівник випускної роботи __________ Проценко Б.О.

Завдання приняв до виконання____________Маландій А.Є.

РЕФЕРАТ

Записка: 54 стор., 44 рис., 2 табл., 4 додатка, 10 джерел.

Об'єкт дослідження — методи обфускації коду.

Мета роботи — розробка алгоритму обфускації JavaScript коду з використанням стандарту кодування base64

Методи дослідження — метод неявного приведення типів даних та використання побітових операторів.

Результати — розроблено алгоритм обфускації JavaScript коду. При цьому час виконання обфускованого скрипту не більше ніж до обфускації. Розроблений алгоритм реалізовано у формі веб-додатку, створеного за допомогою таких мов програмування як: HTML, CSS, JavaScript та PHP.

ОБФУСКАЦІЯ, МІНІФІКАЦІЯ, СТАНДАРТ КОДУВАННЯ BASE64, ПОБІТОВІ ОПЕРАТОРИ, ОСОБЛИВОСТІ НЕЯВНОГО ПРИВЕДЕННЯ ТИПІВ ДАНИХ.

Зміст

ВСТУП **2**

1 ІНФОРМАЦІЙНИЙ ОГЛЯД **3**

1.1 Огляд існуючих рішень 3

1.2 Методи мініфікації коду 4

1.3 Алгоритми обфускації коду 10

1.4 Постановка задачі 19

2 ВИБІР МЕТОДУ РІШЕННЯ **21**

2.1 Основні положення 21

2.2 Вибір мов програмування 24

2.3 Концепція методу обфускації 25

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ **27**

3.1 Проектування макету 27

3.2 Серверна частина 30

3.3 Клієнтська частина 35

3.4 Тестування та аналіз результатів роботи 40

ВИСНОВКИ **44**

СПИСОК ЛІТЕРАТУРИ **45**

ДОДАТКИ **46**

ДОДАТОК А 46

ДОДАТОК Б 48

ДОДАТОК В 50

ДОДАТОК Г 53

ВСТУП

Інтернет-простір є найпопулярнішим джерелом інформації та засобом зв'язку для багатьох людей у всьому світі. Він охоплює велику аудиторію та швидко зростає. Станом на січень 2021 року кількість користувачів Інтернету досягнуло 4.66 мільярди людей, що на 316 мільйонів більше ніж минулого року. Здебільшого це обумовлено широким розповсюдженням стільникових мереж з доступом до інтернет-стандартів 3G, 4G, а у найближчому майбутньому ще й 5G, розвитком соціальних мереж та здешевленням вартості інтернет-трафіку. Такий стрімкий розвиток породжує ризик зниження інформаційної безпеки. Виняткова важливість забезпечення захисту додатку від несанкціонованого доступу обумовлює поширення та постійне удосконалення засобів захисту інформації та розширення їх номенклатури. На сьогоднішній день цю задачу можна забезпечити за рахунок цілого комплексу різносторонніх рішень.

У сучасному світі існують безліч загроз викрадення програм та інтелектуальної власності, порушення авторських прав або недобросовісність замовника. Тому проблема зберігання прав власності на вихідний код достатньо актуальна. Дані проблеми зв'язані з доступністю розроблених програмних засобів та їх виконуваного коду. Існують різні способи боротьби з кіберзагрозами. Наприклад, можна зберігати код програми на захищеному сервері, а результат обчислень передавати по запиту від клієнта, або використовувати код, що компілюється.

На відміну від настільних додатків, у яких існують різні засоби захисту та ліцензування, у веб-додатках такі способи захисту не застосовні і весь вихідний код скриптів доступний для перегляду будь-якій людині через браузер. Найкращим варіантом захисту коду є застосування обфускації. Сьогодні існують різні програмні продукти, які пропонують обфускацію вихідного коду, але ефективні алгоритми обфускації пропонують тільки платні продукти. Безкоштовні працюють здебільшого на рівні мініфікації коду [1].

1 ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1 Огляд існуючих рішень

На даний момент існують 2 способи ускладнити розуміння та аналіз вихідного JavaScript коду:

1. Мініфікація;
2. Обфускація.

Обфускація та мініфікація – два терміни, які частіше всього використовують у програмуванні, особливо у програмування на JavaScript. У програмуванні мініфікація відноситься до техніки, за допомогою якої непотрібні символи видаляються з вихідного коду, зберігая при цьому ті ж функціональні можливості, що й до процесу мініфікації. З іншої сторони, обфускація відноситься до модифікації файлів таким чином, що їх стає важко читати та розуміти. Отже, обидва методи мають різні цілі досягнення.

Мініфікація визначає процес, в результаті якого файли модифікуються шляхом видалення усіх зайвих символів у файлах. Цей процес робить файли легшими, а це допомагає підвищити продуктивність та швидкість завантаження файлу клієнту. Але важливо зберегти функціональність початкового файлу без змін. У більшості великих проектах мініфікація підтримується у процесі збору додатку.

Цей процес зазвичай виконується шляхом аналізу коду і подальшого його виводу у стислому вигляді. Код зазвичай стає нечитабельним неозброєним оком. Він видаляє увесь непотрібний код, такий як пробіли, коментарі та символи нового рядка. Окрім отримання файлу меншого розміру, процес мініфікації також перевіряє правильність коду, так як неправильний код не аналізується и не мінімізується належним чином.

1.2 Методи мініфікації коду

На сьогоднішній день існують безліч мініфікаторів. Вони можуть бути як у вигляді веб-додатку, настільного додатку так і можуть бути вбудовані в інтегроване середовище розробки. Серед усіх мініфікаторів, широко розповсюджені ось такі:

1. Google Closure Compiler
2. UglifyJS
3. Microsoft AJAX Minifier

Ці мініфікатори відрізняються як способом встановлення, так і алгоритмом мініфікації. Наприклад, Google Closure Compiler потребує попереднього встановлення Java, а UglifyJS – Node.js. Google Closure Compiler також включає «пісочницю» для тестування стиснення та веб-сервіс, на який можна відправити код для стиснення.

Як працює мініфікатор? Усі сучасні мініфікатори працюють наступним чином:

1. Розбирають JavaScript-код у синтаксичне дерево.

Так робить будь-який інтерпретатор JavaScript перед тим як виконувати код. Але потім, замість виконування коду…

2. Проходить по цьому дереву, аналізує його та оптимізує.
3. Записує з синтаксичного дерева отриманий код.

Наприклад у нас є даний код (рис. 1.1):

```
1: function User(name) {
2:   this.sayHi = function() {
3:     alert(name);
4:   };
5: }
```

Рисунок 1.1 - Приклад коду

Відповідно синтаксичне дерево буде мати такий вигляд (рис 1.2)

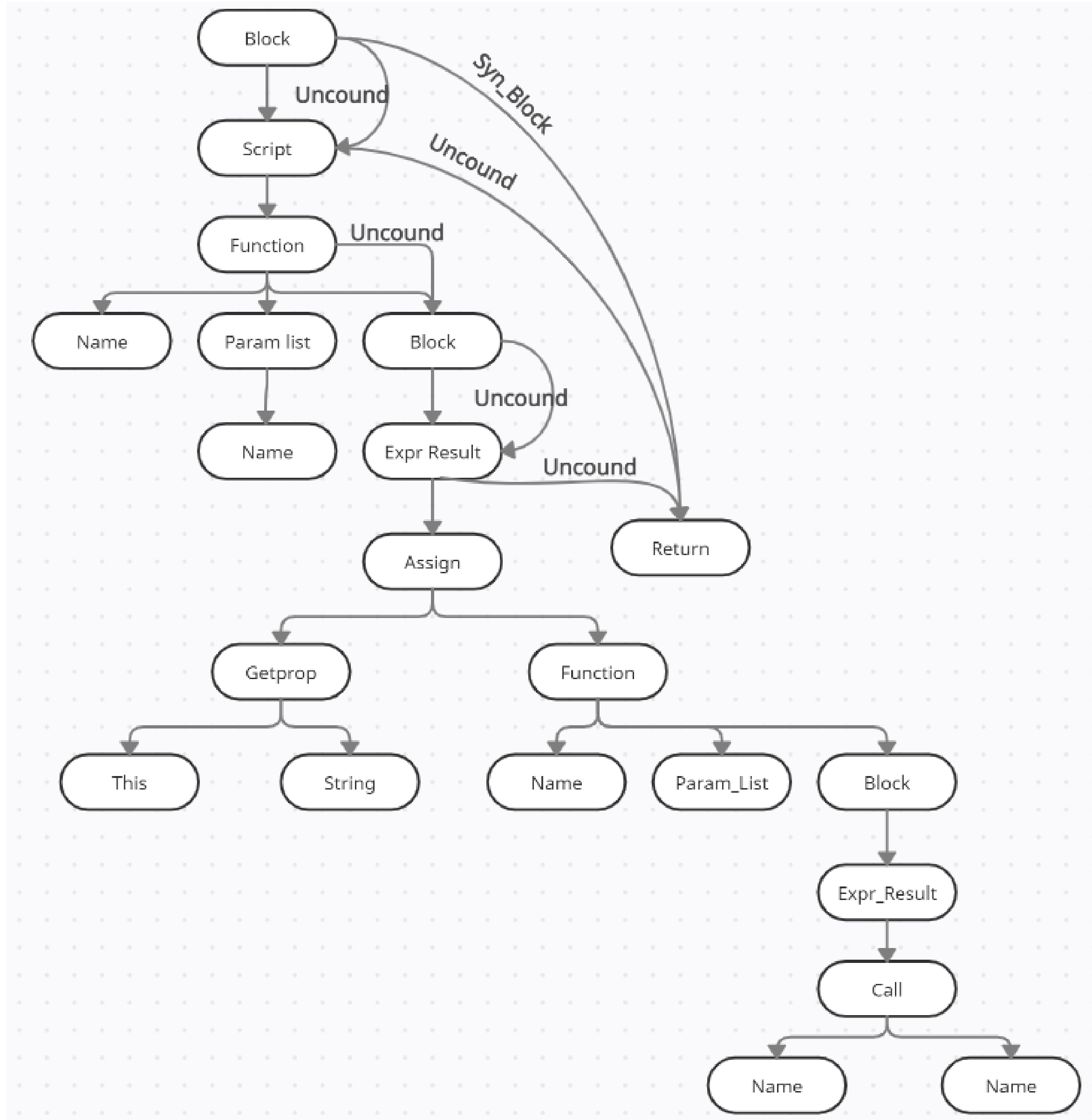

Рисунок 1.2 - Приклад синтаксичного дерева

В вузлах еліпсах стоїть тип, наприклад *Function* (функція) або *Name* (ім’я змінної). Крім цього до кожного вузла прив’язані конкретні дані. Мініфікатор вміє «ходити» по цьому дереву та змінювати його як забажає.

Зазвичай, коли код перетворюється на дерево, то з нього за замовчуванням зникають коментарі та пробіли. Вони не мають значення при виконанні тому ігноруються. Але, наприклад, Google Closure Compiler додає у дерево інформацію з коментарів JSDoc, тобто коментарів типу /** … */.

Наприклад (рис. 1.3):

```
1:  /**
2:  * Номер мінімальної підтримуваної версії Chrome
3:  * @const
4:  * @type {number}
5:  */
6:  let minChromeVersion = 20;
```

Рисунок 1.3 - Коментарі JSDoc

Такі коментарі не створюють нових вузлів дерева, а додаються у якості інформації до існуючих. В даному випадку до змінної *minChromeVersion.* В них може міститись інформація про тип змінної *{number}* та інша інформація, яка допоможе мініфікатору краще оптимізувати код. Мініфікатор проходить по дереву та шукає патерни – відомі йому структури, які він знає, як оптимізувати, та оновлює дерево.

У різних мініфікаторів реалізований різний набір оптимізацій, вони застосовуються у різному порядку, тому результат роботи може бути різний. На прикладі мініфікатора Google Closure Compiler розберемо деякі методи оптимізації.

Методи оптимізації мініфікаторів:

1. Об’єднання та стиснення констант

До оптимізації (рис 1.4):

```
1:  function foo (x, y) {
2:      console.log (x, “hello” + “world”, 600 * 600 * 5, y && 0, 1 &&
    0);
3:  }
```

Рисунок 1.4 - До оптимізації об'єднання та стиснення констант

Після (рис 1.5):

```
1:  function foo(x,y){console.log(x,“helloworld”,18E5,y&&0,0)}
```

Рисунок 1.5 - Після оптимізації об'єднання та стиснення констант

- “hello” + “world” → “helloworld”

- 600 * 600 * 5 → 18E5 (наукова форма числа)
- y && 0 → без змін, так як результат залежить від змінної
- 1 && 0 → 0

2. Скорочування локальних змінних

До оптимізації (рис 1.6):

```
1:  function saying (name, message) {
2:      console.log (name + " said: " + message);
3:  }
```

Рисунок 1.6 - До оптимізації скорочування локальних змінних

Після оптимізації (рис 1.7):

```
1:  function saying(a, b){console.log(a+" said: "+b)}
```

Рисунок 1.7 - Після оптимізації скорочування локальних змінних

- Локальні змінні доступні тільки всередині функції, тому зазвичай її перейменування безпечне
- Також перейменовуються і локальні функції
- Вкладені функції обробляються правильно

3. Об’єднання та видалення локальних змінних

До оптимізації (рис 1.8):

```
1:  function foo (id) {
2:      let elem = document.getElementById(id);
3:      let parent = elem.parentNode;
4:      console.log(parent);
5:  }
```

Рисунок 1.8 - До оптимізації об'єднання та видалення локальних змінних

Після оптимізації (рис 1.9):

```
1: function foo(a){a=document.getElementById(a).parentNode;
2: console.log(a)}
```

Рисунок 1.9 - Після оптимізації об'єднання та видалення локальних змінних

- Локальні змінні бути перейменовані
- Зайві пробіли були видалені. Для цього мініфікатор створює допоміжну внутрішню структуру даних, у якій зберігається інформація про «шлях використання» кожної змінної. Якщо одна змінна завершує свій шлях і починає друга, то можна дати їм одне ім’я
- Крім цього, операції *elem = getElementById* та *elem.parentNode* об’єднані.

