# ДОСЛІДЖЕННЯ З ЗАСТОСУВАННЯМ ДИНАМІЧНОГО МЕТОДУ

Динамічний метод відновлення алгоритмів захисту заснований на використанні програмних зневаджувачних засобів (зневаджувачів). Зневаджувач - це програма, яка завантажує в пам'ять іншу програму і надає користувачеві можливість спостерігати за ходом виконання цієї програми. Існує досить багато зневаджувачів. Серед найбільш поширених можна назвати WinDbg, Oily Debugger, Syser. Більшість сучасних компіляторів мають вбудовані зневаджувачі, багато з них (наприклад, зневаджувач, вбудований в Microsoft Visual Studio) придатні не тільки для налагодження щойно написаних програм, а й для аналізу стороннього програмного забезпечення. Вибір зневаджувача залежить від мети його застосування. З перерахованих зневаджувачів найбільш зручні у використанні Oily Debugger і вбудований зневаджувач Microsoft Visual Studio, що мають зручний багатовіконний інтерфейс. Однак спостерігається наступна закономірнийність: чим більше зневаджувач зручний, тим менше він ефективний. під ефективністю тут розуміється здатність «проникати» в системні області пам'яті і протистояти захисту від зневаджувача. Механізм роботи будь-якого зневаджувача заснований на використанні спеціальних засобів, апаратно реалізованих в процесорах. Для процесорів сімейства Intel х86 до цих засобів відносяться: прапор трасування, точка зупинки і налагоджувальні регістри.

Прапор трасування – це біт 8 регістра прапорів. Коли цей біт дорівнює одиниці, процесор після виконання кожної машинної команди викликає переривання 1. Прапор трасування використовується зневаджувачами для організації покрокового виконання програми.

Машинний код команди виклику переривання 3, на відміну від інших переривань, займає всього один байт. Тому ця команда може бути вставлена в будь-яке місце програми. Зневаджувач перехоплює переривання 3, і після встановлення точки зупинки (заміни першого байта машинної команди на байт CCh - код команди int 3) передає управління програмі. Перед виконанням команди, на якій стоїть крапка зупинки, управління передається в зневаджувач. Перед повторним запуском програми зневаджувач повинен відновити виправлену команду.

Починаючи з моделі 80386 процесори сімейства Intel мають вісім налагоджувальних регістрів. Ці регістри дозволяють розмістити в пам'яті ЕОМ чотири апаратні точки зупину. На відміну від звичайних точок зупину, апаратні володіють більш широкими можливостями. Програма може бути зупинена не тільки при виконанні певної команди, але і при зверненні до певної ділянки пам'яті. При зупинці програми апаратною точкою зупинки викликається переривання 1.

Зневаджувачі зазвичай реалізують такі можливості:

* покроковий прохід програми (трасування);
* покроковий прохід програми, при якому виклик функції обробляється як одна команда;
* покроковий прохід програми з «провалом» в переривання;
* установка / зняття точки зупину із заданою команди, багато зневаджувачів допускають установку умовних точок зупину, які спрацьовують лише в разі виконання деякої додаткової умови. Умовні точки зупину реалізуються як програмне розширення звичайних (програмних або апаратних) точок зупину - точка зупинки спрацьовує незалежно від виконання заданої умови, а потім зневаджувач приймає рішення, чи слід припинити програму або продовжити її виконання;
* установка / видалення апаратної точки зупину;
* установка / видалення точки зупину з порту / портів;
* виконання ділянки програми до найближчої точки зупину;
* виконання ділянки програми до команди, що підсвічується курсором, або до найближчої точки зупину;
* перезавантаження програми;
* перегляд коду / пам'яті / регістрів / прапорів / стека;
* пошук рядка в пам'яті;
* пошук машинної команди в пам'яті;
* зміна вмісту регістрів / прапорів / пам'яті / коду.

У момент зупинки зневаджувачі дозволяють переглядати вміст оперативної пам'яті як у шістнадцятирічному вигляді, так і у вигляді дизасемблювати коду.

