Міністерство освіти і науки України

Національний університет «Одеська політехніка»

Інститут інформаційної безпеки, радіоелектроніки та телекомунікацій

Кафедра кібербезпеки та програмного забезпечення

Чебанік Валерія Олександрівна,

студентка групи РЗ-181

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

Захист виконавчих файлів програм від несанкціонованого використання з застосуванням необоротної обфускації

Спеціальність:

125 Кібербезпека

Спеціалізація, освітня програма:

Кібербезпека

Керівник:

Стопакевич Олексій Аркадійович,

доцент, кандидат технічних наук

Одеса – 2022

Міністерство освіти і науки України

Національний університет «Одеська політехніка»

Інститут інформаційної безпеки, радіоелектроніки та телекомунікацій
Кафедра кібербезпеки та програмного забезпечення

Рівень вищої освіти перший (бакалаврський)

Спеціальність 125 – Кібербезпека

Освітня програма – Кібербезпека

ЗАТВЕРДЖУЮ

Завідувач кафедри КБПЗ

_____

д.т.н., проф. А.А.Кобозєва

____ _____2022р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

Чебанік Валерії Олександрівні

Тема роботи: *Захист виконавчих файлів програм від несанкціонованого використання з застосуванням необоротної обфускації*

1. Керівник роботи: *Стопакевич А.В. к.т.н., доцент*.

затверджені наказом ректора від „17” 05.2022р. № 168-в.

2. Зміст роботи: *розробка алгоритму та програмного комплексу для збирання і систематизації інформації про користувачів ІМ-додатків*

3. Перелік графічного матеріалу*: презентація*

4. Консультанти розділів роботи

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Охорона праці	*доц. Ярова І.А.*		

Дата видачі завдання “_____”_____2022 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання	Примітка
1	*Аналіз літератури з теми кваліфікаційної роботи*	*01-03-2022*	*виконано*
2	*Аналіз ІМ-додатків*	*20-03-2022*	*виконано*
3	*Розробка алгоритму*	*05-04-2022*	*виконано*
3	*Розробка програми*	*20-04-2022*	*виконано*
4	*Збирання і систематизація інформації про користувачів ІМ-додатків*	*29-04-2022*	*виконано*
5	*Підготовка пояснювальної записки.*	*16-05-2022*	*виконано*
6	*Підготовка презентації та доповіді*	*23-05-2022*	*виконано*
7	*Попередній захист*	*01-06-2022*	*виконано*
8	*Нормоконтроль*	*18.06.2022*	*виконано*

Здобувач вищої освіти _____ *Душейко М.Ю.*

Керівник роботи _____ *Стопакевич О.А.*

ЗАВДАННЯ

на розробку розділу “Охорона праці”

Чебанік Валерії Олександрівні, група РЗ-181

Інститут інформаційної безпеки, радіоелектроніки та телекомунікацій
Кафедра кібербезпеки та програмного забезпечення

Тема роботи: *Захист виконавчих файлів програм від несанкціонованого використання з застосуванням необоротної обфускації*

Зміст розділу:

1 Аналіз умов праці і вибір основних заходів виробничої безпеки.
2 Аналіз пожежної безпеки. Вибір заходів та засобів пожежної безпеки.

Керівник роботи	Консультант з охорони праці
_____ (О.А.Стопакевич)	_____ (Ярова І.А.)
«____» _____ 2021 р.	«____» _____2021р.

АНОТАЦІЯ

Кваліфікаційна робота на тему «Захист виконавчих файлів програм від несанкціонованого використання з застосуванням необоротної обфускації» на здобуття першого (бакалаврського) рівня вищої освіти за спеціальністю 125 – Кібербезпека, спеціалізація, освітня програма: Кібербезпека, містить 9 рисунків, 2 додатки, 9 літературних джерела за переліком посилань. Робота виконана на 44 сторінках загального тексту і 42 сторінках основного тексту.

Піратство програмного забезпечення є однією з основних проблем для розробників. Для захисту програмного забезпечення від незаконного розповсюдження недостатньо ліцензії, необхідно захистити програму від зворотної інженерії. Існують сотні способів захисту виконуваних файлів, однак вони також мають свій обхід. Ми створили програму-протектор мовою C++ під операційну систему Windows. Переважно цей протектор було розроблено для захисту власного програмного продукту, щоб захистити його від неасоційованого використання. Цей протектор є конкурентоспроможною програмою, так як у відкритому доступі не має подібних програм, тільки у платному сегменті. Перевагою розробленого протектору є те, що він має власну віртуальну машину, метод обфускації та унікальну ентропію, яка не схожа навіть на платні аналоги. Також перевагою є те, що для вже давно існуючих аналогів – зловмисники навчились відновлювати інструкції, а щоб виконати те саме, після накриття новим розробленим протектором з нуля – потрібно дуже багато часу.

ПІРАТСТВО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ОБФУСКАЦІЯ, ЗВОРОТНЕ ІНЖЕНЕРСТВО, ЗАХИСТ ВИКОНАВЧИХ ПРОГРАМ, НЕСАНКЦІОНОВАНЕ ВИКОРИСТАННЯ

ANNOTATION

Thesis on the topic "Protection of program executable files from unauthorized use using irreversible obfuscation" for the first (Bachelor's) level of higher education in specialty 125 in cybersecurity, specialization, educational program: cybersecurity, contains 9 Figures, 2 Appendices, 9 literature sources on the list of references. The work was completed on 52 pages of general text and 44 pages of main text.

Software piracy is a major issue for developers. It is not enough to protect the software from illegal distribution, it is necessary to protect the program from reverse engineering. There are hundreds of ways to protect executables, but they also have a workaround. We have created a protector program in C ++ for the Windows operating system. Preferably, this protector was designed to protect your own software product to protect it from unauthorized use. This protector is a competitive program, as there are no similar programs in the public domain, only in the paid segment. The advantage of the developed tread is that it has its own virtual machine, obfuscation method and unique entropy, which is not even similar to paid counterparts. Another advantage is that for long-existing counterparts - attackers have learned to restore the instructions, and to do the same, after covering the newly developed tread from scratch - takes a long time.