4. Видалення недосяжного коду, розгортання if-гілок

   До оптимізації (рис 1.10):

```
1:  function foo (node) {
2:      let par = node.parentNode;
3:      if(0){
4:         alert(“unreachable code”);
5:      }else{
6:         alert(“hello”);
7:      }
8:    return;
9:  alert(2);
10:  }
```

Рисунок 1.10 - До оптимізації видалення недосяжного коду та розгортання if-гілок

Після оптимізації (рис 1.11):

```
1: function foo(){alert(“hello”)}
```

Рисунок 1.11 - Після оптимізації видалення недосяжного коду та розгортання if-гілок

- Якщо створюється змінна, але не використовується, вона може бути видалена. У прикладі ця оптимізація була застосована до змінної *par,* а потім до параметру *node*
- Завідома хибна гілка *if(0) {…}* видалена, завідомо істинна – залишена.

Те ж саме буде з умовами в інших конструкціях, наприклад *x = true ? b : k* перетвориться на *x = b*

- Код після *return* буде видалений, як недосяжний

5. Перепис синтаксичних конструкцій

   До оптимізації (рис 1.12):

```
 1:   let counter = 0;
 2:   while (counter++ < 10){
 3:     console.log(counter);
 4:   }
 5:   if(counter){
 6:     console.log(counter);
 7:   }
 8:    if(counter == ‘1’){
 9:      alert(1);
10:   } elseif (i == ‘2’){
11:      alert(2);
12:   } else {
13:      alert(counter)
14:   }
```

Рисунок 1.12 - До оптимізації перепису синтаксичних конструкцій

Після оптимізації (рис 1.13):

```
 1:   for(let counter=0;10>counter++;)console.log(counter);
 2:   counter&&console.log(counter);”1”==counter?alert(1):”2”==i?
      alert(2):alert(counter)
```

Рисунок 1.13 - Після оптимізації перепису синтаксичних конструкцій

- Конструкція *while* переписана у *for*
- Конструкція *if(counter)…* переписана в *counter&&…*
- Конструкція *if(cond) … else …* була переписана в *cond ? … : …*

6. Інлайнінг функцій

Інлайнінг функції – прийом оптимізації, при якому функція замінюється на своє тіло.

До оптимізації (рис 1.14):

```
 1:  function sayHello(message){
 2:     let elem = createMessage('div', message);
 3:     showElement(elem);
 4:     function createMassage(tagName, message){
 5:        let el = document.createElement(tagName);
 6:        el.innerHTML = message;
 7:        return el;
 8:     }
 9:     function showElement(elem){
10:        document.body.appendChild(elem);
11:     }
12:  }
```

Рисунок 1.14 - До оптимізації інлайнінгу функцій

Після оптимізації (рис 1.15):

```
1:  function sayHello(b){let a=document.createElement("div");
2:  a.innerHTML=b; document.body.appendChild(a)}
```

Рисунок 1.15 - Після оптимізації інлайнінгу функцій

- Виклики функцій *createMessage* та *showElement* замінені на тіло функції. В даному випадку це можливо, так як функції використовуються всього один раз
- Дана оптимізація виконується не завжди. Якби кожна функція використовувалась багато разів, то з точки зору розміру вигідніше залишити код без змін.

1.3 Алгоритми обфускації коду

Для обходу захисту програми в більшості випадків потребується вивчення деякого готового приладу або програмного коду, а також документація на нього, щоб зрозуміти принцип роботи. Цей процес називається «реверсивна інженерія»

Найвідомішим та найпопулярнішим способом захисту від реверс-інжинірингу є обфускація. Обфускацією програми називається деяке її перетворення, яке зберігає функціональність, але при цьому надає програмі таку форму, що вилучення програмного коду, ключової інформації щодо алгоритмів

та структур даних, які були реалізовані у програмі, стає трудомістким завданням [2].

Процес перетворення вихідного коду є процесом обфускації, якщо від задовольняє наступним вимогам:

- Візуально код модифікованої програми відрізняється від вихідного коду, але при тих самих вхідних даних повертає той самий результат.
- Аналіз модифікованого коду більш складний та трудомісткий, ніж аналіз вихідного коду.
- При кожному процесі модифікації результуючий код є різним.
- Зворотне перетворення коду є неефективним.

Ціль обфускації програмного коду полягає в тому, щоб ускладнити розуміння та аналіз програмного коду та перешкодити цілеспрямованої його модифікації. Обфускованою програмою називається програма, яка після застосування обфускуючих перетворень на всіх допустимих для вихідної програми вхідних даних видає той же самий результат, що ж оригінальна програма, але більш складна для аналізу, розуміння та модифікації [3].

Виділяють наступні рівні процесу обфускації:

- Нижчий рівень, коли процес обфускації виконується над асемблерним кодом програми, або навіть безпосередньо над двійковим файлом програми, яка зберігає машинний код
- Вищий рівень, коли процес обфускації виконується над вихідним кодом програми, написаному на мові високого рівня [4].

Здійснення обфускації на нижчому рівні вважається менш складним процесом, але при цьому більш важко реалізованим по ряду причин. Одна з таких причин полягає в тому, що повинні бути враховані особливості роботи більшості процесорів, так як спосіб обфускації, який застосовується на одній архітектурі, може виявитись неприйнятним для другої.

Також іноді може бути недоцільним обфускувати весь код програми (наприклад, через те, що в результаті може значно зменшитись час виконання

програми), в таких випадках бажано проводити обфускацію тільки на найбільш важливих ділянках коду.

JavaScript є об'єктно-орієнтовною мовою програмування. Проте використане в мові прототипування обумовлює відмінності у роботі з об'єктами порівняно з традиційно клас-орієнтованими мовами. Крім того, JavaScript має ряд властивостей властивих функціональним мовам – функції як об'єкти першого класу, об'єкти як списки, каррінг, анонімні функції, замикання – що додає мові додаткову гнучкість[5].

Існують мінімум три особливості мови JavaScript, які роблять її дійсно унікальною:

- Повна інтеграція з HTML та CSS
- Прості речі робляться просто
- Підтримується всіма браузерами та ввімкнений за замовчуванням

Дивлячись на те, що цих трьох особливостей немає більше ні в одній браузерній технології, JavaScript є широко розповсюдженим засобом створення браузерних інтерфейсів.

Існують багато методів обфускації JavaScript-програм, але вони мають недоліки та вразливості. Тому задача вдосконалення засобів обфускації програм на JavaScript є актуальною.

Усі сучасні методи обфускації використовують алгоритм, який базується на послідовному застосуванні декількох функцій перетворення вихідного коду.

Схематично даний алгоритм можна зобразити наступним чином (рис 1.16)

Основні відмінності між методами полягають у внутрішніх алгоритмах – алгоритмах функцій перетворення. Розглянемо це на прикладі алгоритму, який лежить в основі функції перетворення логічних виразів. У більшості випадків використовується алгоритм, сенс якого полягає у тому, щоб замість вихідного оператора використовувались заперечення протилежного оператора с заперечними операндами.

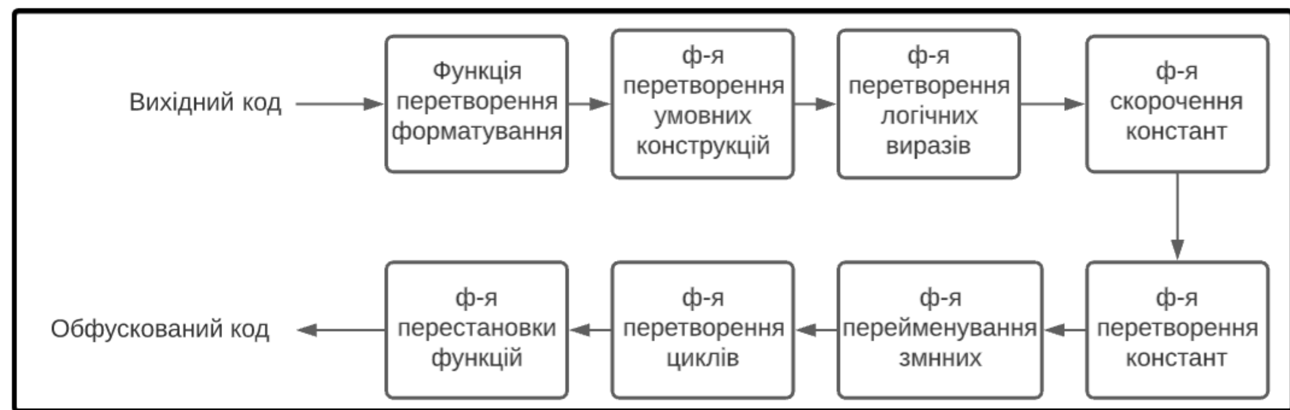

Рисунок 1.16 – Алгоритм послідовного застосування функцій

Необхідно відмітити, що в JavaScript результатом логічного виразу може бути не тільки *true* або *false,* але ще й об'єкт, тому використання більш складних формул перетворення може приводити до помилок.

Таким чином, на прикладі алгоритму функції перетворення логічних виразів, легко помітити, що існуючі методи обфускації не враховують більшість конструкцій синтаксису мови JavaScript останніх версій.

Взагалі існують два етапи обфускації клієнтської частини веб-додатку: створення обфускованого коду та його розшифровка для нормального виконання браузером. Генерувати обфускований код можна на бекенді. А вже розшифровку коду потрібно писати на JavaScript, у такому випадку скрипт сам себе розшифрує та виконає.

Способи обфускації JavaScript:

1. Зберігати код у cookie

У даному випадку код буде зберігатися як звичайний текст, а виконуватись за допомогою функції *eval()*, яка приймає в якості аргументу текст та виконує його як JavaScript-код. Припустимо, що у нас вже встановлені cookie наступного виду:

```
1:  cookieValue=||console.log("hello");||
```

Тепер можемо виконати код наступним чином:

```
1:  eval(unescape(document.cookie).split(“||”)[1]);
```

Тут ми беремо текст усіх cookie записів для нашого хосту та ділимо його на частини у місцях, де стоїть «//». Потім беремо другий елемент ([1]) і передаємо його в якості аргументу функції *eval().* Даний спосіб непоганий, так як код, який ми хочемо виконати, непомітний на самій сторінці, а також тому, що мі можемо змусити код видалити самого себе.

2. Ховати код у HTML та коментарях

Код можна легко заховати в HTML, потім обробити та виконати. Наприклад (рис 1.17):

```
2:  <body><img src=”./cat.png” x=”al” y=”er” z=”t(2+2);”></body>
3:  <script>x = document.body.innerHTML;
    eval(x.split(‘x=”’)[1].split(‘”’)[0]+x.split(‘y=”’)[1].split(‘”’)[0
    ]+a.split(‘z=”’)[1].split(‘”’)[0]);</script>
```

Рисунок 1.17 - Код у HTML

В даному випадку ми сховали код у атрибутах тегу *img,* після чого обробили код усієї сторінки, збираючи розкидані частинки. Таким же чином можна ховати код і в коментарях.

Також варто відмітити, що достатньо ефективно буде ховати код всередину популярних фреймворків, наприклад jQuery. Ці файли не є підозрілими, а їх дослідження забере багато часу (хоча завжди існує можливість автоматичного порівняння оригіналу та заміненого файлу)

3. Субституція стандартних функцій/методів JavaScript

Даний метод орієнтований на те, щоб замість стандартних функцій або методів JavaScript підставляти свої змінні. Наприклад:

До субституції (рис 1.18):

```
1:   document.getElementsByTagName(‘html’)[0].innerHTML =
     document.getElementsByTagName(‘body’)[0].length
```

Рисунок 1.18 - До субституції

Після субституції (рис 1.19):

```
1:   a = document; b = ‘getElementByTagName’; a[b](‘html’).innerHTML =
     a[b](“body”)[0].length;
```

Рисунок 1.19 - Після субституції

В даному випадку ми замінили об’єкт «document» змінною «a», а метод «getElementsByTagName» змінною «c». Слід зауважити, що методи можна також замінювати ключем у масиві (якщо розглядати об’єкт як масив). Якщо у нас є об’єкт «document», а у ньому є елемент «getElementsByTagName», це значить, що ми можемо викликати його обома способами:

- document.getElementsByTagName
- document[“getElementsByTagName”]

Отже, у другому способі використовуються рядкові дані, тому їх можна замінити на зміну, яка містить у собі рядок – назва методу. Субституція корисна у випадку частого використання одного й того ж стандартного об’єкту/функції/змінної. Це сильно змінює код, а також стискує його.

4. Флуд коментарями та кодом

Даний спосіб розрахований на те, щоб вставити у обфускований код флуд, те що не несе смислового навантаження для скрипту. Флудити можна як кодом, так і коментарями.

Флуд коментарями (рис 1.20):

```
1:   /* dfsdfsdf */ let /* sdfd dsfsdfds */ a /* fdsfdsf sd */
2:   = /* dfsdfsdfsdf */ 10 /* fsdf dsfsdfdsfdf */ ; /* dfsdfsdf */
     alert /* sdasdasfgt */ ( /* dfsdfdsfsdf */ a /* dfsdghybgfd */ ) /*
     shyntwefc v */ ; /* dsfdsf sdfd */
```

Рисунок 1.20 - Флуд коментарями

Флуд кодом (рис 1.21):

```
1:   fsdsdf = ‘fdsfsdf’; rgbrgb=’dfgdfgdf’; v = ‘hello’; cewfvev =
     ‘ntnht’; alert(v); c=fsdgbrtbr;
```

Рисунок 1.21 - Флуд кодом

В даному випадку флуд коментарями занадто щільний, насправді код виглядає таким чином: *let a = 10; alert(a);* Флуд кодом можна використовувати разом з флудом коментарями. Також можна використати багаторядкові коментарі у формі фейкового закриття (рис 1.22):

```
1:   /*/ console.log(3) /*/ alert(123); /*/ alert(1) /*/
```

Рисунок 1.22 - Коментарі у формі фейкового закриття

Дійсний код у цьому прикладі: *alert(123);*

При такому варіанті код стає важче аналізувати, але це не є ефективним методом, тому що використовуючи навіть найпростіші мініфікатори, код позбавиться цього флуду.