Можливості зневаджувачів дають простий і зрозумілий в ідейному плані підхід до відновлення алгоритмів захисту, який отримав назву «динамічний метод». Для реалізації даного підходу необхідно запустити програму зневаджувача, відкрити програму що досліджується, потім встановити кілька точок зупину і, рухаючись по програмі, з тих чи інших обставин (вид екрану, відкриття файлів і т.п.) виділити і проаналізувати необхідні фрагменти програми.

У порівнянні зі статичним методом аналізу динамічний метод дозволяє проводити пошук алгоритмів захисту більш ціленаправлено. Аналітик може запускати і переглядати саме ті ділянки програмного забезпечення, в яких, як йому відомо, використовуються алгоритми захисту що його цікавлять. Динамічний метод дозволяє в моменти зупинки програми спостерігати стан процесора і оперативної пам'яті, що полегшує розуміння дій програми.

Аналіз програми динамічним методом розбивається на три основних етапи:

1. пошук підходів до необхідних функцій програми (знайти шлях, по якому йти);
2. пошук необхідних функцій програми (пройти цей самий шлях);
3. аналіз необхідних функцій програми.

При аналізі машинного коду завданням першого етапу є виявлення в масиві машинного коду програми деяких «зачіпок», що дозволяють досить швидко підібратися до необхідних функцій.

## 4.1 Опис експерименту та застосованої програмної розробки

В якості досліджуваної програми будемо розглядувати просту програму своєї розробки, яка шифрує та дешифрує рядок введений з клавіатури за допомогою криптографічної бібліотеки з використанням алгоритму AES та режиму ЕСВ. Динамічний аналіз будемо проводити за допомогою зневаджувача для платформи .NET Framework – dnSpy. dnSpy дизасемблює програми які працюють на .NET Framework до коду IL. В ході експерименту будуть шукатися функції шифрування та дешифрування та проводитися аналіз їх алгоритмів.

## 4.2 Результати дослідження з застосуванням динамічного методу

При завантаженні програми до dnSpy бачимо дизасембльований код IL. За замовчуванням dnSpy переходе до функції Main. У головному вікні можна встановлювати точки зупинки на будь-яку з команд IL.