SOFTWARE PIRACY, OBFUSCATION, REVERSE ENGINEERING, PROTECT EXECUTABLES, UNAUTHORIZED USE

ЗМІСТ

ВСТУП

Модуль, що виконується (здійснюваний), виконуючий файл - файл, що містить програму у вигляді, в якому вона може бути виконана комп'ютером, після завантаження в пам'ять і налаштування за місцем. Найчастіше він містить двійкове уявлення машинних інструкцій для певного процесора (з цієї причини на програмістському сленгу щодо нього використовують слово «бінарник» — кальку з англійської binary, але може містити інструкції мовою програмування, що інтерпретується, для виконання яких потрібен інтерпретатор. Щодо останніх часто використовується термін «скрипт».

Виконанням бінарних файлів займаються апаратно- та програмно-реалізовані машини. До перших відносяться процесори, наприклад сімейств x86 або SPARC. До других - віртуальні машини, наприклад, віртуальна машина Java або .NET Framework. Формат бінарного файлу визначається архітектурою виконує його машини. Відомі машини, реалізовані як апаратно, і програмно, наприклад, процесори сімейства x86 і віртуальна машина VMware.

Статус виконання файлу найчастіше визначається прийнятими угодами. Так, в одних операційних системах здійсненні файли розпізнаються завдяки угоді про іменування файлів (наприклад, шляхом вказівки в імені розширення файлу -. операційних системах). У сучасних комп'ютерних архітектурах файли містять великі обсяги даних, що не є комп'ютерною програмою: опис програмного оточення, в якому програма може бути виконана, дані для налагодження програми, використовувані константи, дані, які можуть знадобитися операційній системі для запуску процесу (наприклад, рекомендований розмір купи), і навіть описи структур вікон графічної підсистеми, що використовуються програмою.

Найчастіше виконані файли містять дзвінки бібліотечних функцій, наприклад, дзвінки функцій операційної системи. Таким чином, поряд з процесоро-залежністю (машино-залежним є будь-який бінарний здійсненний

файл, що містить машинний код) файлам може бути властива залежність від версії операційної системи та її компонент.

В даний час існує велика кількість файлових форматів виконуваних файлів для різних операційних систем. При цьому документація, хоч і існує, часто розрізнена або практично недоступна (наприклад, опис формату LX/LE стає все важчим і важчим знайти). Плюс до цього різні виробники програмного забезпечення вводять додаткові розширення, які часто стає важко зіставити з оригінальним.

Що стосується побудови файлів такого типу, як правило, вони містять заголовки (передбачуване виконання інструкцій, параметри, формати коду), самі інструкції (машинні, вихідні або байт-коди). Іноді в структуру можуть включатися опису оточення, дані для налагодження, вимоги до ОС, списки супутніх бібліотек, зображення, звук, графіка, іконки ярликів і т. д. до Речі, багато хто, напевно, звертали увагу, що в більшості своїй в кожній операційній системі іконка у такого файлу є спочатку (за умови, що він зі старту повинен нею працювати).

1 АНАЛІЗ ПРОБЛЕМИ

1.1 Аналіз форматів виконуючих файлів та статичних бібліотек в операційних системах

В залежності від типу вмісту файлу визначається його розширення. Наприклад, в Windows найбільш поширеним виконувані файли мають розширення EXE. Це відноситься до всіх програм, розрахованим на роботу в середовищі цих операційних систем. Такі об'єкти містять машинні коди. Дуже схожими (теж з вмістом таких кодів) є файли BIN. Ще одним типом виконуваних файлів є пакетні об'єкти CMD, BAT і COM, причому перший тип є пакетним файлом Windows, а другий і третій відносяться до систем DOS. Ймовірно, багато зустрічали і файли MSI і MSU. Це може бути або «рідний» інсталятор Windows installer апдейтів системи.

Окрему категорію складають скрипти і макроси (VB, VBS, VBE, SCR, JS, JSE). Також часто зустрічаються файли начебто JAR і JAD, призначені для установки додатків на мобільні гаджети або використання в середовищі JAVA. Всі такі об'єкти в змісті мають вже не машинні коди, а коди віртуальних машин.

Якщо більш уважно аналізувати, можна примітити, що в деяких системах зустрічаються досить специфічні компоненти. Наприклад, в Windows є спеціальна категорія виконуваних компонентів (PS1 – виконуваний файл PowerShell, PIF – інформація про програми, WSF – файл сценарію тощо). Взагалі, в будь-якій системі можна знайти і стандартні, і спеціальні компоненти. Проте є і деякі загальні формати (наприклад, HTA – виконуваний документ HTML), які працюють скрізь і незалежно від застосовуваної «операційки». Що стосується інших систем, наприклад, в «маках» виконувані файли мають розширення APP, якщо це програми, і PKG, якщо це дистрибутив. А ось з Linux-системами справа йде трохи інакше. Справа в тому, що в них поняття розширення відсутнє як таке. Розпізнати виконуваний файл можна тільки по атрибутах (прихований,

тільки для читання, системний тощо). Таким чином, відпадає навіть проблема зміни розширень, щоб запустити або прочитати шуканий файл. Втім, у будь-якій системі або навіть у мобільних гаджетах можна знайти величезну кількість об'єктів такого типу. Далеко ходити не потрібно. У тій же ОС Android виконуваний файл інсталятора має розширення APK, а в «яблучних» девайсах це файли IPA.

Одним з найпростіших форматів виконуваних файлів на архітектурі x86 є файл з розширенням. імені файлу). Витоки даного формату файлу йдуть з операційної системи CP/M для процесора Intel 8080, відомої широкою популярністю. Формат файлу передбачає використання створення команд (COM - скорочення від COMMAND), що розширює базову функціональність операційної системи.

Під час створення MS-DOS формат .COM зберігся майже зміни і забезпечує сумісність лише на рівні вихідного коду з CP/M. Двійковий формат є специфічним для кожного типу процесора. Формат має на увазі виконання в межах одного сегмента (64Кб), що не дозволяло використовувати можливості оперативної пам'яті повною мірою. Формат двох байт файлу на відміну від формату .COM, де для коду, даних та стека використовується той самий сегмент, у форматі .EXE були зняті дані обмеження, що робило доступним весь адресний простір комп'ютера.