5. Заміна тексту шістнадцятирічними кодами (рис 1.23)

```
1:   alert(document["\x67\x65\x74\x45\x6c\x65\x6d\x65\x6e\x74\x42\x79\x4
     9\x64"])
```

Рисунок 1.23 -1 Заміна тексту шістнадцятирічними кодами

В даному випадку ми використали метод «getElementById()» об'єкту «document», як елемент масиву. Також перевели ім'я методу у шістнадцятирічний формат. Якби ми використовували метод через крапку «document.getElementById()», то ми б не змогли перевести звернення до нього у шістнадцятирічний формат, так як у масиві ключ є рядком. Цей метод має великий плюс у тому, що для декодування рядка не потрібно створювати функції-розшифровувачі, так як такий формат браузер розуміє і автоматично дешифрує рядок.

6. Числа за допомогою побітових операторів та неявного перетворення типів даних

Існують декілька побітових операторів:

- & (побітове AND) – порівнює кожне бінарне число біт двох чисел та повертає нове ціле число, зі значенням 1, якщо обидва числа дорівнюють 1, у протилежному випадку 0

- | (побітове OR) – порівнює кожний біт двох цілих чисел та повертає 1, якщо хоча б один з них дорівнює 1
- ~ (побітове NOT) – змінює значення кожного біта на протилежне: з 0 на 1 та с 1 на 0
- ^ (побітове XOR) – порівнює бінарні числа даних чисел. Якщо один або другий біт дорівнює одиниці, в результаті буде 1, у іншому випадку 0
- << (побітовий зсув вліво) – зліва від оператора знаходиться ціле число, яке буде зсунуто, а справа значення на скільки біт потрібно зсунути. Наприклад, 20 << 3 робимо зсув числа 20 на 3 біти вліво. В результаті отримаємо 160. При зсуві «пусті» біти заповнюються нулями
- >> (побітовий зсув вправо) – принцип роботи аналогічний лівому зсуву, тільки різниця в тому, що біти зсуваються вправо. Також важливою відмінністю правого зсуву від лівого є те, чим заповнюються «пусті» біти. В даному випадку, якщо ми працюємо з від'ємним числом (у бінарному представленні крайній лівий біт дорівнює 1), усі пусті місця заповнюються одиницями. Якщо число додатне (крайній лівий біт дорівнює 0), у такому випадку пусті місця заповнюються нулями.
- >>> (побітовий зсув вправо с заповненням нулями) – аналогічний побітовому зсуву вправо, але цей оператор завжди заповнює пусті біти нулями. Тобто результат буде завжди додатнім.

Отже, використовуючи ці оператори разом з неявним перетворенням даних можна закодувати числа. Наприклад:

- *~20* → -21 (цей оператор працює за принципом -(число + 1))
- *~[]* → -1 (так трапляється тому що пустий масив представляє собою елемент з числовим значенням 0, отже ~0 = -1)
- *-~[]* → 1
- *~true* → -2 (Так як true представляє собою елемент з числовим значенням 1, тому ~1 = -2)

- *~false* → -1 (Так як false представляє собою елемент з числовим значенням 0, тому ~0 = -1)
- *-~[]*(“”+-~[]+-~-~[]+-~-~true)* → 123

7. Літери та рядки без строкових даних

Іноді потрібно отримати букву або якийсь текст без його явного написання. Зробити це дозволяє одна особливість JavaScript. У цій мові існують різні внутрішньо системні повідомлення, які можна перетворити в текст, а потім цей текст обробити. Наприклад, потрібно отримати слово «code». Цей рядок міститься в іменах таких методів, як *charCodeAt()*, *fromCharCode()* та інших. Отримати текст можна наступним чином:

```
1:   let x = (alert+’’).split(“ive ”)[1].substr(0,4);
```

У цьому випадку змінна x буде містити текст «code». Якщо виконати код «*alert + ‘’*» то отримаємо «function alert() {[native code]}». У JavaScript у кожного об’єкту, функції та іншого є опис. Щоб отримати до нього доступ, потрібно явно змінити тип даного об’єкту чи функції на рядковий, додав, наприклад, пустий рядок (+’’)

8. Заміна назв замінних

В обфускованому коді слід використовувати наступні символи та їх комбінації:

- «o», «O», «0»
- «i», «I», «1», «l»
- «_» (та варіанти «__», «___» …)
- «$» (та варіанти «$$», «$$$» …)

Завдяки таким символам код стає важким для читання, особливо після процесу мініфікації.

Існують також такі обфускатори, які перетворюють код за допомогою всього лише шести символів «[]()!+». Наприклад звичайний код типу *console.log(1);* перетворюється на 4262 символи. Такі обфускатори працюють на основі неявного приведення типів. Хоча такий код буде важко деобфускувати,

проте він займає багато місця, відповідно розмір скрипту буде набагато більший[6].

Окрім попереднього обфускатора є ще більш дивні. Цей обфускатор змінює вихідний код смайликами. Наприклад, код виду *console.log('Hi!');* буде мати такий вигляд (рис 1.24)

```
°ω°ﾉ= /｀ｍ´）ﾉ ~┻━┻   //*´∇｀*/ ['_']; o=(°ｰ°)  =_=3; c=(°Θ°) =(°ｰ°)-(°ｰ°); (°Д°) =(°Θ°)= (o^_^o)/
(o^_^o);(°Д°)={°Θ°: '_' ,°ω°ﾉ : ((°ω°ﾉ==3) +'_') [°Θ°] ,°ｰ°ﾉ :(°ω°ﾉ+ '_')[o^_^o -(°Θ°)] ,°Д°ﾉ:((°ｰ°
==3) +'_')[°ｰ°] }; (°Д°) [°Θ°] =((°ω°ﾉ==3) +'_') [c^_^o];(°Д°) ['c'] = ((°Д°)+'_') [ (°ｰ°)+(°ｰ°)-(°Θ°)
];(°Д°) ['o'] = ((°Д°)+'_') [°Θ°];(°o°)=(°Д°) ['c']+(°Д°) ['o']+(°ω°ﾉ +'_')[°Θ°]+ ((°ω°ﾉ==3) +'_') [°
ｰ°] + ((°Д°) +'_') [(°ｰ°)+(°ｰ°)]+ ((°ｰ°==3) +'_') [°Θ°]+((°ｰ°==3) +'_') [(°ｰ°) - (°Θ°)]+(°Д°) ['c']+((°
Д°)+'_') [(°ｰ°)+(°ｰ°)]+ (°Д°) ['o']+((°ｰ°==3) +'_') [°Θ°];(°Д°) ['_'] =(o^_^o) [°o°] [°o°];(°ε°)=((°ｰ°
==3) +'_') [°Θ°]+ (°Д°) .°Д°ﾉ+((°Д°)+'_') [(°ｰ°) + (°ｰ°)]+((°ｰ°==3) +'_') [o^_^o -°Θ°]+((°ｰ°==3) +'_') [°
Θ°]+ (°ω°ﾉ +'_') [°Θ°]; (°ｰ°)+=(°Θ°); (°Д°)[°ε°]='¥¥'; (°Д°).°Θ°ﾉ=(°Д°+ °ｰ°)[o^_^o -(°Θ°)];(o°ｰ°o)=(°
ω°ﾉ +'_')[c^_^o];(°Д°) [°o°]='¥"';(°Д°) ['_'] ( (°Д°) ['_'] (°ε°+(°Д°)[°o°]+ (°Д°)[°ε°]+(°Θ°)+ (°ｰ°)+
(o^_^o)+ (°Д°)[°ε°]+(°Θ°)+ ((°ｰ°) + (°Θ°))+ ((°ｰ°) + (o^_^o))+ (°Д°)[°ε°]+(°Θ°)+ ((°ｰ°) + (°Θ°))+
((o^_^o) +(o^_^o))+ (°Д°)[°ε°]+(°Θ°)+ ((o^_^o) +(o^_^o))+ (o^_^o)+ (°Д°)[°ε°]+(°Θ°)+ ((°ｰ°) + (°Θ°))+
((°ｰ°) + (o^_^o))+ (°Д°)[°ε°]+(°Θ°)+ ((°ｰ°) + (°Θ°))+ (°ｰ°)+ (°Д°)[°ε°]+(°Θ°)+ (°ｰ°)+ ((°ｰ°) + (°Θ°))+
(°Д°)[°ε°]+((°ｰ°) + (°Θ°))+ ((o^_^o) +(o^_^o))+ (°Д°)[°ε°]+(°Θ°)+ ((°ｰ°) + (°Θ°))+ (°ｰ°)+ (°Д°)[°ε°]+
(°Θ°)+ ((°ｰ°) + (°Θ°))+ ((°ｰ°) + (o^_^o))+ (°Д°)[°ε°]+(°Θ°)+ (°ｰ°)+ ((°ｰ°) + (o^_^o))+ (°Д°)[°ε°]+((°
ｰ°) + (°Θ°))+ (c^_^o)+ (°Д°)[°ε°]+(°ｰ°)+ ((°ｰ°) + (o^_^o))+ (°Д°)[°ε°]+(°Θ°)+ (°Θ°)+ (c^_^o)+ (°Д°)[°
ε°]+(°Θ°)+ ((°ｰ°) + (°Θ°))+ (°Θ°)+ (°Д°)[°ε°]+(°ｰ°)+ (°Θ°)+ (°Д°)[°ε°]+(°ｰ°)+ ((°ｰ°) + (o^_^o))+ (°
Д°)[°ε°]+((°ｰ°) + (°Θ°))+ (°Θ°)+ (°Д°)[°o°]) (°Θ°)) ('_');
```

Рисунок 1.24 - Обфускація смайликами

Отже, при комбінації даних методів можна бути впевненим, що звичайний, або навіть середній користувач не зможе прочитати та проаналізувати обфускований код. Але необхідно пам'ятати, що будь-який обфускований код можна деобфускувати, питання лише у бажанні та часі.

1.4 Постановка задачі

Аналіз існуючих обфускаторів та виявлені недоліки обґрунтовують рішення розробити власний веб-додаток для здійснення ефективної обфускації програмного коду, написаного на JavaScript. Провести аналіз щодо об'єму вихідного коду та обфускованого та щодо часу виконання скрипту до обфускації та після.

Схема роботи веб-додатку наступна:

1. Користувач вставляє JavaScript код, який він хоче обфускувати, у відповідне віконце.
2. Натискає на кнопку «Obfuscate» для здійснення процесу обфускації

3. Обфускований код відображається у другому віконці. Код можна скопіювати до буферу обміну, натиснувши на відповідну кнопку біля віконця

Процес обфускації буде базуватися на перетворенні вихідного коду у формат base64 з додаванням «солі». Процес деобфускації для коректного виконання коду буде базуватися на особливостях мови JavaScript та роботи браузера, а саме:

- неявне приведення типів даних
- використання побітових операторів для кодування чисел
- заміна тексту шістнадцятирічними кодами
- заміна назв змінних
- використання стандартних JavaScript методів для роботи з рядками.

Також необхідно створити наступні сторінки:

- головна – безпосередньо там де можна буде отримати обфускований код
- інформаційна – для перегляду інформації щодо використання ресурсу

2 ВИБІР МЕТОДУ РІШЕННЯ

2.1 Основні положення

Сучасна веб-розробка базується на трьох основних напрямках:

- клієнтська частина (Front-end)
- серверна частина (Back-end)
- бази даних (Databases)

Front-end – напрям у веб-розробці, який працює у браузері, з яким відповідно взаємодіє користувач. Це динамічні інтерфейси, меню, події по діям користувача, обмін даними з серверною частиною, одним словом клієнтська частина це та частина, яку бачить користувач.

В свою чергу Front-end базується на трьох основних складових:

1. HTML
2. CSS
3. JavaScript

HTML

HTML – це мова гіпертекстової розмітки. Вона відповідає за структуру та зміст сторінки. Вона складається з тегів, в свою чергу тег складається з імені всередині кутових дужках. Наприклад: *<div>, <span>, <a>*. Також теги можуть мати атрибути (але для більшості тегів це не обов'язково). Теги у HTML не чутливі до регістру, проте прийнято писати теги маленькими літерами [7].

HTML забезпечує основу для будь-якої сторінки. Розширенням даного файлу є *.html, водо дає зрозуміти браузеру, що всередині файлу знаходиться код веб-сторінки. Браузери розбирають його структуру, визначають взаємо розміщення елементів та візуалізують їх. Разом з CSS, з HTML стало працювати простіше, тому що не потрібно використовувати теги, які відповідають за візуальне оформлення вмісту тегу (наприклад тег *<small>* зменшував розмір відображуваного тексту, проте у сучасній веб-розробці краще використовувати CSS-правило *font-size* для зміни розміру тексту)

CSS

CSS – це каскадна таблиця стилів, яка використовується для стилізації розмітки HTML. Наприклад: змінити колір тексту, вирівняти блок по центру, заокруглити кути ображення, зробити відступи більше або менше і тд. Розробник також може робити так, щоб елементі зникали або переміщувались, тобто робити просту анімацію, також можна зробити, щоб при наведенні на елемент він плавно змінював колір, або змінював свою форму [8].

Стилі можна писати всередині HTML – інлайново або підключити через зовнішній файл. Для цього існує спеціальний тег у HTML <*link*> у якого існують два атрибути: *«href»* - для вказання посилання на CSS файл та *«rel»* - для вказання, що це саме таблиця стилів. Хоча використання інлайнових стилів мають вищій пріоритет, проте частіше підключають зовнішній файл з розширенням *.css. Так Розмітка розділяється від стилізації, відповідно зменшується вірогідність помилки, зовнішній файл кешується та не завантажується повторно.