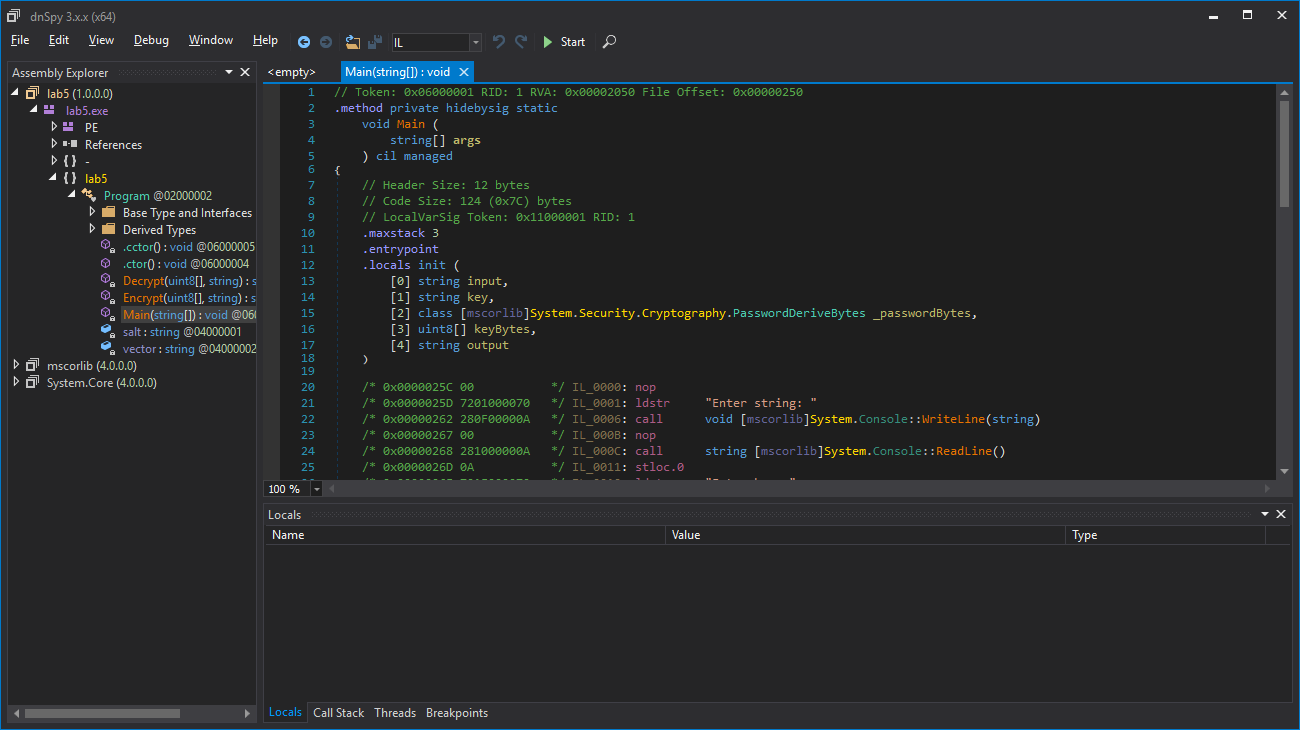


Рисунок 4.1 – Головне вікно програми dnSpy

### 

### 4.2.1 Застосування методу маячків

При застосовуванні методу маячків знайдемо функцію, виклик якої зрозумілий навіть без контексту. Бачимо, що на рядку 43 виконується виклик функції з іменем Encrypt. Встановимо точку зупинки на цю функцію (рис 4.2).