Розвиток машин класу IBM PC призвело до появи нових режимів адресації, появі віртуальної пам'яті та інших механізмів, що робило формат .EXE MZ недостатньо гнучким та не пристосованим до реалій. Зокрема, багато виробників реалізовували так звані оверлеї, що дозволяло здійснювати найпростіше підкачування потрібних функцій в ОЗУ. Плюс, неефективне використання повторюваного коду, що виконується, призвело до реалізації бібліотек динамічного зв'язування, код яких міг використовуватися різними процесами без його дублювання у фізичній пам'яті. Через війну формат .EXE було розширено і було реалізовано його найсучасніші версії LX/LE і NE. Формат .EXE NE був розроблений Microsoft для застосування в операційній

системі OS/2 та оболонки Windows. Цей формат орієнтований, як і .EXE MZ, на 16-розрядні середовища виконання. Основна додаткова функціональність, що забезпечується даним форматом, - це підтримка .DLL.

З приходом на ринок мікропроцесорів 386+ виникла потреба підтримки 32-розрядного коду, що, природно, призвело до появи такого формату, як LE (використовувався в оболонках сімейства Windows 3.x та операційних систем сімейства Windows 9x) та LX (використовувався в операційній системі) OS/2). Дані формати дозволили змішувати як 16-розрядний код, так і 32-розрядний. Крім змішаного коду перехідний період також зажадав реалізації такого механізму, як thunking, що також наклало відбиток на дані формати файлів.

Під час розробки OS/2 NT 3.0 (надалі - Windows NT 3.51) розробили формат .EXE PE, який було призначено зберігання коду орієнтованого різні апаратні платформи. В даний час цей формат є основним для сімейства операційних систем Windows. У крос платформної OS/2 (OS/2 PPC, Workstation OS) кінцевим форматом файлів був прийнятий формат ELF, про який буде сказано трохи далі.

У світі операційних систем сімейства Unix також спостерігався розвиток форматів файлів, що виконуються. Перший формат a.out з'явився з першою версією UNIX. На назву вплинула специфіка процесу отримання двійкового файлу. На відміну від систем сімейства CP/M, для яких завжди була характерна нестача пам'яті, Unix системи дозволяли здійснювати повний цикл: компіляція, компіляція, компонування. a.out – це скорочення від assembler output. Формат a.out структурою схожий на формати .EXE. За своє існування зазнала низка модифікацій.

З появою бібліотек, що розділяються, формат a.out через ряд обмежень був замінений на формат COFF. З основних нововведень - це додавання налагоджувальної інформації та відносної віртуальної адреси, що дозволило завантажувати його за довільною фактичною адресою. Використання

формату COFF в Unix системах в даний час обмежено, проте формат .EXE PE є найвідомішим варіантом формату COFF.

Формат COFF був прийнятий не всіма виробниками сімейства Unix і багато хто все ще продовжував використовувати a.out. Ситуація змінилася із появою формату ELF. Досить вдале рішення та врахування того, що формат не був орієнтований під специфічні особливості певної архітектури, він набув широкого поширення. Більшість сучасних реалізацій Unix та низка інших операційних систем використовують саме його. Існує також "універсальний" формат ELF, що містить двійкові образи для більшості різних платформ. Складно сказати, чи буде формат FatELF широко використаний, але якщо озирнутися на історію подібний до "багатосистемних" форматів, то, швидше за все, FatELF не набуде широкого поширення.

Наявність великої кількості форматів, що накопичилися з часом, призвело до того, що в кінці дев'яностих років з'явилося ряд дослідницьких проектів з розробки деякого "абстрактного" формату файлів, а також бібліотек, що надають єдиний інтерфейс доступу до даних будь-якого формату. Причин виникнення таких інструментів було кілька. Одна з причин – інструментарій для середовищ розробки. Наприклад, підтримка компонувальником найбільшого числа форматів. Інша причина - необхідність перенесення та запуску двійкових програм на іншій апаратній платформі.

Статична бібліотека (або «архів») складається з підпрограм, які безпосередньо компілюються та лінкуються з вашою програмою. При компіляції програми, яка використовує статичну бібліотеку, весь функціонал статичної бібліотеки (той, що використовує ваша програма) стає частиною файлу, що виконується. У Windows статичні бібліотеки мають розширення .lib (скор. від "library"), тоді як у Linux статичні бібліотеки мають розширення .a (скор. від "archive").

Однією з переваг статичних бібліотек є те, що вам потрібно поширити лише виконуваний файл, щоб користувачі могли запустити та використовувати вашу програму. Оскільки статичні бібліотеки стають частиною вашої програми, то ви можете використовувати їх подібно до функціоналу своєї власної програми. З іншого боку, оскільки копія бібліотеки стає частиною кожного виконуваного файлу, це може призвести до збільшення розміру файлу. Також, якщо вам потрібно буде оновити статичну бібліотеку, вам доведеться перекомпілювати кожен файл, який її використовує.

Виконуваний файл можна пов'язати з бібліотекою DLL (завантажити її) одним із двох способів:

Неявне зв'язування — операційна система завантажує бібліотеку DLL у той час, коли вона використовується виконуваним файлом. Виконуваний файл клієнта викликає експортовані функції бібліотеки DLL так само, як статично скомпоновані і включені до складу виконуваного файлу функції. Процес неявного зв'язування також іноді називають статичним завантаженням або динамічним компонуванням часу завантаження.

Явне зв'язування — операційна система завантажує бібліотеку DLL на запит під час виконання. Виконуваний файл, який використовує бібліотеку DLL, має явно завантажувати та вивантажувати її. Крім того, в ньому повинен бути налаштований вказівник функції для доступу до кожної функції, що використовується з бібліотеки DLL. На відміну від викликів функцій у статично скомпонованій або неявно пов'язаній бібліотеці DLL, під час роботи з явно пов'язаною DLL виконуваний файл клієнта повинен викликати експортовані функції за допомогою покажчиків функцій. Процес явного зв'язування також іноді називають динамічним завантаженням або динамічним компонуванням часу виконання.

Для зв'язування з однією бібліотекою DLL виконуваний файл може використовувати будь-який з цих способів. Крім того, вони не є

взаємовиключними, тобто два різні файли, що виконуються, можуть зв'язуватися з однією бібліотекою DLL різними способами.