JavaScript

Сучасний front-end не обходиться без мови програмування JavaScript. Це інтерпретована мова сценаріїв. Це не складна мова програмування, вода дозволяє сторінці включати сценарії, які будуть реагувати на якійсь події користувача, наприклад, клік мишкою, відправлення форми, прокручування сторінки та ін. Такі сценарії з відносно невеликим зусиллями можуть створити складну поведінку веб-сторінок.

За допомогою JavaScript можна створювати такі речі, як:

- Складна анімація елементів сторінок
- Відстежування подій та подальша їх обробка
- Запит інформації з серверу та відображення їх без перезавантаження всієї сторінки
- Робота с cookie
- Створення таблиць, слайдерів, табів
- Обробка та валідація форм

JavaScript має ряд переваг:

- Незамінність для веб-розробки. Усі популярні браузери підтримують цю мову та вона завжди ввімкнена за замовчуванням
- Швидкість роботи. JavaScript дозволяє частково обробляти сторінки на комп'ютерах користувача без запитів до серверу. Це економить час та інтернет-трафік.
- Потужна інфраструктура. За той час поки існує ця мова було створено безліч готових рішень у відкритому доступі, це робить мову більш гнучкою та простішою.
- Простота. Просту задачу можна вирішити за декілька хвилин. Для більш складних задач вже існують готові рішення, якими можна скористатися.
- Зручність інтерфейсів. Заповнення форм, вибір дії, активація кнопок, реагування на наведення миші и тд. Все це підвищує рівень юзабіліті.
- Постійна підтримка та регулярні оновлення

Але також JavaScript має і ряд недоліків, а саме:

- Неможливість читати з файлу. Головна причина цього недоліку це безпека.
- Нестрога типізація. У мові має місце різна інтерпретація даних. Немає можливості попереднього виявлення помилки.
- Доступність для злочинців. У вільну скриптову мову легше всього впровадити шкідливий код, який може нашкодити користувачу

Для зручності програмування на JavaScript можна використовувати бібліотеку jQuery. Це зручна, швидка та багатофункціональна бібліотека. Вона надає багато функції, за допомогою яких можна виконати різні завдання. Її використання значно полегшує виконання тих чи інших задач, також значно зменшує об'єм коду. API-інтерфейси можуть виконувати такі функції, як обробка подій, маніпулювання елементами HTML, додавання анімацій на сторінку. Також бібліотека дозволяє взаємодіяти з сервером без перезавантаження сторінки [9].

Back-end – це серверна частина веб-додатку, яка не помітна для користувача. Сюди відноситься: авторизація, зберігання та обробка даних, email розсилка и тд. Якщо у front-end тільки одна мова – JavaScript. То у back-end мов більше, а саме:

1. PHP
2. Asp net
3. Ruby
4. Python
5. Java
6. Node js

PHP

PHP – одна с поширених мов програмування серверної частини з відкритим вихідним кодом. Мова призначена для динамічної генерації HTML-сторінки. PHP має повну інтеграцію з HTML тому може бути впроваджений безпосередньо в сам HTML-файл. Код PHP починається у трикутних дужках зі знаками питання:

```
1:   <?php
2:   код php …
3:   ?>
```

PHP має одну особливість – це інтегрованість майже з усіма сучасними інтернет-технологіями. PHP підтримує багато сучасних веб-протоколів: SNMP, POP, XML, FTP та ін. Окрім цього PHP дозволяє зручно працювати з базами даних [10].

2.2 Вибір мов програмування

Враховуючі зазначені вище задачі, було обрано такі мови програмування:

Клієнтська частина:

1. HTML (для розмітки)
2. CSS (для надання сторінці візуальної привабливості)
3. JavaScript (а саме бібліотека jQuery – для динамічної роботи веб-додатку)

Серверна частина:

1. PHP (для проведення безпосередньо обфускації вхідного коду)

2.3 Концепція методу обфускації

Обфускація буде проводитись базуючись на особливостях неявного приведення типів даних, використанню побітових операторів, переведення рядків у шістнадцятирічний формат та з використанням стандарту кодування base64.

Процес обфускації коду буде проводитись на back-end на мові програмування PHP. Користувач буде вводити JavaScript код у відповідне віконце, далі, після натиснення на кнопку «Obfuscate» код буде відправлятися на back-end, де буде переведений у формат base64. Далі до отриманого рядка буде додаватися випадково згенерована «сіль» на початку рядка, всередині та вкінці. Кожного разу «сіль» буде мати різну довжину (від 1 до 20 символів) та буде включати різні символи, за генерацію «солі» буде відповідати окрема функція. Також на сервері будуть генеруватися назви змінних, які будуть використані для вирізання «солі» на front-end. Окрім цього буде створений масив чисел від одного до двадцяти, ці числа будуть записані, використовуючи побітові оператори та особливості неявного приведення типів даних. Ці числа будуть означати скільки символів «солі» було додано до вихідного рядка і знадобляться, щоб правильно відрізати «сіль», щоб закодований код правильно декодувався та правильно відпрацював.

Метод кодування base64 відноситься до групи binary-to-text encoding схем, що перетворюють набір байтів у рядковий ASCII формат. Алгоритм кодування у формат base64 переводить набір бітів у індекси завчасно відомого рядка розміром 64 символи. За стандартом RFC 4648 для кодування використовується наступний рядок:

```
1:   ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=
```

Можна помітити, що у рядку 65 символів, а не 64. Це тому, що останній символ «=» потрібен для обробки особливих випадків.

Перетворення байтів у символи ASCII проходить ітеративно, групами по 3 байти (24 біти). Якщо у обраній групі менше ніж 3 байти, тоді відсутні байти конвертуються у символ «=». Далі з трьох восьмибітних груп виділяються чотири шестибітні. Потім бітові послідовності переводяться у десяткову систему числення і отримуються індекси. За отриманими індексами обираються символи із кодуючого рядка та записується результат.

- Base64 корисний там, де потрібно відправляти невеликі бінарні дані разом з текстовими, наприклад для скорочення http запитів за ресурсами
- За допомогою Base64 можна генерувати hash. Така техніка використовується у webpack при генерації імен CSS класів або імен файлів
- Base64 збільшує розмір файлів. Наприклад при використанні алгоритма до png зображенню, яке важить 2.0 mb в результаті отримаємо 2.6 mb (в середньому файл збільшується на 25%)
- Base64 погано підходить для кодування текстового контенту типу SVG, XML та ін. При кодування такого контенту ми збільшуємо час на парсинг та погіршуємо ефективність стиснення.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ

3.1 Проектування макету

Зазвичай розробка веб-додатків проводиться у декілька етапів. Перший і найважливіший етап це створення макету. Макет – це ескіз, на якому відображена майбутня сторінка. Від того, наскільки якісно створений макет, залежить загальне враження на сторінці. При роботі над макетом у графічному редакторі у дизайнера немає обмежень. Якщо обраний шрифт не підходить, його можна замінити в один клік, або якщо кнопка виглядає непривабливою, то її можна з легкістю замінити. Тому задача у дизайнера не така проста, він повинен зробити такий макет, щоб верстальнику було простіше відтворити макет. Створення макету має ряд переваг:

- Визначення візуальних недоліків на перших стадіях роботи над проектом
- Можливість наочно показати клієнту майбутній дизайн сторінки
- Спрощення роботи верстальника та програміста
- Розуміння того, як буде виглядати готова сторінка
- Зведення до мінімуму непорозумінь між дизайнером та замовником

Існують багато інструментів для створення макетів. Найпопулярніший це Photoshop та онлайн-сервіс Figma. На мій погляд Figma буде кращим сервісом, ніж Photoshop, тому що цей ресурс має такі плюси: figma працює як на Windows, Mac так і на телефонах, це безкоштовний продукт, уся робота ведеться на сервері, тому не потрібно турбуватися за передачу макету між дизайнером та верстальником, figma дає можливість працювати над макетом декількох людям одночасно.

Всього буде 2 сторінки: головна та інформаційна. Інформаційна сторінка буде включати інформацію щодо правильного використання ресурсу, а на головній можна буде безпосередньо обфускувати код. Макет головної сторінки (рис 3.1) та макет інформаційної сторінки (рис 3.2)

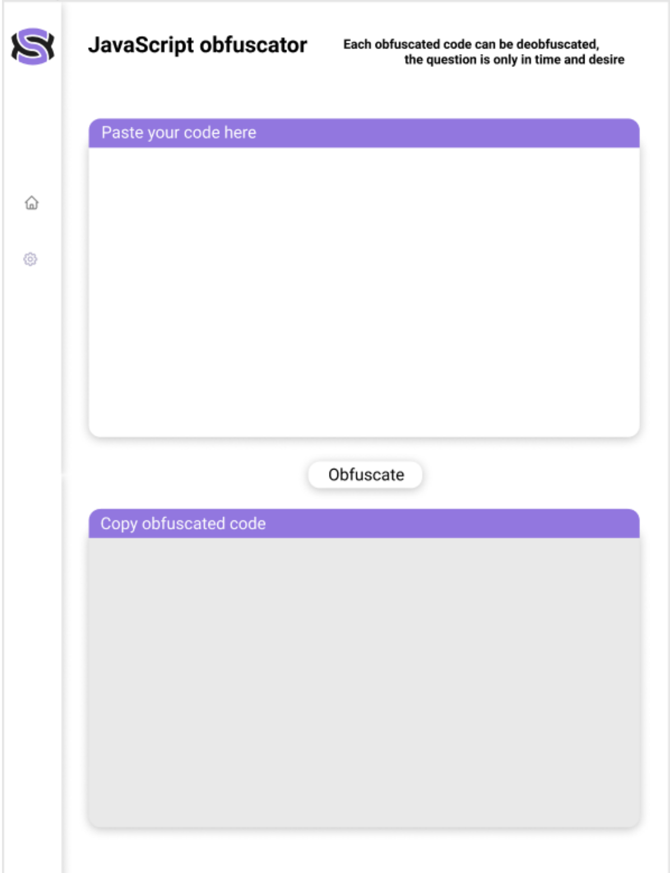

Рисунок 3.1 - Макет головної сторінки

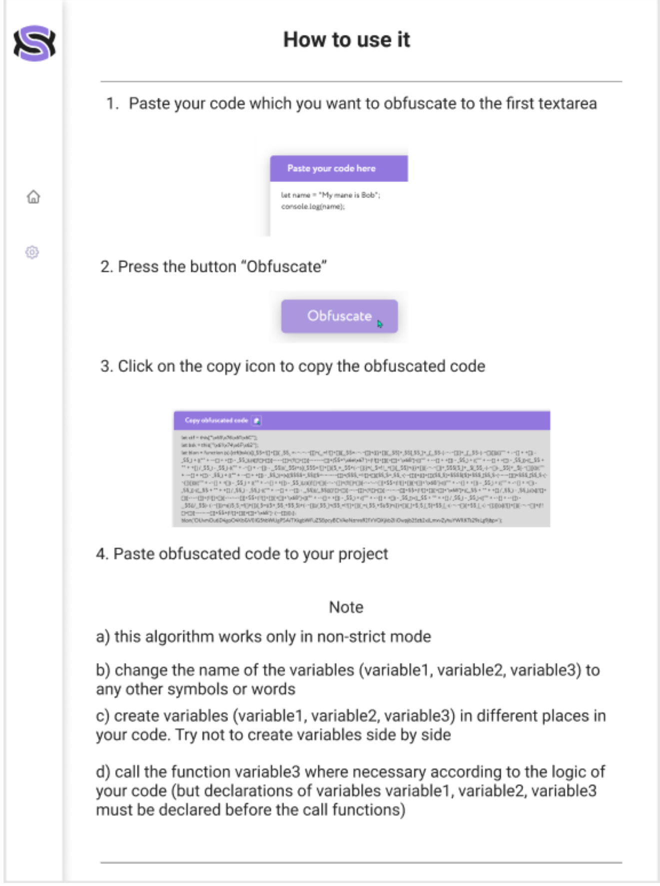

Рисунок 3.2 - Макет інформаційної сторінки

Ієрархія папок головної сторінки буде виглядати наступним чином (рис. 3.3): папка від назвою «php» всередині якої буде зберігатися php файл, у якому буде відбуватися процес обфускації, папка «script», всередині якої буде файл скрипту головної сторінки, папка «style», де будуть зберігатися таблиці стилів, також тут є головний html файл, фавікон та зображення логотипу.