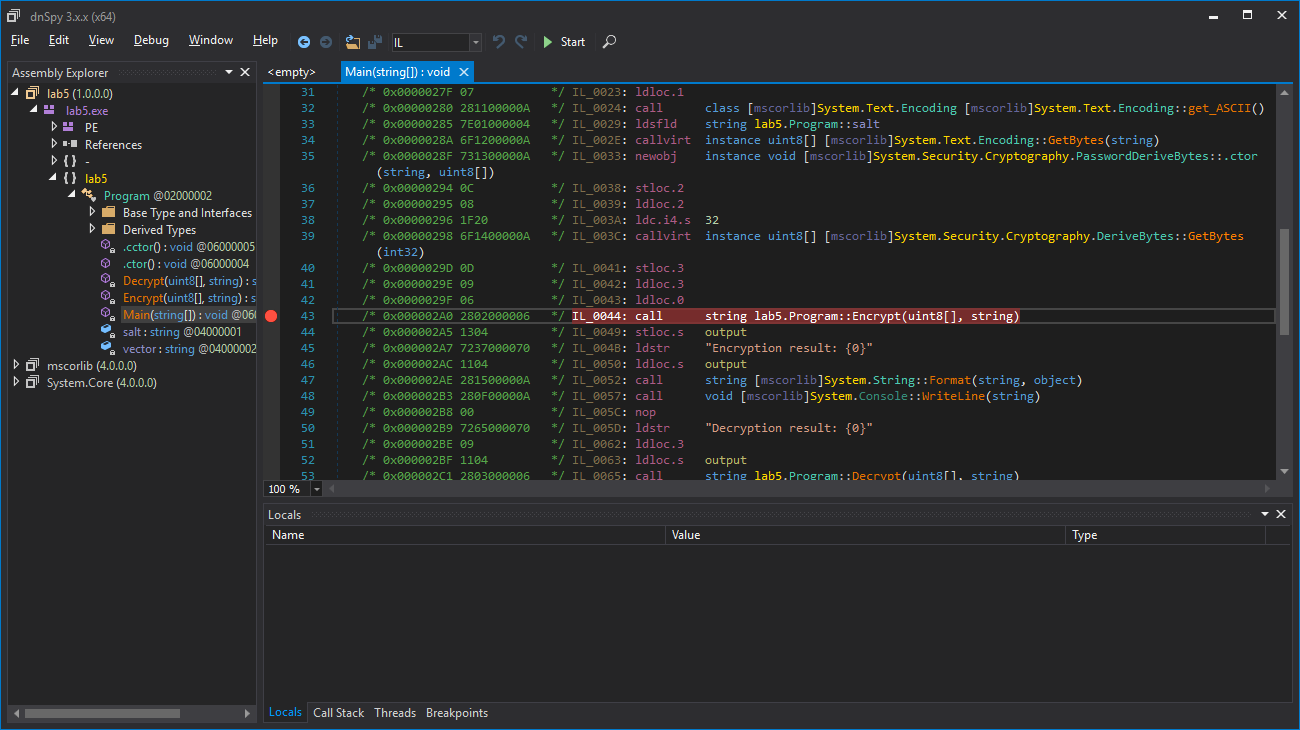


Рисунок 4.2 – Встановлення точки зупинки

Після встановлення точки зупинки, запустимо зневаджувач за допомогою натискання клавіші «F5». Введемо рядок для шифрування «Test string» та ключ «123456». У вікні «Locals» бачимо, введені рядки зберігаються у змінних V\_0 та V\_1, далі вони передаються до функції Encrypt. За допомогою клавіші «F11» передемо до функції Encrpyt (рис 4.3). Бачимо, що ключ було перетворено до массиву байт для подальшого використання у криптографічній бібліотеці.