Неявне зв'язування відбувається в момент, коли з коду програми викликається експортована функція бібліотеки DLL. При компіляції або збиранні вихідного коду виклику виконуваного файлу для виклику функції DLL у коді об'єкта створюється посилання на зовнішню функцію. Для дозволу цього зовнішнього посилання програма має зв'язатися з бібліотекою імпорту (LIB-файл), яку надає розробник бібліотеки DLL.

Бібліотека імпорту містить лише код для завантаження бібліотеки DLL та реалізації викликів її функцій. Під час пошуку зовнішньої функції в бібліотеці імпорту компонувальник визначає, що код цієї функції знаходиться у бібліотеці DLL. Для дозволу зовнішніх посилань на бібліотеки DLL компонувальник просто додає у файл відомості, на підставі яких система визначає, де слід шукати код DLL при запуску процесу.

При запуску програми, яка містить посилання, що динамічно зв'язуються, система використовує відомості з виконуваного файлу для пошуку необхідних бібліотек DLL. Якщо знайти бібліотеку DLL не вдалося, система завершує процес і відображає діалогове вікно з повідомленням про помилку. В іншому випадку система зіставляє модулі DLL в адресному просторі процесу.

1.2 Аналіз того, що потрібно захищати в виконуваному файлі.

В першу чергу в виконуваному файлі потрібно обфусцирувати Entry Point, щоб зловмисник та типові програми для взлому не могли отладити належним чином програму. Взагалі Entry Point (або Точка входу) – адрес в оперативній пам’яті комп’ютера, з якого починається виконання програми. Іншими словами – це адреса за якою зберігається перша команда програми.

Обфускація або ж іншими словами заплутування коду — приведення початкового коду або виконуваного програмного коду до

вигляду, який зберігає його функціональність, але ускладнює аналіз, розуміння алгоритму роботи і модифікації при декомпіляції.

«Заплутування» коду може здійснюватися на рівні алгоритму, початкового коду та/або асемблерного коду. Для створення заплутаного асемблерного коду можуть використовуватися спеціалізовані компілятори, які використовують неочевидні або недокументовані можливості середовища виконання програми. Існують також спеціальні програми, що здійснюють обфускацію, які називаються обфускаторами. Зазвичай, обфускація на рівні машинного коду зменшує швидкість виконання і відповідно збільшує час виконання програми. Тому вона застосовується в критичних до безпеки, але не критичних до швидкості місцях програми, таких як перевірка реєстраційного коду. Найпростіший спосіб обфускації машинного коду — вставка в нього недіючих конструкцій (таких як `xor`).

Перш за все, обфускація коду проводиться для безпеки програмного продукту. Розробник може переслідувати і комерційні цілі (конкурентний захист від підробки або приховування значень/логіки). В результаті виходить стислий (видалені невикористовувані класи, атрибути, методи) або оптимізований (перевірені та переписані оператори) софт. Обфусцований код застосовується, наприклад, на Android та Java.

Процедура виконується вручну (довго, складно привести у вихідний вигляд - тобто "деобфусцировать") або автоматично (швидко, виконується спеціальними програмами "обфускаторами" з функцією "деобфускації"). Завдання виконує програміст з метою, щоб жодний інший програміст не зміг прочитати програмний код і розшифрувати алгоритми обфускатора.

В другу чергу потрібно мутувати код Call-ів, визови API-функцій, точніше створення віртуальних інструкцій, щоб той же патчер не зміг за патернами знайти потрібні строки в пам’яті.

Мутація коду - це один із найпростіших і основних способів захисту, що полягає в заміні асемблерних інструкцій на їх повні аналоги. Мутація коду практично не погіршує швидкодію коду, що захищається, і призначена для ускладнення пошуку типових сигнатур коду, які замінюються в використовуваних бібліотеках та які типові для конкретних компіляторів. Також мутація передбачає переупорядкування інструкцій без зміни їхньої результуючої дії. Мутацію коду також іноді позначають терміном metamorf.

В інтерпретованих мовах обфускований код займає менше місця, ніж початковий, і часто виконується швидше, ніж початковий. Сучасні обфускатори також замінюють константи числами, оптимізують код ініціалізації масивів, і виконують іншу оптимізацію, яку на рівні початкового тексту провести проблематично або неможливо.

Захист виконуваних файлів найбільш гострий для компаній, представлених у наступних сферах:

- діяльність автоматизованих систем управління технологічними процесами підприємств;
- виробництво засобів захисту інформації;
- виробництво інформаційних та телекомунікаційних систем, захищених з використанням засобів захисту інформації;
- міжнародна діяльність;
- діяльність, спрямована на протидію тероризму;
- діяльність із забезпечення безпеки у надзвичайних ситуаціях;
- діяльність із забезпечення безпеки у сфері використання атомної енергії;
- бази даних, що містять конфіденційну інформацію;
- власний продукт (софт).

1.3Аналіз принципів захисту виконуваних програм від злому

У процесі захисту незахищений виконуваний файл розкладається на складові. Складові частини файлу, що виконується, перетворюються з використанням різних технологій захисту. Також файл доповнюється допоміжними частинами для захисту. Для використання деяких технологій захисту необхідно використовувати SDK, тобто вносити зміни у вихідний код продукту, що підлягає захисту.

Існує два методи захисту даних: перенесення даних у контейнер (приховування даних) та перевірка цілісності файлів.

Read-only дані захищаються шляхом поміщення окремих файлів або цілих каталогів у захищений контейнер. Для зовнішнього світу контейнер виглядає як один великий файл із зашифрованим вмістом. Додаток може читати файли з контейнера одним з двох способів:

- Контейнер як віртуальна файлова система. Для реалізації цього способу разом із додатком до системи автоматично встановлюється спеціальний драйвер. Він забезпечує прозору для програми роботу з файлами з контейнера: програма замість контейнера «бачить» файли всередині нього;
- Доступ до файлів через API. У цьому випадку драйвер не потрібен, але доступ до файлів повинен здійснюватися не через стандартні функції операційної системи, а через API-функції зі складу StarForce SDK.