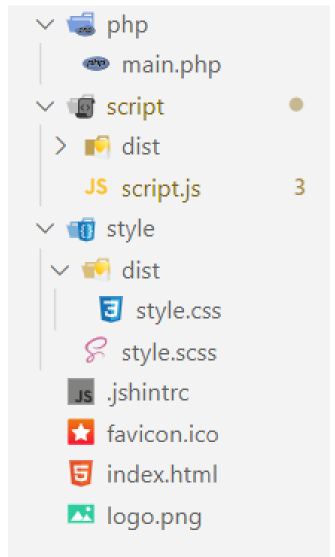

Рисунок 3.3 - Ієрархія папок головної сторінки

Ієрархія папок інформаційної сторінки буде виглядати наступним чином (рис 3.4): папка «img» містить у собі зображення, «script» - скрипти сторінки, «style» - таблиці стилів, головний файл html, логотип та фавікон

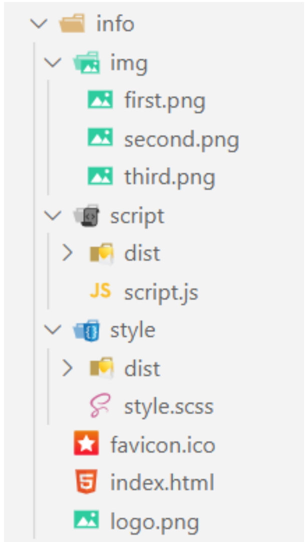

Рисунок 3.4 - Ієрархія папок інформаційної сторінки

Спершу зверстаємо сторінки на основі створеного макету (додаток А, Б)

3.2 Серверна частина

Тепер напишемо серверну частину у файлі *php/main.php* (додаток В). Створимо змінну *$code* у якій буде зберігатися JavaScript код який необхідно обфускувати. Далі цей код потрібно перевести у формат base64. Хоча у PHP існує вбудований метод переводу рядка у формат base64 *base64_encode(),* але я вирішим написати власну функцію. Тому створимо цю функцію, називатись вона буде *base64Encode()*(рис 3.5)

```
 1:  function base64Encode($input){
 2:      $base64_chars =
     "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/="
     ;
 3:      $output = "";
 4:
 5:      $x = 0;
 6:      while($x < strlen($input)){
 7:          $chr1 = ord($input[$x++]);
 8:          $chr2 = ord($input[$x++]);
 9:          $chr3 = ord($input[$x++]);
10:
11:          $enc1 = $chr1 >> 2;
12:          $enc2 = (($chr1 & 3) << 4) | ($chr2 >> 4);
13:          $enc3 = (($chr2 & 15) << 2) | ($chr3 >> 6);
14:          $enc4 = $chr3 & 63;
15:
16:          if ($chr2 == 0) {
17:              $enc3 = $enc4 = 64;
18:          } else if ($chr3 == 0) {
19:              $enc4 = 64;
20:          }
21:
22:          $output .=
     $base64_chars[$enc1].$base64_chars[$enc2].$base64_chars[$enc3].$bas
     e64_chars[$enc4];
23:      }
24:      return $output;
25:  }
```

Рисунок 3.5 - Функція переведення рядка у base64

$base64_chars – це змінна, у якій зберігається набір символів, які використовуються для переведення рядка у формат base64. За допомогою циклу *while* проходимо по усьому вхідному рядку та записуємо у змінні *$chr1, $chr2, $chr3* відповідно перші три числа від 0 до 255 перших трьох символів з вихідного рядка, отриманих за допомогою функції *ord().* Далі нам потрібно з цих чисел отримати індекси, тому змінні *$enc1, $enc2, $enc3, $enc4* будуть містити індекси, які ми отримуємо за допомогою використанню побітових операторів, по яким ми

зможемо перевести рядок у формат base64. Рядок 11: зміщуємо на 2 біти вправо та отримуємо 6 біт. Рядок 12: Беремо 2 останніх біти *$chr1* та складаємо їх з чотирма з *$chr2*. Рядок 13: беремо 4 останніх біти *$chr2* та складаємо їх з двома з *$chr3*. Рядок 14: беремо 6 останніх біт з *$chr3*. Так як перетворення у base64 проводиться групами по 3 байти (3 символи), то далі потрібно перевірити у кінці рядка, чи вистачає нам байтів. Якщо ні, то додаємо 65 символ (=) до рядка, один, якщо одного не вистачає і два, якщо двох. Далі формуємо закодований рядок на основі отриманих індексів та повертаємо результат. Закодований рядок буде зберігатися у змінній *$code_ecrypted*.

Наступна функція яка буде створена це *generate_string()*. Вона буде генерувати випадкову комбінацію символів заданої довжини. Це потрібно для генерування випадкової «солі» для закодованого рядка у base64 та для генерації назв змінних, які вже нам знадобляться для деобфускації на клієнтській частині (рис 3.6)

```
 1:  $permitted_chars_salt =
     '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ';
     // символы, которые могут быть в строке для соли
 2:  $permitted_chars_vars = '_$'; // символы, которые могут быть в
     строке для переменных
 3:  function generate_string($input, $strength) {
 4:      $input_length = strlen($input);
 5:      $random_string = '';
 6:      for($i = 0; $i < $strength; $i++) {
 7:          $random_character = $input[mt_rand(0, $input_length - 1)];
 8:          $random_string .= $random_character;
 9:      }
10:
11:      return $random_string;
12:  }
```

Рисунок 3.6 -2 функція випадкової генерації рядків

Змінна *$permited_chars_salt* містить у собі набір символів, які будуть використані для генерації солі, як можна помітити символи співпадають з символами, які використовуються у форматі base64, окрім 3 символів (+ / =) це

потрібно для того аби користувач не зміг зрозуміти, що до рядка була додана «сіль» певної довжини.

Змінна *$permitted_chars_vars* містить у собі набір символів, які будуть використані для генерації назв змінних. Так як ці змінні будуть використані у JavaScript мові, то допустимі лише такі символи: $ та _. Завдяки особливості JavaScript можна використовувати у якості назв змінних ці символи. Також вони можуть значно ускладнити аналіз коду, тому що легко сплутати такі змінні як __ та ___ або $_$ та $__$, особливо коли таких символів багато.

Далі створимо масив, згенерованих функцією *generate_string(),* змінних (рис 3.7)

```
1:  $array_of_vars = array();
2:  $array_of_vars_unique = array_unique($array_of_vars);
3:  while(count($array_of_vars_unique) != 16){
4:      $array_of_vars = array();
5:      for($i = 0; $i < 16; $i++){
6:          $array_of_vars[$i] = generate_string($permitted_chars_vars,
    mt_rand(1, 4));
7:      }
8:      $array_of_vars_unique = array_unique($array_of_vars);
9:  }
```

Рисунок 3.7 -3 Створення масиву змінних

Оскільки згенеровані рядки будуть використовуватись у якості змінних, тому потрібно зробити так, щоб у згенерованих рядках не було дублікатів. Для цього можна використати функцію *array_unique().* Всього змінних буде 16 штук, хоча насправді реальних змінних буде всього 4. Решта змінних будуть дублювати 4 реальних змінні та дані, які вони містять. Це потрібно, щоб зробити обфускований код більш різноманітним та заплутаним. Також було прийнято рішення зробити довжину змінних від одного символу до чотирьох. На мій погляд якщо, робити довжину змінної у п’ять символів і більше, то вона буде більш помітною, тому зможе дещо спростити ручну деобфускацію коду, що не є прийнятним. Згенеровані назви змінних будуть зберігатися у змінній *$array_of_vars_unique.*

Тепер потрібно до рядка, у якому вихідний код закодований у формат base64, додати «сіль» напочатку, всередині та вкінці рядка (рис. 3.8).

```
1:  do{
2:      $start_paste = 0;
3:      $middle_paste = 0;
4:      $end_paste = 0;
5:
6:      while($start_paste % 4 == 0 || $middle_paste % 4 == 0 ||
    $end_paste % 4 == 0){
7:          $start_paste = random_int(1, 20);
8:          $middle_paste = random_int(1, 20);
9:          $end_paste = random_int(1, 20);
10:     }
11:
12:      if(strlen($code) % 3 == 0){
13:         $code_changed = generate_string($permitted_chars_salt,
    $start_paste).substr($code_encrypted, 0,
    strlen($code_encrypted)/2).generate_string($permitted_chars_salt,
    $middle_paste).substr($code_encrypted, strlen($code_encrypted)/2,
    (strlen($code_encrypted)/2)).generate_string($permitted_chars_salt,
    $end_paste);
14:         $count_of_equal = 0;
15:     }

16:
17:     if(strlen($code) % 3 == 2){
18:       … // якщо довжина вихідного коду при діленні на 3 буде мати
    остачу 2, но «сіль» вкінці вставляємо перед 1 знаком =
19:         $count_of_equal = 1;
20:     }
21:
22:     if(strlen($code) % 3 == 1){
23:       … // якщо довжина вихідного коду при діленні на 3 буде мати
    остачу 1, но «сіль» вкінці вставляємо перед 2 знаками =
24:          $count_of_equal = 2;
25:     }
26:
27: }while(strlen($code_changed) % 4 != 0);
```

Рисунок 3.8 -4 Цикл додавання "солі"

Оскільки рядок у форматі base64 завжди має парну кількість символів, то вставити «сіль» рівно посередині буде не важко. Також потрібно передбачити ситуацію, коли недостатні байти конвертуються у знак «=», в такому випадку «сіль» вкінці повинна вставлятися перед знаком «=», а не після нього. Для цього були створені перевірки на рядках 12, 17, 22. Також у змінну *$count_of_equal* запишемо число знаків «=» це буде потрібно для правильної деобфускації на клієнті. Ще одним нюансом є довжина «солі», оптимальною довжиною я вважаю буде від одного до двадцяти символів, але при цьому довжина «солі» не повинна бути кратна чотирьом. Довжина солі буде зберігатися у змінних *$start_paste* –

«сіль» напочатку, *$middle_paste* - «сіль» всередині, *$end_paste* - «сіль» вкінці. Тому що, як вже було сказано (див 2.3) base64 бере 3 символи та конвертує у 4. Відповідно якщо довжина «солі» буде кратна чотирьом, то свою функцію вона не виконає, тому що якщо такий рядок написати у будь-який онлайн декодер base64, то вихідний код можна буде з легкістю отримати, тому аби не допустити цього, «сіль» не повинна бути довжиною кратною чотирьом. Також довжина закодованого рядка у base64 завжди кратна чотирьом, тому проводимо генерацію «солі» до тих пір поки довжина результуючого рядка буде кратна чотирьом, а також до тих пір поки довжина «солі» напочатку або всередині або вкінці буде кратна чотирьом.

Тепер створимо масив закодованих чисел від одного до двадцяти (рис 3.9).

```
1:  $encodeNumb = array(
2:      1 => '(-~[])',
3:      2 => '('.$array_of_vars_unique[1].')',
4:      3 => '('.$array_of_vars_unique[0].')',
5:      4 => '('.$array_of_vars_unique[1].' + '.$array_of_vars_unique[0].' - -~[])',
6:      5 => '('.$array_of_vars_unique[0].' + '.$array_of_vars_unique[0].' - -~[])',
7:      6 => '('.$array_of_vars_unique[0].' + '.$array_of_vars_unique[1].' + -~[])',
8:      7 => '(("" + -~[] + +[]) - '.$array_of_vars_unique[0].')',
9:      8 => '('.$array_of_vars_unique[1].' << -~-~[])',
10:     9 => '(("" + -~[] + -~[]) - '.$array_of_vars_unique[1].')',
11:     10 => '(('.$array_of_vars_unique[1].' + "" + +[] /
    '.$array_of_vars_unique[0].') / -~-~[])',
12:     11 => '(('.$array_of_vars_unique[0].' + '.$array_of_vars_unique[0].') *
    '.$array_of_vars_unique[1].' - -~[])',
13:     12 => '(+(-~[] + "" + '.$array_of_vars_unique[1].'))',
14:     13 => '(("" + +!![] + ('.$array_of_vars_unique[1].' + -~-~-~[])) -
    '.$array_of_vars_unique[1].')',
15:     14 => '((("" + -~[] + +[]) - '.$array_of_vars_unique[0].') + (("" + -~[] +
    +[]) - '.$array_of_vars_unique[0].'))',
16:     15 => '(+("" + +!![] + ('.$array_of_vars_unique[1].' + -~-~-~[])))',
17:     16 => '(("" + -~[] + +[]) - '.$array_of_vars_unique[0].') + ((-~[] + "" +
    +[]) - '.$array_of_vars_unique[1].' + -~[])',
18:     17 => '(('.$array_of_vars_unique[1].' + "" + +[] /
    '.$array_of_vars_unique[0].') - '.$array_of_vars_unique[0].')',
19:     18 => '(('.$array_of_vars_unique[1].' << '.$array_of_vars_unique[0].') +
    '.$array_of_vars_unique[1].')',
20:     19 => '((("" + -~[] + +[]) - '.$array_of_vars_unique[0].') +
    (('.$array_of_vars_unique[1].' + "" + +[] / '.$array_of_vars_unique[0].') / -~-
    ~[]) + '.$array_of_vars_unique[1].')',
21:     20 => '((('.$array_of_vars_unique[1].' + "" + +[] /
    '.$array_of_vars_unique[0].') / -~-~[]) + (('.$array_of_vars_unique[1].' + "" +
    +[] / '.$array_of_vars_unique[0].') / -~-~[]))',
22: );
```

Рисунок 3.9 - Масив закодованих чисел

Це ті числа, що означають скільки символів «солі» було додано рядка. Кодувати числа будемо за допомогою побітових операторів та використовуючи особливості неявного приведення типів даних (див. 1.1.2). Задля більшого заплутування використаємо назви змінних, які згенерували раніше. Оскільки назви змінних ніяк не впливають на їх вміст, то зробимо так, що перша змінна у масиві містить число 3, друга – число 2 (також закодовані за допомогою побітових операторів). Чому саме такі числа розберемо нижче.