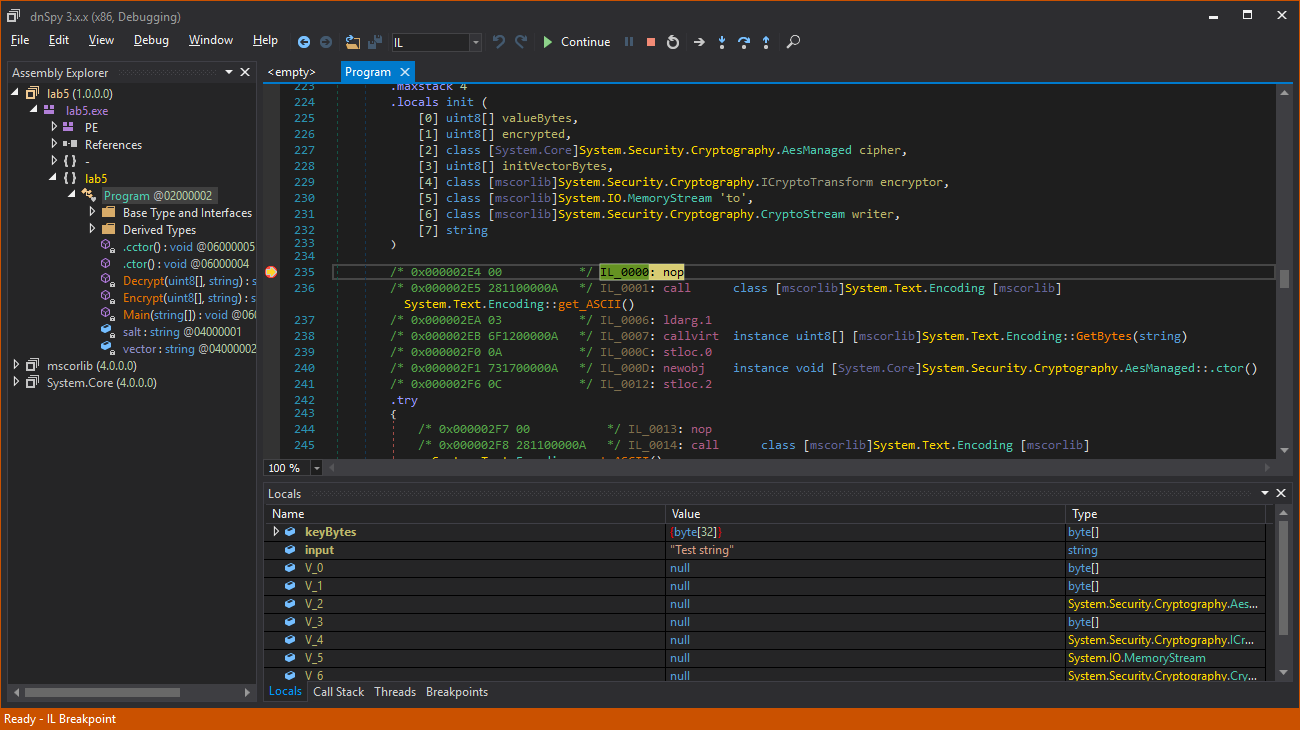


Рисунок 4.3 – Функція Encrypt

Аналізуючи функцію Encrypt бачимо, що для шифрування використовується об’єкт класу System.Security.Cryptography.AesManaged з встановленням режиму ЕСВ. Шифрація виконується за допомогою об’екту CryptoStream та функції FlushFinalBlock. Встановимо точку зупину на його використанні та переглянемо вікно Locals. Так як перед викликом функції використовується команда ldloc.0 (передача змінної V\_0 на стек для завантаження у функцію), можна визначити, що результат знаходиться у змінній V\_0 (рис 4.4).

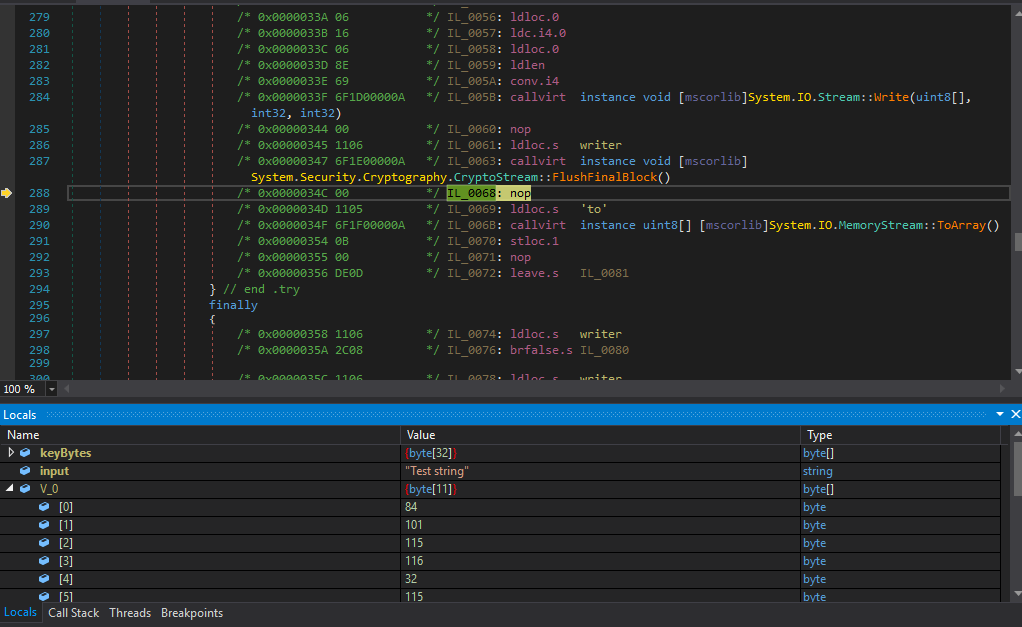


Рисунок 4.3 – Результат функції FlushFinalBlock

### 4.2.2 Дослідження методом Step-Trace першого етапу

При застосуванні методу Step-Trace першого етапу, знайдему точку у програмі, де дані які нас цікавлять модифікуються. У даному випадку нас цікавить функція дешифрування вхідного рядка. Почнемо проходити по програмі за допомогою команди «Step» зневаджувача. Після вводу рядка для шифрування «Test string» та ключу «123456», звернемо увагу на вікно «Locals» зневаджувача (рис 4.4).