Read-only дані зберігаються у відкритому вигляді на диску. Перевірка здійснюється через виклик API функції. Ця функція звіряє цифровий підпис, зашифрований у контейнері, із цифровим підписом файлу даних. При інтеграції захисту файл даних повинен бути доданий до контейнера з опцією «Використовувати цифровий підпис замість файлу». Таким чином, ускладняється аналіз та модифікація файлів даних захищеної програми.

Для полегшення реалізації захисту власної DRM створюється підтримка зовнішньої прив'язки. Вона потрібна для того, щоб забезпечити невід'ємність функціонала з перевірки автентичності ліцензії від програми,

що захищається, а також для захисту цього функціонала від аналізу та модифікації.

Ідеального способу захисту програмного забезпечення від несанкціонованого використання та розповсюдження не існує. Жодна існуюча система не забезпечить абсолютний захист і не позбавить потенційного зломщика самої можливості її нейтралізації. Однак використання якісного та ефективного захисту може максимально ускладнити процес злому програмного продукту, більш того, зробити злом недоцільним з погляду витраченого на це часу та зусиль. При створенні захисту програмного продукту можуть переслідуватися різні цілі та вирішуватися різноманітні завдання, проте основа будь-якої схеми захисту – захист програми від вивчення, оскільки саме стійкість до зворотної інженерії визначає ефективність захисту загалом.

Вивчення програмного продукту може здійснюватися методом статичного та динамічного аналізу. При статичному аналізі розробка алгоритму злому захисту проводиться на основі аналізу результатів дизасемблювання або декомпіляції програми, що зламується. Динамічний аналіз найчастіше застосовують при зломі програмних продуктів, виконуваний код яких зашифрований або динамічне змінюється, тому що статичний аналіз подібних програм пов'язаний із певними труднощами.

Для динамічного аналізу програму, що зламується, запускають у середовищі відладчика, при цьому стає можливим контроль усіх змін, що виникають у процесі функціонування програми. У процесі динамічного аналізу, використовуючи режим налагодження, виробляють покрокове проходження «захисних» алгоритмів програми, зокрема алгоритми перевірки та генерації коректного реєстраційного коду. Ще одним засобом, що застосовується при динамічному аналізі, є монітори активності, що відстежують звернення програми, що зламується, до файлів, системних сервісів, портів і зовнішніх пристроїв.

Основним інструментом, що застосовується для захисту програм від злому, є програми-протектори. Захист, реалізований більшістю протекторів, у тому, що вони упаковують і/або шифрують вихідний виконуваний файл, приділяючи основну увагу захисту процедури розпакування/розшифровки файла.

Подібний алгоритм роботи протекторів дуже часто виявляється причиною недостатнього ступеня захисту, що забезпечується більшістю з них. Якщо програма захищена з використанням упаковки, після закінчення роботи розпакувальника в розпорядженні зловмисника може виявитися вихідний файл, який може бути отриманий при дампі певної області пам'яті. Більш того, для боротьби з найпоширенішими протекторами зломщики розробили безліч програмних інструментів, що дозволяють зламувати захист в автоматичному режимі. Аналогічний підхід застосовується при боротьбі з шифруванням: після отримання ліцензійного ключа, який може бути легально придбаний, зломщик зможе розшифрувати захищені ділянки коду.

Ряд програм-протекторів застосовують різні антиналагоджувальні прийоми. Проте застосування антиналагодження значно знижує швидкодію програми. Також слід пам'ятати, що антиналагоджувальні прийоми дієві лише проти динамічного аналізу і не ефективні проти статичного. Більше того, всі антиналагоджувальні прийоми, які застосовуються в сучасних протекторах, вже давно вивчені, а зломщиками написано безліч утиліт, які їх повністю нейтралізують. На ефективність використання моніторів активності вбудовані в програму засоби захисту від налагодження не впливають.

Більш ефективними методами захисту програми є обфускація і віртуалізація, що ускладнюють аналіз коду програми, що захищається. Загалом ефективність цих методів досягається з допомогою використання особливостей людського чинника — що складніший вихідний код, що більше ресурсів використовує додаток, тим людині його аналізує важче зрозуміти логіку роботи програми, отже, і зламати застосовані засоби захисту.

Під час обфускування виконується заплутування коду програми за рахунок введення додаткових інструкцій. При віртуалізації вихідний код перетворюється на байт-код, виконання якого здійснюється на спеціальному інтерпретаторі, що імітує віртуальну машину зі специфічною системою команд. Отже, застосування віртуалізації призводить до високого та незнижувального ступеня заплутаності результуючого коду, а за певного підходу до реалізації цього методу захищений код не містить методів відновлення оригінального коду у явному вигляді. Таким чином, найважливішою перевагою віртуалізації є те, що в момент виконання віртуалізованої ділянки коду не відбувається його зворотного перетворення на машинні коди процесора, а це виключає можливість отримання зломщиком оригінального програмного коду.

Завдання зворотного інжинірингу віртуалізованих фрагментів зводиться до вивчення архітектури віртуальної машини, створення дизассемблера, відповідного архітектурі віртуальної машиною процесора, що імітується, і аналізу дизассемблированного коду. При певному підході до реалізації віртуальної машини завдання створення дизассемблера віртуалізованого коду стає досить трудомістким. Єдиним недоліком віртуалізації є порівняно низька швидкість роботи, тому цей метод слід застосовувати тільки для захисту ділянок коду, некритичних до виконання.

У переважній більшості сучасних протекторів методи обфускації та віртуалізації відіграють другорядну роль, а рівень їхньої реалізації недостатній, що дозволяє зломщикам виконувати зняття подібного захисту в автоматизованому або ручному режимі. Ще одним слабким місцем багатьох сучасних протекторів є використання недокументованих функцій Windows, що створює обмеження для застосування в нових версіях операційної системи або при включеному режимі DEP.

Загальні бібліотеки - файли .so (або Windows.dll, або в OS X.dylib). Весь код, пов'язаний з бібліотекою, міститься в цьому файлі, і на нього посилаються програми, які використовують його під час виконання.

Програма, яка використовує спільну бібліотеку, посилається лише на код, який він використовує у спільній бібліотеці.