3.3 Клієнтська частина

Створимо слухач подій на кнопці «Obfuscate». При кліку на яку будемо відправляти на сервер код, який необхідно обфускувати та виводити обфускований код у друге віконце. Після того як запит на сервер успішно був відправлений, сформуємо ту інформацію, яка буде виводитись у друге віконце (додаток Г). Вона має таку структуру (рис 3.10)

```
1:   let var1 = window.eval;
2:   let var2 = window.atob;
3:   let var3 = function(s) {var1(var2(s))};
4:   var3(//рядок у форматі base64);
```

Рисунок 3.10 - Загальна структура виводу

Створюємо змінну *var1* яка зберігає функцію *eval()*. Вона приймає у якості аргументу рядок, та виконує його як JavaScript код. Хоча використання даної функції не є безпечним, проте у нашому випадку використання її є доречним. Наступна змінна *var2* зберігає функцію *atob()*. Вона приймає у якості аргументу рядок, закодований у форматі base64 та розкодовує його. Змінна *var3* це функція, яка у якості аргументу буде приймати рядок, отриманий з серверу. Всередині функції ми беремо закодований у формат base64 рядок з «сільою», відрізаємо «сіль», декодуємо отриманий рядок функцією *var2(atob()),* отриманий код виконуємо через функцію *var1(eval()).*

Відрізати «сіль» будемо використовуючи функції *substr(),* також знадобиться властивість рядків *length.* Для того, щоб отримати закодований код

без «солі» потрібно використати таку конструкцію (рис 3.11)

```
1:   str.substr(start, (str.length - start - middle - end) / 2) +
     str.substr(start + (str.substr(start, (str.length - start - middle
     - end) / 2)).length + middle, ((str.length - start - middle - end)
     / 2) - 2) + str.substr(str.length - 2);
```

Рисунок 3.11 - Рядок без "солі"

Де *str* – рядок, у якому потрібно видалити «сіль», start – кількість вставлених символів «солі» напочатку рядка, middle – всередині рядка, end – вкінці. Отже, якщо розбити цей код на 3 частини, то: *str.substr(start, (str.length - start - middle - end) / 2)* – це вихідний рядок від кінця «солі» напочатку до початку «солі» всередині, *str.substr(start + (str.substr(start, (str.length - start - middle - end) / 2)).length + middle, ((str.length - start - middle - end) / 2) - 2)* - це вихідний рядок від кінця «солі» всередині до початку «солі» вкінці, *str.substr(str.length – (0 або 1 або 2))* – це вихідний рядок від кінці «солі» вкінці до кінця рядка, тобто дана частина коду вирізає знаки «=», якщо вони є. Отже, якщо скласти ці частини, то отримаємо вихідний рядок закодований у формат base64 без «солі».

Тепер задля ускладнення розуміння як саме працює процес видалення «солі» потрібно дещо змінити код. Наприклад, змінну *var1* можна замінити наступним чином *let var1 = this["\\x65\\x76\\x61\\x6C"];*, змінну *var2 - let var2 = this["\\x61\\x74\\x6F\\x62"];* Оскільки у голові всього стоїть *this,* то використаємо субституцію та заміну тексту шістнадцятирічними кодами (див 1.1.2), таким чином не кожен користувач зможе здогадатися, що тут закодовано.

Частина коду, де проходить процес видалення «солі», буде закодована за допомогою неявного перетворення типів даних та побітових операторів (див 1.1.2). Спершу потрібно закодувати слово *substr*. Для цього потрібно неявно отримати літери s, u, b, s, t, r. Їх ми можемо отримати з таких слів як: *true*, *false* та *object.* Якщо в результаті деякої операції ми отримаємо, наприклад, *true,* а потім додамо пустий рядок, то отримаємо вже не булевий тип *true*, а рядковий "*true*". Це означає, що тепер ми зможемо отримати будь-яку літеру цього слова, просто вказавши індекс. Наприклад: *"true"[0] == "t"*.

Отримати рядок “*false*” можна наступним чином: *![] + []*. Оскільки пустий масив є *true,* то використовуючи заперечення отримаємо булеве *false,* якщо додати пустий масив, то булеве *false* перетвориться на рядковий “*false*”. Тобто якщо до чогось додати пустий масив, то отримаємо рядковий тип даних того, що додавали. Для зручності збережемо таке перетворення у змінній *xx: xx = ![] + []*. Оскільки маємо рядкове представлення “*false*”, можемо отримати першу літеру з рядка *substr* – літеру *s.* Тобто *xx[3] == “s”,* але щоб не використовувати явне написання цифр, можна їх написати, використовуючи побітові операції, цифра 3 буде еквівалентна такому запису: *x = -~-~-~[],* також для зручності збережемо цю цифру у змінній *x.* Варто відмітити, що оголошення змінної без ключового слова *var*, *let* або *const* можна використовувати, проте це застаріла можливість, і працює вона тільки у нестрогому режимі. Отже, якщо зібрати ці два записи, то отримаємо неявно літеру «s»: *(xx = ![] + [])[x = -~-~-~[]] => "false"[3] => «s».* Аналогічним чином отримаємо решту необхідних літер для “*substr*” (рис 3.12)

```
 1:   // (xx = ![] + []) => "false"
 2:   // [x = -~-~-~[]] => [3]
 3:   //    (xx = ![] + [])[x = -~-~-~[]] => "false"[3] => s
 4:
 5:   // (xxx = !![] + []) => "true"
 6:   // [xxxx = -~-~[]] => [2]
 7:   //    (xxx = !![] + [])[xxxx = -~-~[]] => "true"[2] => u
 8:
 9:   // ({} + []) => "[object Object]"
10:   // [xxxx] => [2]
11:   //    ({} + [])[xxxx] => "[object Object]"[2] => b
12:
13:   // xx => "false"
14:   // [x] => [3]
15:   //    xx[x] => "false"[3] => s
16:
17:   // xxx => "true"
18:   // [xxxx - (-~-~[])] => [0]
19:   //    xxx[xxxx - (-~-~[])] => "true"[0] => t
20:
21:   // xxx => "true"
22:   // [xxxx - (-~[])] => [1]
23:   //    xxx[xxxx - (-~[])] => "true"[1] => r
```

Рисунок 3.12 - Отримання неявно літер "substr"

Отже, якщо скласти рядки 3, 7, 11, 15, 19, 23 отримаємо “*substr*”. Також у нас є такі змінні:

- *xx = “false”*
- *x = 3*
- *xxx = “true”*
- *xxxx = 2*

Згідно з конструкції (рис 3.11) нам потрібно використати метод *substr()* чотири рази, тому, аби не повторювались частини коду, будемо використовувати ці змінні. Також будемо створювати додаткові змінні, які будуть дублювати значення, які зберігаються у змінних *x, xx, xxx, xxxx.* Це для того, аби зробити код біль різноманітим за заплутаним. Взагалі назви цих змінних умовні, реальні назви змінних будуть генеруватися на сервері, які будуть складатися з символів $ та _ (див. 3.2). Закодуємо решту назв методів *substr()* (рис 3.13)

```
1:   [(yy = ![] + [])[y = -~-~-~[]] + (yyy = !![] + [])[yyyy = -~-~[]]
    + ({} + [])[yyyy] + yy[y] + yyy[yyyy - (-~-~[])] + yyy[yyyy - (-
    ~[])]]
2:
3:   [(qq = ![] + [])[q = yyyy + (-~[])] + (qqq = !!yyy + [])[yyyy] +
    ({} + [])[-~-~[]] + qq[q] + qqq[y-(-~[]) - yyyy] + qqq[-~[]]]
4:
5:   [(ww = yy)[w = -~-~-~[]] + (www = !![] + [])[wwww = -~-~[]] + ({}
    + [])[wwww] + ww[w] + www[wwww - (-~-~[])] + www[wwww - (-~[])]]
6:
7:   [(bb = qq)[b = -~-~-~[]] + (bbb = !![] + [])[bbbb = -~-~[]] + ({}
    + [])[bbbb] + bb[b] + bbb[bbbb - (-~-~[])] + bbb[bbbb - (-~[])]]
```

Рисунок 3.13 - 4 різні варіанти кодування назви методу substr()

Окрім методу substr() необхідно також замаскувати властивість *length* (рис 3.14).

```
1:  (![]+[])[-~-~[]]+(!![]+[])[-~-~-~[]]+($$_$='\x6e\x67')+(!![]+[])
    [+[]]+'\x68'
```

Рисунок 3.14 - Маскування length

Принцип такий самий як і під час маскування *substr()*:

- *(![]+[])[-~-~[]]* → “false”[2] → “l”

- *(!![]+[])[-~-~-~[]]* → “true”[3] → “e”
- *($$_$='\x6e\x67')* → “ng” (*$$_$* - змінна, яка також генерується на сервері, щоб далі по коду не виділялися шістнадцяткові коди)
- *(!![]+[])[+[]]* → “true”[0] → “t”
- *'\x68'* → “h”

Тепер необхідно правильно скласти усі замасковані методи та властивості, щоб отримати (рис 3.11), але в замаскованому вигляді. Отже, функція, яка видаляє «сіль» у замаскованому вигляді має такий вид (рис 3.15):

```
1:   let variable3 = function (s) {variable1(variable2(s[(${data.vars[2]}=![]+[])[
     ${data.vars[0]} =-~-~-~[]]+(${data.vars[3]}=!![]+[])[${data.vars[1]}=-~-
     ~[]]+({}+[])[${data.vars[1]}]+${data.vars[2]}[${data.vars[0]}]+${data.vars[3]}[${
     data.vars[1]}-(-~-~[])]+${data.vars[3]}[${data.vars[1]}-(-
     ~[])]](${start},(s[(![]+[])[-~-~[]]+(!![]+[])[-~-~-
     ~[]]+(${data.vars[15]}='\\x6e\\x67')+(!![]+[])[+[]]+'\\x68']-${start}-${middle}-
     ${end})/${data.vars[1]})+s[(${data.vars[4]}=![]+[])[${data.vars[5]}=${data.vars[1
     ]}+(-~[])]+(${data.vars[6]}=!!${data.vars[3]}+[])[${data.vars[1]}]+({}+[])[-~-
     ~[]]+${data.vars[4]}[${data.vars[5]}]+${data.vars[6]}[${data.vars[0]}-(-~[])-
     ${data.vars[1]}]+${data.vars[6]}[-
     ~[]]](${start}+(s[(${data.vars[7]}=${data.vars[2]})[${data.vars[8]}=-~-~-
     ~[]]+(${data.vars[9]}=!![]+[])[${data.vars[10]}=${data.vars[0]}-(-
     ~[])]+({}+[])[${data.vars[10]}]+${data.vars[7]}[${data.vars[8]}]+${data.vars[9]}[
     ${data.vars[10]}-(-~-~[])]+${data.vars[9]}[${data.vars[10]}-(-
     ~[])]](${start},(s[(![]+[])[-~-~[]]+(!![]+[])[-~-~-
     ~[]]+${data.vars[15]}+(!![]+[])[+[]]+'\\x68']-${start}-${middle}-
     ${end})/${data.vars[1]}))[(![]+[])[-~-~[]]+(!![]+[])[-~-~-
     ~[]]+${data.vars[15]}+(!![]+[])[+[]]+'\\x68']+${middle},((s[(![]+[])[-~-
     ~[]]+(!![]+[])[-~-~-~[]]+${data.vars[15]}+(!![]+[])[+[]]+'\\x68']-${start}-
     ${middle}-
     ${end})/${data.vars[1]})${countOfEqual})+s[(${data.vars[11]}=![]+[])[${data.vars[
     12]}=((${data.vars[8]}+${data.vars[0]}+${data.vars[10]})+(-
     ~[]))/${data.vars[0]}]+(${data.vars[13]}=!![]+[])[${data.vars[14]}=(${data.vars[0
     ]}+${data.vars[8]})/${data.vars[8]}]+({}+[])[${data.vars[14]}]+${data.vars[11]}[$
     {data.vars[12]}]+${data.vars[13]}[${data.vars[14]}-(-~-
     ~[])]+${data.vars[13]}[${data.vars[14]}-(-~[])]](s[(![]+[])[-~-~[]]+(!![]+[])[-~-
     ~-~[]]+${data.vars[15]}+(!![]+[])[+[]]+'\\x68']${countOfEqual})));};
```

Рисунок 3.15 - Замаскована функція, яка видаляє "сіль"

Змінні *data.vars[0-15]* це назви змінних, які були згенеровані на сервері, *contOfEqual* – кількість знаків «=», також у закодованому вигляді, start/middle/end – числа, що позначають кількість символів «солі», які були додані напочатку, всередині та вкінці (також у закодованому вигляді)

3.4 Тестування та аналіз результатів роботи

Тестування є одним з найважливіших етапів розробки будь-якого продукту, адже на етапі тестування виявляються баги та недоліки, які можуть бути недопустимі при релізі.

Спочатку перевіримо чи працює код без обфускації. Для цього напишемо невеликий скрипт (рис 3.16), який буде обчислювати вираз (3.1) та виводити результат обчислень на сторінку. Змінна x буде змінюватись від 0 до 15.

$$\frac{(\cos x - x * \sin x) * \sqrt[3]{x^3 + 3x + 1} - \frac{x * \cos x}{3 * \sqrt[3]{(x^3 + 3x + 1)^2}} * (3x^2 + 3)}{\sqrt[3]{(x^3 + 3x + 1)^2}}$$

(3.1)

```
 1:  function calc(x) {
 2:      let ans = ((Math.cos(x) - x * Math.sin(x)) *
     (Math.pow(Math.pow(x, 3) + 3 * x + 1, 1 / 3)) - ((x * Math.cos(x))
     / (3 * Math.pow((Math.pow(x, 3) + 3 * x + 1), 2 / 3))) * (3 *
     Math.pow(x, 2) + 3)) / (Math.pow(Math.pow(x, 3) + 3 * x + 1), 2 /
     3);
 3:      let elem = document.createElement('p');
 4:      elem.innerHTML = ans;
 5:      let page = document.getElementById('page');
 6:      page.appendChild(elem);
 7:  }
 8:
 9:  for (let i = 0; i < 15; i++) {
10:      calc(i);
11:  }
```

Рисунок 3.16 - Скрипт обчислення прикладу

В результаті скрипт відпрацював правильно (рис 3.17)

1.5

-1.326829163500156

-7.240714056078865

-3.052261410295701

18.830539322514127

37.60554009349039

16.01013881464105

-48.96747849579249

-96.55592346683342

-51.01806046543244

82.17885047634323

183.03192709362733

116.9378036333423

-106.94000594774428

-292.7219400886376

Рисунок 3.17 - Робота скрипту без обфускації

Тепер цей код обфускуємо та порівняємо результати. Перейдемо до головної сторінки та вставимо вихідний код у перше віконце (рис 3.18)