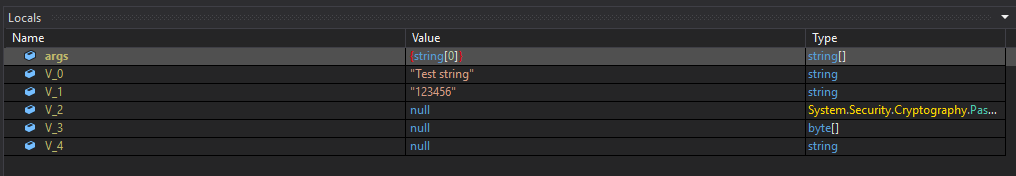


Рисунок 4.4 – Вікно «Locals»

Продовжимо проход програми командою «Step». Після проходження IL команди «call string lab5.Program::Encrypt(uint8[], string)», у вікні «Locals» бачимо, що значення змінної V\_4 змінилося на «aJU/z59ikZD+e3HJ5ewEIQ==», що є зашифрованим рядком. Так, можемо визначити, що функцією шифрування є «lab5.Program::Encrypt», а ії результат заноситься до змінної V\_4. Продовжимо проход програми командою «Step». Можемо побачити, що після визову команди «call string lab5.Program::Decrypt(uint8[], string)» іде визов виводу рядку на екран, та на екрані відображається дешифрований рядок. Так як результат функцій у коді IL заноситься до стеку, можна припустити, що функція «lab5.Program::Decrypt» дешифрує рядок, та ії результат використовується у функції виводу на екран без збереження результату. Перезапустимо програму, та після використання команди «Step» для переходу до виклику функції «lab5.Program::Decrypt» та використаємо команду «Trace» для входу до ії тіла. За допомогою вікна «Locals» бачимо, що до функції у якості параметрів keyBytes та input були передані ключ та шифрований рядок (рис 4.5).

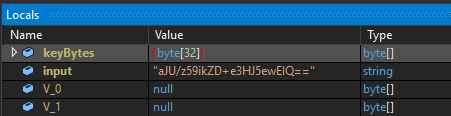


Рисунок 4.5 – Вхідні параметри функції «lab5.Program::Decrypt»

Продовжимо використання команди «Step» для проходу функції. Після проходження виклику системної функції «System.Text.Encoding::GetString», бачимо, що до змінної V\_9 була записана дешифрований рядок. Але системна функція «System.Text.Encoding::GetString» є лише функцією конвертації масиву байт до рядку. Аналіз потребує додаткове використання статичного методу для виявлення масиву який конвертується, та його використання в інших функціях. Функція «System.Text.Encoding::GetString» використовує три параметри, перший з яких є масивом байт для переводу. Оскільки параметри для передачі до функції заносяться до стеку перед ії викликом, подивимося на 3ю команду до виклику системної функції, це є команда «ldloc.1». Це означає, що до стеку була занісена змінна V\_1, тобто в ній знаходиться масив байт для конвертації. Знайдемо використання команди «stloc.1», тобто збереження даних до змінної V\_1. Команда використовується на рядку 117, де за допомогою команди «newarr» створюється новий масив. Далі змінна знов зберігається на стек та використовується віртуальним викликом системної функції «System.IO.Stream::Read». Ця функція зчитує дані з потоку який є першим параметром. Припустимо, що цей параметр є потоком об’єкту дешифратора. Він передається до функції «System.IO.Stream::Read» за допомогою команди «ldloc.s reader». До цього збереження до змінної «reader» є результатом роботи функції «System.Security.Cryptography.CryptoStream::.ctor» (рис 4.6).