Статичними бібліотеками є .a(або Windows.lib) файли. Весь код, пов'язаний з бібліотекою, знаходиться в цьому файлі, і він безпосередньо пов'язаний із програмою під час компіляції. Програма, яка використовує статичну бібліотеку, бере копії коду, який він використовує зі статичної бібліотеки, і робить її частиною програми. [Windows також має .lib файли, які використовуються для посилання .dll файлів, але вони діють так само, як і перший.

У кожному методі є переваги та недоліки. Загальні бібліотеки зменшують обсяг коду, який дублюється у кожній програмі, яка використовує бібліотеку, зберігаючи мінімальні двійкові файли. Він також дозволяє замінити загальний об'єкт на той, що функціонально еквівалентний, але може мати додаткові переваги продуктивності, не перекомпілюючи програму, що її використовує. Однак загальні бібліотеки будуть мати невелику додаткову вартість для виконання функцій, а також вартість завантаження під час виконання, оскільки всі символи в бібліотеці повинні бути пов'язані з ними. Крім того, загальні бібліотеки можуть бути завантажені в додаток під час виконання, що є загальним механізмом для реалізації двійкових систем, що підключаються. Статичні бібліотеки збільшують загальний розмір двійкового файлу, але це означає, що вам не потрібно носити копію бібліотеки, яка використовується. Оскільки код підключено під час компіляції, додаткові витрати на завантаження не потрібні. Код просто там. Особисто я віддаю перевагу бібліотекам, що розділяються, але використовую статичні бібліотеки, коли вам потрібно переконатися, що в двійковому коді не так багато зовнішніх залежностей, які можуть бути важко зустріти, наприклад, конкретні версії стандартної бібліотеки C++ або конкретні версії Boost C++ бібліотека.

Статична бібліотека схожа на книгарню, а бібліотека, що розділяється, схожа на звичайну бібліотеку. З першим ви отримуєте свою копію книги/функції, щоб забрати додому; з останнім ви і всі інші ходите до бібліотеки, щоб використовувати ту ж книгу/функцію. Тому будь-хто, хто хоче використати (загальну) бібліотеку, повинен знати, де це, тому що вам потрібно піти на книгу/функцію. За допомогою статичної бібліотеки книга/функція належить вам, і ви тримаєте її у своєму будинку/програмі, і як тільки ви її отримаєте, вам все одно де і коли ви її отримали.

Статичні бібліотеки скомпільовані як частина програми, а бібліотеки, що розділяються, - ні. Коли ви розповсюджуєте програму, яка залежить від загальних бібліотек, бібліотеки, наприклад. dll на MS Windows.

2 ОПИС ТА ВИБІР ТИПОВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Опис алгоритму захисту виконуваного файлу

Алгоритм захисту виконуваних файлів включає в себе:

- виявлення відладчика - дана функція дозволяє забороняти;
- виконання програми, що захищається під відладчиком;
- захист імпорту — ця функція дозволяє ховати інформацію про API, що використовуються у файлі;
- захист ресурсів - дана функція запобігає вилученню та модифікації ресурсів програми (у системах Win32/Win64);
- захист пам'яті — дана функція дозволяє захищати програму від атаки шляхом зняття дампа пам'яті програми, що працює.

Існує безліч методів захисту від кряка. Я перерахую найефективніші сучасні методи захисту, а далі у статті ми розглянемо кожен із них докладніше:

- створення залежностей між протектором та основним кодом;
- захист від налагодження програми та зміни асемблерного коду;
- обфускація (мутація) нативного коду;
- віртуалізація нативного коду.

Нативним кодом називатимемо код, що виконується безпосередньо на процесорі (тобто те, що ми отримуємо після компіляції нашого коду під цільову архітектуру). Відладник – у нашому випадку програма для покрокового обходу та вивчення роботи асемблерного коду, що дозволяє побачити значення всіх змінних у процесі виконання та послідовність виконання коду.

Як тільки ми додали перевірку ліцензії в нашу програму, пірату достатньо було лише видалити з асемблерного коду, і він отримував версію програми, що працює без ліцензії. Необхідно написати програмне забезпечення так, щоб від наявності протектора залежала подальша робота. Один із найбільш очевидних способів – це додавання залежностей між протектором та функціональною частиною. Наприклад, ініціалізація важливих значень та об'єктів, які використовуються далі. При цьому значення будуть залежати від ліцензійного ключа або будуть отримані через інтернет під час перевірки ліцензії.

Тепер реверс інженер не може просто так видалити протектор, оскільки це призведе до подальшої неправильної роботи або фарбування в програмі. Але ПЗ все ще погано захищено, тому що під відладчиком можна легко зламати перевірку ліцензування в протекторі або, аналізуючи сам протектор, отримати необхідні для програми значення та ініціалізувати їх в обхід захисту.

Наступним кроком нам необхідно захистити код від налагодження. З цього моменту, крім ліцензування та ініціалізації, наш протектор ловитиме відладчик. Якщо протектор виявить сліди відладчика, він ламає стек фрейми програми чи інакше робить відновлення роботи програми неможливим.

Перше, що нам потрібно зробити, це запускати протектор в іншому потоці. Регістр DR7 – debug control register, може сигналізувати про те, що програма запущена під відладчиком. Також можна перевіряти значення регістрів, що використовуються під час налагодження, таких як DR0 – DR3.

Є безліч можливостей обійти цей захист, але його реалізація в протекторі не буде зайвою.

Другий спосіб детектування відладчика – підписатися на хуки ОС Windows, які викликатимуть перевизначену нами функцію при спробі прикріпити до процесу відладчик або змінити наш нативний код. Наприклад, можемо використовувати хук DbgUiRemoteBreakin(), який викличеться, коли до процесу прикріпиться відладчиком, повідомивши про це. Даний метод не спрацює, якщо реверс-інженер запустить процес спочатку під відладчиком. У такому разі під час виклику хука протектор ще не буде запущений, а значить і зловити його не зможе. Для комплексного захисту використовують велику кількість різних хуків.

Третій спосіб – створення ситуацій, які не впливають на роботу програмного забезпечення, але викликають дії у відладчику. Наприклад, ми можемо викинути виняток, відразу ж його обробити та заміряти час між викидом та обробкою. Виняток у відладчика може спричинити зупинку, після чого, обробивши виняток, за часом протектор зрозуміє, що програма працює під відладчиком.