Paste your code here

```
function calc(x) {
  let ans = ((Math.cos(x) - x * Math.sin(x)) * (Math.pow(Math.pow(x, 3) + 3 * x + 1, 1 / 3)) - ((x * Math.cos(x)) / (3 * Math.pow((Math.pow(x, 3) + 3 * x + 1), 2 / 3))) * (3 * Math.pow(x, 2) + 3)) / (Math.pow(Math.pow(x, 3) + 3 * x + 1), 2 / 3);
  let elem = document.createElement('p');
  elem.innerHTML = ans;
  let page = document.getElementById('page');
  page.appendChild(elem);
}

for (let i = 0; i < 15; i++) {
  calc(i);
}
```

Рисунок 3.18 - Вставка вихідного коду у віконце для обфускації

Натиснемо на кнопку «Obfuscate» та отримаємо обфускований код (рис 3.19)

Copy obfuscated code

```
let variable1 = this["\x65\x76\x61\x6C"];
let variable2 = this["\x61\x74\x6F\x62"];
let variable3 = function (s) {variable1(variable2(s[(_$_=![]+[])[_$$ =-~-~-~[]]+(_$$$=!![]+[])[_$_=-~-~[]]+({}+[])[_$_]+_$_[_$$]+_$$$[_$_-(-~-~[])]+_$$$[_$_-(-~[])]]((_$$ + _$$ - -~[]),(s[(![]+[])[-~-~[]]+(!![]+[])[-~-~-~[]]+(_$='\x6e\x67')+(!![]+[])[+[]]+'\x68']-(_$$ + _$$ - -~[])-(("" + +!![] + (_$_ + -~-~-~[])) - _$_)-(_$_))/_$_)+s[($_$=![]+[])[_=_$_+(-~[])]+($=!!_$$$+[])[_$_]+({}+[])[-~-~[]]+$_$[_]+$[_$$-(-~[])-_$_]+$[-~[]]]((_$$ + _$$ - -~[])+(s[(_$_$=_$_)[$_=-~-~-~[]]+($$=!![]+[])[_=_$$-(-~[])]+({}+[])[_]+_$_$[$_]+$$[_-(-~-~[])]+$$[_-(-~[])]]((_$$ + _$$ - -~[]),(s[(![]+[])[-~-~[]]+(!![]+[])[-~-~-~[]]+_$+(!![]+[])[+[]]+'\x68']-(_$$ + _$$ - -~[])-(("" + +!![] + (_$_ + -~-~-~[])) - _$_)-(_$_))/_$_))[(![]+[])[-~-~[]]+(!![]+[])[-~-~-~[]]+_$+(!![]+[])[+[]]+'\x68']+(("" + +!![] + (_$_ + -~-~-~[])) - _$_),((s[(![]+[])[-~-~[]]+(!![]+[])[-~-~-~[]]+_$+(!![]+[])[+[]]+'\x68']-(_$$ + _$$ - -~[])-(("" + +!![] + (_$_ + -~-~-~[])) - _$_)-(_$_))/_$_))+s[($$_=![]+[])[$_$$=(($_+_$$+_)+(-~[]))/_$$]+($_$=!![]+[])[$$$$=(_$$+$_)/$_]+({}+[])[$$$$]+$$_[$_$$]+$_$[$$$$-(-~-~[])]+$_$[$$$$-(-~[])]](s[(![]+[])[-~-~[]]+(!![]+[])[-~-~-~[]]+_$+(!![]+[])[+[]]+'\x68'])));};
variable3('j9qBWZnVuY3Rpb24gY2FsYyh4KSB7CiAgICBsZXQgYW5zID0gKChNYXRoLmNvcyh4KSAtIHggKiBNYXRoLnNpbih4KSkgKiAoTWF0aC5wb3coTWF0aC5wb3coeCwgMykgKyAzICogeCArIDEsIDEgLyAzKSkgLSAoKHggKiBNYXRoLmNvcyh4KSkgLyAoMyAqIE1hdGgucG93KChNYXRoLnBvdyh4LCAzKSArIDMgKiB4ICsgMSksIDIgLyAzKSkpICogKDMgKiBNYXRoLnBvdyh4LCAyKSArIDMpKSAvIChNYXRoLnBvdyhNcQfE9ayGJD0huYXRoLnBvdyh4LCAzKSArIDMgKiB4ICsgMSksIDIgLyAzKTsKICAgIGxldCBlbGVtID0gZG9jdW1lbnQuY3JlYXRlRWxlbWVudCgncCcpOwogICAgZWxlbS5pbm5lckhUTUwgPSBhbnM7CiAgICBsZXQgcGFnZSA9IGRvY3VtZW50LmdldEVsZW1lbnRCeUlkKCdwYWdlJyk7CiAgICBwYWdlLmFwcGVuZENoaWxkKGVsZW0pOwp9Cgpmb3IgKGxldCBpID0gMDsgaSA8IDE1OyBpKyspIHsKICAgIGNhbGMoaSk7Cn0KcS');
```

Рисунок 3.19 -5 Отримання обфускованого коду

Вставимо обфускований код у будь-яку IDE та перевіримо, що скрипт відпрацював так само як і до обфускації (рис. 3.20)

1.5 -96.55592346683342

-1.326829163500156 -51.01806046543244

-7.240714056078865 82.17885047634323

-3.052261410295701 183.03192709362733

18.830539322514127 116.9378036333423

37.60554009349039 -106.94000594774428

16.01013881464105 -292.7219400886376

-48.96747849579249

Рисунок 3.20 - Результат виконання обфускованого коду

Результат роботи обох скриптів однаковий, а це означає, що обфускація пройшла успішно.

В ході аналізу коду до та після обфускації була створена таблиця часу виконання скрипту. Було модифіковано попередній тестовий скрипт (рис 3.18) так, що перед тим як обчислювати приклад на сторінку додавався елемент параграфу, потім результат записувався у цей елемент та елемент видалявся. Це змусить скрипт виконуватись довше.

Таблиця 3.1 Час виконання скрипту до обфускації та після

Час роботи без обфускації	**Час роботи з обфускацією**	**Кількість ітерацій**
0.76196 ms	1.00903 ms	10
1.85595 ms	2.13525 ms	100
13.50195 ms	14.67797 ms	1000
106.62377 ms	111.65014 ms	10000
1121.42529 ms	958.81298 ms	100000
10727.15307 ms	11864.77490 ms	1000000
121851.49975 ms	118243.97485 ms	10000000

Можна помітити, що чим більше проходить ітерацій, тим довше виконується скрипт. Але час виконання до обфускації та після практично не відрізняється, це

означає, що процес деобфускації не забирає зайвих ресурсів, тому що цей процес створений на основі побітових операторів та неявного приведення типів даних, що самі по собі ці операції не потребують багато ресурсів.

Також був проведений аналіз об'єму коду до обфускації та після.

Таблиця 3.2 Порівняння об'єму коду до обфускації та після

Кількість символів до обфускації	**Кількість символів після обфускації**	**На скільки збільшився об'єм коду (%)**
499	2268	354.5%
996	2954	196.5%
1617	3849	138%
2495	5107	104.6%
4628	7731	67%
6515	10705	64.3%
10003	14557	45.5%

З таблиці помітно, що чим більша кількість символів до обфускації тим відсоток символів після обфускації менше, це пов'язано з тим, що об'єм також займають «статичні» символи. Це визначення змінних *variable1* та *variable2* та алгоритм видалення «солі». Основний приріст до об'єму дає перетворення вихідного тексту у формат base64 – 3 символи конвертуються в 4.

ВИСНОВКИ

Метою роботи було створення веб-додатку, який дозволяв би виконувати обфускацію JavaScript коду. Для отримання максимально ефективного результату було виконано наступні задачі: проаналізовані сучасні методи обфускації коду, було обрано засоби реалізації додатку, які найбільш підходять під постановку задачі, виконано верстання веб-сторінки, був розроблений алгоритм кодування та декодування вихідного коду, проведено тестування та аналіз результатів. При розробці алгоритму був врахований час виконання скрипту, тому обфускований код працює не повільніше ніж вихідний.

Процес обфускації базується на перетворенні вихідного коду у формат base64 з додаванням «солі». Процес деобфускації для коректного виконання коду базується на особливостях мови JavaScript та роботи браузера, а саме:

- неявне приведення типів даних
- використання побітових операторів для кодування чисел
- заміна тексту шістнадцятирічними кодами
- заміна назв змінних
- використання стандартних JavaScript методів для роботи з рядками.

Тому спираючись на сукупність вищеперерахованих чинників, можна сказати, що веб-додаток відповідає усім вимогам.

СПИСОК ЛІТЕРАТУРИ

1. Медгаус С.В., Чернышова А.В. Обфускатор программного кода языка JavaScript. Информатика и кибернетика, № 4(6), – Донецк: ДонНТУ, 2016. с. 59 – 66.
2. Проектирование обфускатора для языка JavaScript. ИУСМКМ – 2016: VII Международная научно-техническая конференция, 26 мая 2016: – Донецк: ДонНТУ, 2016. с. 167-173.
3. Варновский Н.П., Захаров В.А., Кузюрин Н.Н., Шокуров А.В. Современное состояние исследований в области обфускации программ: определения стойкости обфускации // Труды Института системного программирования РАН (электронный журнал), том 26, № 3, с. 167-198.
4. Козачок А.В. Комплекс алгоритмов контролируемого разграничения доступа к данным, обеспечивающий защиту от несанкционированного доступа / А.В. Козачок, Л.М. Туан // Системы управления и информационные технологии. – Воронеж, 2015. – № 3(61). – С. 58–61.
5. Обзор существующих обфускаторов и их алгоритмов. Компьютерная и программная инженерия – 2015 год: – Донецк: ДонНТУ, 2015. с. 117-119
6. Никольская К. Ю., Хлестов А. Д. Обфускация и методы защиты программных продуктов. Вестник УрФО «Безопасность в информационной сфере».– 2015.–Том 16.– С.7-10.
7. Brooks, David R. Programming in HTML and PHP. – 2017. c. 1-5.
8. Кит Грант, CSS для профи. – 2019. с. 24-36.
9. Джон Резиг, Беэр Бибо, Иосип Марас Секреты JavaScript ниндзя. Второе издание. – 2017. с 30-33.
10. Лукьянов М. Ю. PHP. Полное руководство и справочник функций. – 2020. с. 21-35.

ДОДАТКИ

ДОДАТОК А

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>JS Obfuscator</title>
    <link rel="stylesheet" href="style/dist/style.css">
    <link      rel="stylesheet"      href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/5.15.3/css/all.min.css"
integrity="sha512iBBXm8fW90+nuLcSKlbmrPcLa0OT92xO1BIsZ+ywDWZCvqsWgccV3gFoRBv0z+8dLJgyAH
IhR35VZc2oM/gI1w=="
        crossorigin="anonymous" />
    <link rel="icon" href="favicon.ico" type="image/x-icon">
</head>
<div class="pageWrapper">
    <div class="menuBar__wrapper">
        <div class="menu__logo">
            <img src="logo.png" alt="logo">
        </div>
        <div class="menu__navbar">
            <div class="nav__home">
                <i class="fas fa-home"></i>
            </div>
            <div class="nav__info">
                <i id='info' class="fas fa-info-circle"></i>
            </div>
        </div>
    </div>
    <div class="mainContent__wrapper">
        <div class="mainContent__header__wrapper">
            <div class="header__name__wrapper">
                <div class="header__name">
                    <span>JavaScript obfuscator</span>
                </div>
            </div>
            <div class="header__rule__wrapper">
                <div class="header__rule">
                    <pre>Each obfuscated code can be deobfuscated,
                              the question is only in time and desire</pre>
                </div>
            </div>
        </div>
        <div class="mainContent__body__wrapper">
            <div class="body__input__wrapper textarea__wrapper">
                <div class="body__input body__area">
                    <div class="body__hat">
                        <span>Paste your code here</span>
                    </div>
```

```
                    <textarea     name="enter"     id="enter"     cols="50"     rows="18"
placeholder='console.log("Paste your code here");'></textarea>
                </div>
            </div>
            <div class="body__submit__wrapper">
                <div class="body_submit">
                    <button id="submit">Obfuscate</button>
                </div>
            </div>
            <div class="body__result__wrapper textarea__wrapper">
                <div class="body__result body__area">
                    <div class="body__hat">
                        <span>Copy obfuscated code</span>
                        <div class="copy__wrapper">
                            <i class="fas fa-copy"></i>
                        </div>
                        <!-- <div class="copied"> -->
                        <span class="copied">Copied!</span>
                        <!-- </div> -->
                    </div>
                    <textarea    name="result"    id="result"     cols="50"     rows="18"
readonly></textarea>
                </div>
            </div>
        </div>
    </div>
</div>
<script        src=https://code.jquery.com/jquery-3.5.1.min.js        integrity="sha256-
9/aliU8dGd2tb6OSsuzixeV4y/faTqgFtohetphbbj0=" crossorigin="anonymous"></script>
<script src="script/script.js"></script>
</body>
</html>
```

ДОДАТОК Б

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>JS Obfuscator | Info</title>
    <link rel="stylesheet" href="style/dist/style.css">
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/5.15.3/css/all.min.css"
        integrity="sha512-
iBBXm8fW90+nuLcSKlbmrPcLa0OT92xO1BIsZ+ywDWZCvqsWgccV3gFoRBv0z+8dLJgyAHIhR35VZc2oM/gI1w=
="
        crossorigin="anonymous" />
    <link rel="icon" href="favicon.ico" type="image/x-icon">
</head>

<div class="pageWrapper">
    <div class="menuBar__wrapper">
        <div class="menu__logo">
            <img src="logo.png" alt="logo">
        </div>
        <div class="menu__navbar">
            <div class="nav__home">
                <i id='home' class="fas fa-home"></i>
            </div>
            <div class="nav__info">
                <i class="fas fa-info-circle"></i>
            </div>
        </div>
    </div>
    <div class="mainContent__wrapper">
        <div class="mainContent__body">
            <div class="mainContent__header">
                <span>How to use?</span>
            </div>
            <div class="mainContent__info">
                <div class="step">
                    <p>1. Paste your code which you want to obfuscate to the first
textarea</p>
                    <div class="imgWrapper">
                        <img src="img/first.png" alt="first_textarea">
                    </div>
                </div>
                <div class="step">
                    <p>2. Press the button "Obfuscate"</p>
                    <div class="imgWrapper">
                        <img id='button' src="img/second.png" alt="button">
                    </div>
                </div>
                <div class="step">
```

```
                    <p>3. Click on the copy icon to copy the obfuscated code</p>
                    <div class="imgWrapper">
                        <img id='result' src="img/third.png" alt="result">
                    </div>
                </div>
                <div class="step">
                    <p>4. Paste obfuscated code to your project</p>
                </div>
                <div class="step">
                    <p>Note:</p>
                </div>
                <div class="step">
                    <p>a) this algorithm works ONLY in non-strict mode </p>
                </div>
                <div class="step">
                    <p>b) change the name of the variables (variable1, variable2,
variable3) to any other symbols or words </p>
                </div>
                <div class="step">
                    <p>c) create variables (variable1, variable2, variable3) in
different places in your code. Try not to create variables side by side</p>
                </div>
                <div class="step">
                    <p>d) call the function variable3 where necessary according to the
logic of your code (but declarations of variables variable1, variable2, variable3 must
be declared before the call functions)</p>
                </div>
            </div>
        </div>
    </div>
</div>
<script src="https://code.jquery.com/jquery-3.5.1.min.js"
    integrity="sha256-9/aliU8dGd2tb6OSsuzixeV4y/faTqgFtohetphbbj0="
crossorigin="anonymous"></script>
<script src="script/script.js"></script>
</body>
</html>
```