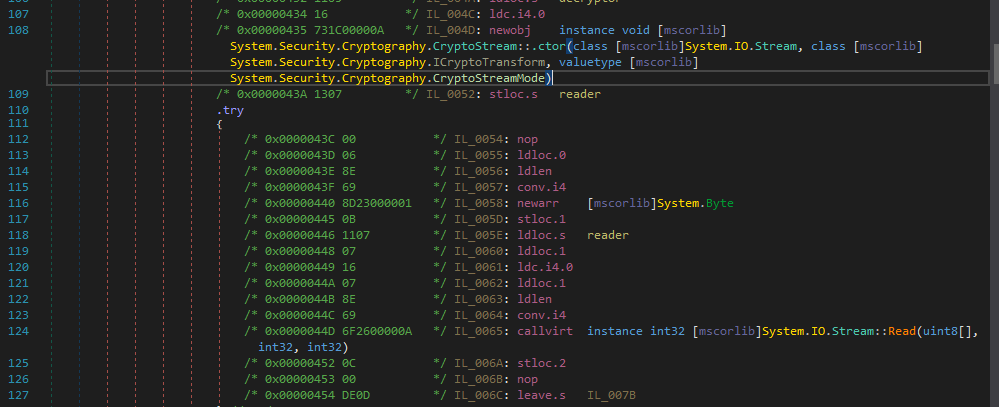


Рисунок 4.6 – Використання дешифратору

Поток дешифратору знайден. Перезапустимо програму та за допомогою команд «Step» та «Trace» перейдемо до тіла функції «System.Security.Cryptography.CryptoStream::.ctor». Оскільки виклик функції «System.IO.Stream::Read» є віртуальним, це означає, що функція яка викликалась насправді, знаходиться у об’єкті класа який був переданий першим на стек. В цьому випадку це «reader». Оскільки функція «System.Security.Cryptography.CryptoStream::.ctor» є конструктором класу «System.Security.Cryptography.CryptoStream» та ії результат був записан до «reader», це означає, що віртуальний виклик функції «System.IO.Stream::Read» насправді викликав функцію «System.Security.Cryptography.CryptoStream::Read». Ця функція є документованою як частина криптобіблотеки «System.Security.Cryptography», функція дешифрації знайдена, аналіз закінчено (рис 4.7).

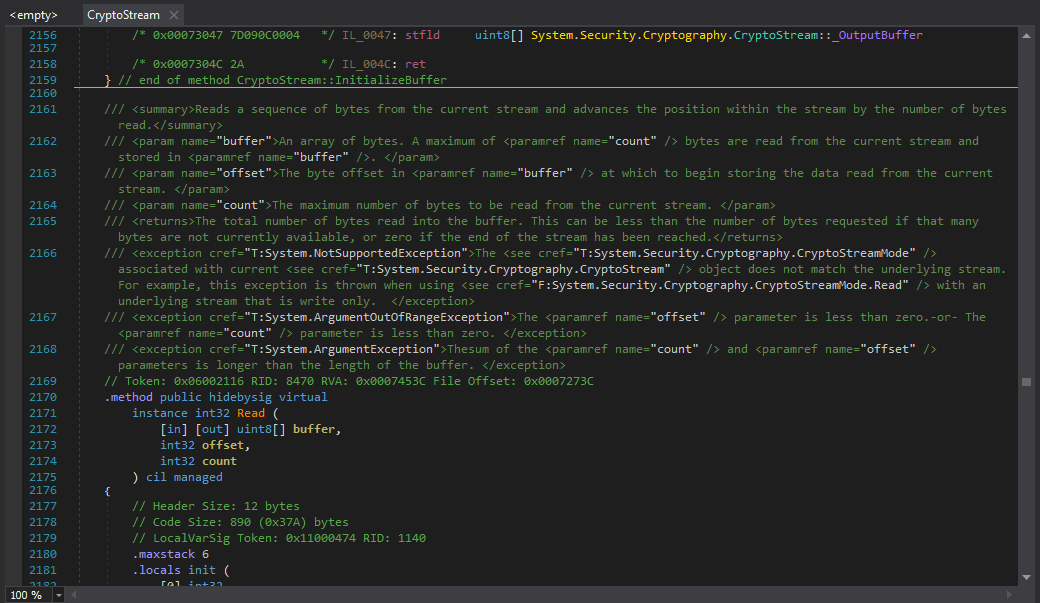


Рисунок 4.7 – Функція дешифрації

### 4.2.3 Результати застосування метод апаратної точки зупинки

Так як досліджувана програма написана на мові програмування C#, використовування методу апаратної точки зупинки є неможливим. В C# та віртуальній машині CLR (Common Language Runtime) немає підтримки встановки апаратних точок зупинки. Але навіть якщо такі точки зупинки підтримувались, цей метод є неефективним, оскільки всі дані у віртуальній машині розташовані у динамічно розподіленій пам’яті. Також у CLR наявний «прибиральник сміття», який завжди управляє пам’ятю та оптимізує ії для раціонального розподілення. Дані які знаходилися в одному місці у пам’яті, церез деякий час можуть бути переміщені «прибиральником сміття» для оптимізації.