Четвертий метод – скористуватися інтерфейсом ОС. У Windows API є безліч функцій, що дозволяють дізнатися про наявність відладчика, прикріпленого до програми. Можна використовувати такі функції, як:

- IsDebuggerPresent() – дізнатись чи активний відладчик;
- CheckRemoteDebuggerPresent() – віддалений відладчик;
- NtQueryInformationProcess() – отримує інформацію про процес;
- RtlQueryProcessHeapInformation() – отримує прапори купи процесу;
- RtlQueryProcessDebugInformation() – отримує прапори відладчика процесу;
- NtQuerySystemInformation() – отримує інформацію із системи.

Отримана інформація дозволить визначити наявність відладчика у системі.

Описані вище методи дуже ускладнять запуск коду під відладчиком. Реверс-інженеру доведеться обходити кожен із способів захисту окремо, але через деякий час пірату вдасться запустити наш додаток під відладчиком, тоді наш код знову буде вразливий для кряка. Необхідно сам асемблерний код захистити від аналізу. Для цього підійде обфускація. Обфускація - приведення коду програми до виду, що зберігає її функціональність, але утруднює аналіз, розуміння алгоритмів роботи та модифікацію при декомпіляції. Обфускувати можна як скомпільовану програму, так і вихідний код, що досить актуально для скриптових мов.

У нашому випадку ми будемо заплутувати асемблерний код. Таку обфускацію називають мутацією нативного коду. Для обфускування коду використовують спеціальні програми – обфускатори. Даний метод захисту створює зайві інструкції, заплутує, додає хибні залежності в коді, тим самим роблячи код складнішим для аналізу та реверсу, проте це сповільнює програму. Щоб не сильно впливати на роботу програмного забезпечення ми будемо обфуцювати тільки наш протектор. Оскільки він пов'язаний з роботою програмного забезпечення, реверс-інженеру доведеться розбиратися в роботі мутованого коду протектора. Мутація лише протектора незначно вплине на продуктивність всієї програми, але значно посилить безпеку. Для більшої безпеки можна обфузкувати мутований код протектора повторно.

Незважаючи на все, навіть такий код схильний до аналізу досвідчених реверсерів. Перейдемо до ще потужнішого механізму захисту коду

Більш надійною альтернативою мутації є віртуалізація. При віртуалізації ми виконуємо наш код не безпосередньо на процесорі, а маємо унікальну віртуальну машину, яка виконує захищену ділянку коду. У нашому проекті ми скористаємося методом захисту замість звичайної обфускації. Принцип його наступний: наш нативний код протектора транслюється в байткод і для нього створюється унікальна віртуальна машина. Користувач отримує віртуальну машину з байткодом усередині. При запуску програми користувачем протектор запускається через віртуальну машину. Цей спосіб

захисту знижує продуктивність у тисячі разів, але оскільки протектор – це лише мала частина програмного забезпечення, загальне зниження продуктивності все ще незначне.

2.2 Вибір типового програмного забезпечення

Для реалізації створення програми-протектора захисту виконуваних файлів від несанкційного використання було обрано мову програмування C++.

C++ - компільована, статично типізована мова програмування загального призначення.

Підтримує такі парадигми програмування як процедурне програмування, об'єктно-орієнтоване програмування, узагальнене програмування, забезпечує модульність, роздільну компіляцію, обробку винятків, абстракцію даних, оголошення типів (класів) об'єктів, віртуальні функції. Стандартна бібліотека включає, у тому числі, загальновживані контейнери та алгоритми. C++ поєднує властивості як високорівневих, і низькорівневих мов. У порівнянні з його попередником - мовою C, найбільшу увагу приділено підтримці об'єктно-орієнтованого та узагальненого програмування.

C++ широко використовується для розробки програмного забезпечення, будучи однією з найпопулярніших мов програмування. Область його застосування включає створення операційних систем, різноманітних прикладних програм, драйверів пристроїв, додатків для систем, високопродуктивних серверів, а також розважальних додатків. Існує безліч реалізацій мови C++ як безкоштовних, так і комерційних і для різних платформ. C++ - мова, що складається еволюційно. Кожен елемент C++ запозичувався з інших мов окремо та незалежно від інших елементів (ніщо із запропонованого C++ за всю історію його розвитку не було нововведенням у Computer Science), що зробило мову надзвичайно складною, з безліччю дублюючих та взаємно суперечливих елементів, блоки яких ґрунтуються на

різних формальних базах. Критики C++ не протиставляють йому будь-яку конкретну мову, а навпаки, стверджують, що для будь-якого випадку застосування C++ завжди існує альтернативний інструментарій, що дозволяє вирішити те завдання більш ефективно і якісно. У свою чергу, прихильники C++ вважають некоректним порівнювати різні аспекти C++ з різними мовами, так як загальний набір засобів і можливостей C++ істотно ширше, ніж у більшості мов, з якими проводиться порівняння, і сама по собі широта можливостей , на їхню думку, є вагомим виправданням недосконалості кожної окремо взятої можливості. Більше того, на їхню думку, висока сумісність із Сі є однією з важливих рис мови, і тому всі недоліки C++ виправдані перевагами, що надаються цією сумісністю.

Перевагами C++ можна відокремити:

- висока сумісність із мовою Сі;
- обчислювальна продуктивність;
- підтримка різних стилів програмування: структурне, об'єктно-орієнтоване, узагальнене програмування, функціональне програмування, мета-програмування, що породжує;
- автоматичний виклик деструкторів об'єктів (у порядку зворотному виклику конструкторів) спрощує та підвищує надійність управління пам'яттю та іншими ресурсами;
- перевантаження операторів;
- шаблони (дають можливість побудови узагальнених контейнерів та алгоритмів для різних типів даних);
- можливість розширення мови для підтримки парадигм, які не підтримуються компіляторами;
- доступність - існує безліч навчальної літератури.