ДОДАТОК В

```
<?php
header('Content-Type: application/json');
// код, который пришел с фронта
$code = $_POST["code"];
// кодируем код в base64
// $code_encrypted = base64_encode($code);
$code_encrypted = base64Encode($code);
// функция кодировки кода в base64
function base64Encode($input){
    $base64_chars =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=";
    $output = "";
    $x = 0;
    while($x < strlen($input)){
        $chr1 = ord($input[$x++]);
        $chr2 = ord($input[$x++]);
        $chr3 = ord($input[$x++]);

        $enc1 = $chr1 >> 2;
        $enc2 = (($chr1 & 3) << 4) | ($chr2 >> 4);
        $enc3 = (($chr2 & 15) << 2) | ($chr3 >> 6);
        $enc4 = $chr3 & 63;

        if ($chr2 == 0) {
            $enc3 = $enc4 = 64;
        } else if ($chr3 == 0) {
            $enc4 = 64;
        }

        $output .=
$base64_chars[$enc1].$base64_chars[$enc2].$base64_chars[$enc3].$base64_chars[$enc4];
    }
    return $output;
}
// функция генерирования рандомной строки определенной длинны
$permitted_chars_salt =
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'; // символы, которые
могут быть в строке для соли
$permitted_chars_vars = '_$'; // символы, которые могут быть в строке для переменных
function generate_string($input, $strength) {
    $input_length = strlen($input);
    $random_string = '';
    for($i = 0; $i < $strength; $i++) {
        $random_character = $input[mt_rand(0, $input_length - 1)];
        $random_string .= $random_character;
    }
    return $random_string;
}
$array_of_vars = array();
$array_of_vars_unique = array_unique($array_of_vars);
```

```
while(count($array_of_vars_unique) != 16){
    $array_of_vars = array();
    for($i = 0; $i < 16; $i++){
        $array_of_vars[$i] = generate_string($permitted_chars_vars, mt_rand(1, 4));
    }
    $array_of_vars_unique = array_unique($array_of_vars);
}
// цикл в котором добавляем "соль" к зашифрованной строке. Перегенирируем до тех пор
пока длинна измененного зашифрованого кода в base64 не будет кратна 4
do{
    $start_paste = 0; // количество вставляемых символов в начало зашифрованной строки
    $middle_paste = 0; // количество вставляемых символов в середину зашифрованной
строки
    $end_paste = 0; // количество вставляемых символов в конец зашифрованной строки

    // цикл в котором рандомим длинну вставляемых символов. Рандомим до тех пор пока
количество стоящих рядом символов не будет кратно 4
    while($start_paste % 4 == 0 || $middle_paste % 4 == 0 || $end_paste % 4 == 0){
        $start_paste = random_int(1, 20);
        $middle_paste = random_int(1, 20);
        $end_paste = random_int(1, 20);
    }

    // если длинна изначального кода при делении на 3 будет иметь остаток 2, тогда
$end_paste вставляем перед одним знаком =
    if(strlen($code) % 3 == 2){
        $code_changed = generate_string($permitted_chars_salt,
$start_paste).substr($code_encrypted, 0,
strlen($code_encrypted)/2).generate_string($permitted_chars_salt,
$middle_paste).substr($code_encrypted, strlen($code_encrypted)/2,
(strlen($code_encrypted)/2) - 1).generate_string($permitted_chars_salt,
$end_paste).'=';
        $count_of_equal = 1;
    }
    // если длинна изначального кода при делении на 3 будет иметь остаток 1, тогда
$end_paste вставляем перед двумя знаками ==
    if(strlen($code) % 3 == 1){
        $code_changed = generate_string($permitted_chars_salt,
$start_paste).substr($code_encrypted, 0,
strlen($code_encrypted)/2).generate_string($permitted_chars_salt,
$middle_paste).substr($code_encrypted, strlen($code_encrypted)/2,
(strlen($code_encrypted)/2) - 2).generate_string($permitted_chars_salt,
$end_paste).'==';
        $count_of_equal = 2;
    }
    // если длинна изначального кода при делении на 3 будет иметь остаток 0, тогда
$end_paste вставляем просто в конец строки
    if(strlen($code) % 3 == 0){
        $code_changed = generate_string($permitted_chars_salt,
$start_paste).substr($code_encrypted, 0,
strlen($code_encrypted)/2).generate_string($permitted_chars_salt,
```

```
$middle_paste).substr($code_encrypted, strlen($code_encrypted)/2,
(strlen($code_encrypted)/2)).generate_string($permitted_chars_salt, $end_paste);
        $count_of_equal = 0;
    }
}while(strlen($code_changed) % 4 != 0);
// кодируем кличество вставленных символов соли
$encodeNumb = array(
    1 => '(-~[])',
    2 => '('.$array_of_vars_unique[1].')',
    3 => '('.$array_of_vars_unique[0].')',
    4 => '('.$array_of_vars_unique[1].' + '.$array_of_vars_unique[0].' - -~[])',
    5 => '('.$array_of_vars_unique[0].' + '.$array_of_vars_unique[0].' - -~[])',
    6 => '('.$array_of_vars_unique[0].' + '.$array_of_vars_unique[1].' + -~[])',
    7 => '(("" + -~[] + +[]) - '.$array_of_vars_unique[0].')',
    8 => '('.$array_of_vars_unique[1].' << -~-~[])',
    9 => '(("" + -~[] + -~[]) - '.$array_of_vars_unique[1].')',
    10 => '(('.$array_of_vars_unique[1].' + "" + +[] / '.$array_of_vars_unique[0].') /
-~-~[])',
    11 => '(('.$array_of_vars_unique[0].' + '.$array_of_vars_unique[0].') *
'.$array_of_vars_unique[1].' - -~[])',
    12 => '(+(-~[] + "" + '.$array_of_vars_unique[1].'))',
    13 => '(("" + +!![] + ('.$array_of_vars_unique[1].' + -~-~-~[])) -
'.$array_of_vars_unique[1].')',
    14 => '((("" + -~[] + +[]) - '.$array_of_vars_unique[0].') + (("" + -~[] + +[]) -
'.$array_of_vars_unique[0].'))',
    15 => '(+("" + +!![] + ('.$array_of_vars_unique[1].' + -~-~-~[])))',
    16 => '(("" + -~[] + +[]) - '.$array_of_vars_unique[0].') + ((-~[] + "" + +[]) -
'.$array_of_vars_unique[1].' + -~[])',
    17 => '(('.$array_of_vars_unique[1].' + "" + +[] / '.$array_of_vars_unique[0].') -
'.$array_of_vars_unique[0].')',
    18 => '(('.$array_of_vars_unique[1].' << '.$array_of_vars_unique[0].') +
'.$array_of_vars_unique[1].')',
    19 => '((("" + -~[] + +[]) - '.$array_of_vars_unique[0].') +
(('.$array_of_vars_unique[1].' + "" + +[] / '.$array_of_vars_unique[0].') / -~-~[]) +
'.$array_of_vars_unique[1].')',
    20 => '((('.$array_of_vars_unique[1].' + "" + +[] / '.$array_of_vars_unique[0].') /
-~-~[]) + (('.$array_of_vars_unique[1].' + "" + +[] / '.$array_of_vars_unique[0].') / -
~-~[]))',
);
$answer = array('codeChanged' => $code_changed,
                'start' => $start_paste,
                'middle' => $middle_paste,
                'end' => $end_paste,
                'defaultCode' => $code_encrypted,
                'countOfEqual' => $count_of_equal,
                'vars' => $array_of_vars_unique,
                'numbs' => $encodeNumb,
                'sourceCode' => $code
            );
echo json_encode($answer);
die();
?>
```

ДОДАТОК Г

```
$(window).on('load', () => {
    let submit = $('#submit');
    let info = $('.copied');
    let copy = $('.copy__wrapper');
    copy.on('click', () => {
        $('#result').select();
        document.execCommand("copy");
        window.getSelection().empty();
        info.fadeIn(300);
        setTimeout(()=>{
            info.fadeOut(300);
        }, 1500);
    });

    let infoBtn = $('#info');
    infoBtn.on('click', () => {
        $(location).attr('href', 'info/index.html');
    });

    submit.on('click', (event) => {
        event.preventDefault();
        let inputField = $('#enter').val();
        $.ajax({
            url: 'php/main.php',
            type: 'post',
            data: {
                code: inputField
            },
            // dataType: 'json',
            success: function (data) {

                let start = 0;
                let middle = 0;
                let end = 0;
                let countOfEqual;

                if (data.countOfEqual == 1){
                    countOfEqual = '- (-~[])';
                }
                if (data.countOfEqual == 2){
                    countOfEqual = '- ((-~[]) + (-~[]))';
                }
                if (data.countOfEqual == 0){
                    countOfEqual = '';
                }

                for (const key in data.numbs) {
                    if(key == data.start){
                        start = data.numbs[key];
                    }
```

```
                    if(key == data.middle){
                        middle = data.numbs[key];
                    }
                    if(key == data.end){
                        end = data.numbs[key];
                    }
                }
            let pasteData = `let variable1 = this["\\x65\\x76\\x61\\x6C"];
let variable2 = this["\\x61\\x74\\x6F\\x62"];
let variable3 = function (s) {variable1(variable2(s[(${data.vars[2]}=![]+[])[
${data.vars[0]} =-~-~-~[]]+(${data.vars[3]}=!![]+[])[${data.vars[1]}=-~-
~[]]+({}+[])[${data.vars[1]}]+${data.vars[2]}[${data.vars[0]}]+${data.vars[3]}[${data.v
ars[1]}-(-~-~[])]+${data.vars[3]}[${data.vars[1]}-(-~[])]](${start},(s[(![]+[])[-~-
~[]]+(!![]+[])[-~-~-~[]]+(${data.vars[15]}='\\x6e\\x67')+(!![]+[])[+[]]+'\\x68']-
${start}-${middle}-
${end})/${data.vars[1]})+s[(${data.vars[4]}=![]+[])[${data.vars[5]}=${data.vars[1]}+(-
~[])]+(${data.vars[6]}=!!${data.vars[3]}+[])[${data.vars[1]}]+({}+[])[-~-
~[]]+${data.vars[4]}[${data.vars[5]}]+${data.vars[6]}[${data.vars[0]}-(-~[])-
${data.vars[1]}]+${data.vars[6]}[-
~[]]](${start}+(s[(${data.vars[7]}=${data.vars[2]})[${data.vars[8]}=-~-~-
~[]]+(${data.vars[9]}=!![]+[])[${data.vars[10]}=${data.vars[0]}-(-
~[])]+({}+[])[${data.vars[10]}]+${data.vars[7]}[${data.vars[8]}]+${data.vars[9]}[${data
.vars[10]}-(-~-~[])]+${data.vars[9]}[${data.vars[10]}-(-~[])]](${start},(s[(![]+[])[-~-
~[]]+(!![]+[])[-~-~-~[]]+${data.vars[15]}+(!![]+[])[+[]]+'\\x68']-${start}-${middle}-
${end})/${data.vars[1]}))[(![]+[])[-~-~[]]+(!![]+[])[-~-~-
~[]]+${data.vars[15]}+(!![]+[])[+[]]+'\\x68']+${middle},((s[(![]+[])[-~-
~[]]+(!![]+[])[-~-~-~[]]+${data.vars[15]}+(!![]+[])[+[]]+'\\x68']-${start}-${middle}-
${end})/${data.vars[1]})${countOfEqual})+s[(${data.vars[11]}=![]+[])[${data.vars[12]}=(
(${data.vars[8]}+${data.vars[0]}+${data.vars[10]})+(-
~[]))/${data.vars[0]}]+(${data.vars[13]}=!![]+[])[${data.vars[14]}=(${data.vars[0]}+${d
ata.vars[8]})/${data.vars[8]}]+({}+[])[${data.vars[14]}]+${data.vars[11]}[${data.vars[1
2]}]+${data.vars[13]}[${data.vars[14]}-(-~-~[])]+${data.vars[13]}[${data.vars[14]}-(-
~[])]](s[(![]+[])[-~-~[]]+(!![]+[])[-~-~-
~[]]+${data.vars[15]}+(!![]+[])[+[]]+'\\x68']${countOfEqual})));};
variable3('${data.codeChanged}');`;
                $('#result').val(pasteData);
                console.log(data);
            }
        });
    });
});
```