2.3 Опис програми із захисту виконуваних файлів

Отже створена програма-протектор бере пачку інструкцій, видокремлює стекову віртуальну машину та змінює у ній опкоди на

мутовані, виконує їх, та повертає результат в оригінальний стек програми. Після цього протектор мутує дані регістрів. Далі він проводить XOR - операцію кожного байту секції коду на певні значення в файлі, і замінюємо точку входу виконуваного файлу на мутовану, але вона розшифровує сама себе. Виняткова диз'юнкція, також операція XOR , додавання за модулем два — логічна та бітова операція, що набуває значення «істина» тоді й лише тоді, коли значення «істина» має суто один з її операндів. XOR виконується з двома бітами (a та b). Результат виконання операції XOR (що виключає АБО) дорівнює 1, коли один з бітів b або a дорівнює 1. В інших ситуаціях результат застосування оператора XOR дорівнює 0. Протектор створює повну копію програми, але з обфускованним кодом. Візуальної різниці зовсім немає. Але код виконуваного файлу стає складним для аналізу та обратної інженерії, майже неможливим. Протектор захищає програми постійно. Дамп пам'яті програми на диск з наступним запуском не зніме захист. Також він ховає точку входу до бібліотеки або програми, шифрує її, але вона дешифрує сама себе при виклику, щоб запустити файл. Також протектор ховає інформацію про функції Windows API, які викликають вашу програму. Інформація про бібліотеки, що потрібні для запуску, також стають недоступними для зловмисника. Весь код нашого протектору, який зв'язується з виконуваним файлом, піддається мутації та розмазуванню. Щоразу цей код генерується наново, використовуючи техніку віртуалізації. Розроблений протектор не дозволяє зовнішнім програмам відстежувати звернення захищеної програми до файлів або реєстру. Через те, що накладення захисту призводить до досить сильного збільшення розміру файлу на диску, для зменшення цього розміру є підтримка стиснення програми (з вбудованим кодом захисту).

Вміст форм (фактично *.dfm файл) не можна буде вилучити як з *.exe, так і під час виконання програми. Технологія CodeReplace витягує частини програми, вставляє замість них код для «сміття», щоб змішати оригінальний код з мутованним кодом захисту і зберігає його в іншому місці програми. Після запуску ділянки коду, захищеного CodeReplace, після численних

перевірок він виймається з контейнеру та розшифровується. Після виконання знову заповнюється пустими командами, що слугують для візуального заповнення кода з метою заплутати зловмисника при аналізі коду, щоб зробити розуміння алгоритму та логіки роботи програми майже неможливим. Протектор може сам проаналізувати вашу програму і вибрати здається йому найбільш доцільними для такого захисту функції.

Віртуальна машина розробленого протектору – один із найсильніших способів захисту програми. Суть його полягає в тому, що частини коду виконуваного файлу замінюються згенерованим асемблерним кодом, але не рідним для Intel процесорів, а кодом абстрактного віртуального процесора зі своєю системою команд і внутрішньою структурою. Протектор щоразу випадково генерує план його архітектури. Таким чином, код захисту стає складно аналізувати та зрозуміти взагалі.

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Розробка тестової програми

Для протектору було розроблено простий та лаконічний дизайн. Програма містить дві кнопки. Choose file – ми обираємо виконуваний файл, який хочемо «накрити» нашим протектором. Після вибору файлу протектор відображає характеристики цього файлу, які він бере з метаданих. Кнопка Protect виконує основну роботу протектору і ми можемо одразу зберегти обфускований виконуваний файл програми із будь-якою назвою, навіть такою самою.

На рисунку 3.1 представлено реалізацію інтерфейсу розробленого протектору:

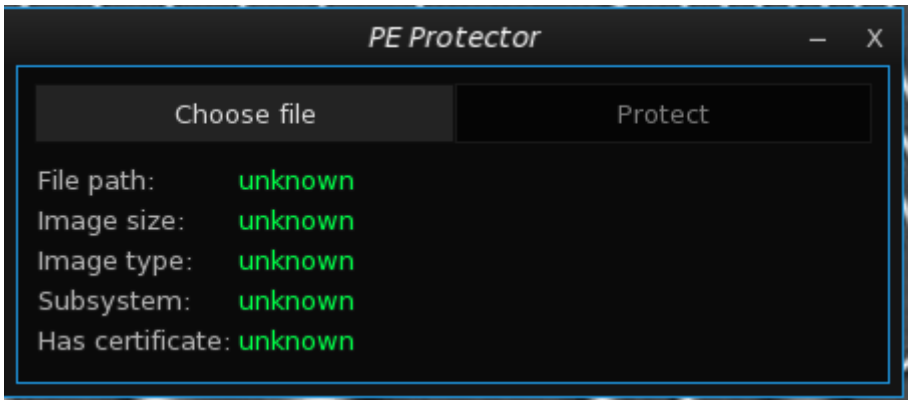


Рисунок 3.1 – Інтерфейс програми

На рисунку 3.2 представлено вигляд програми після вибору файлу, який потрібно захистити:

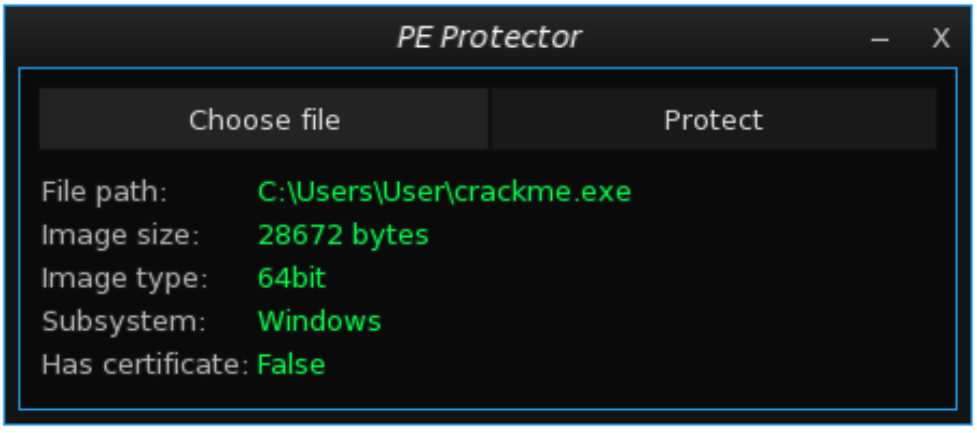


Рисунок 3.2 – Інтерфейс програми після вибору файлу

На рисунку 3.3 представлено роботу програми після натиснення на кнопку Protect:

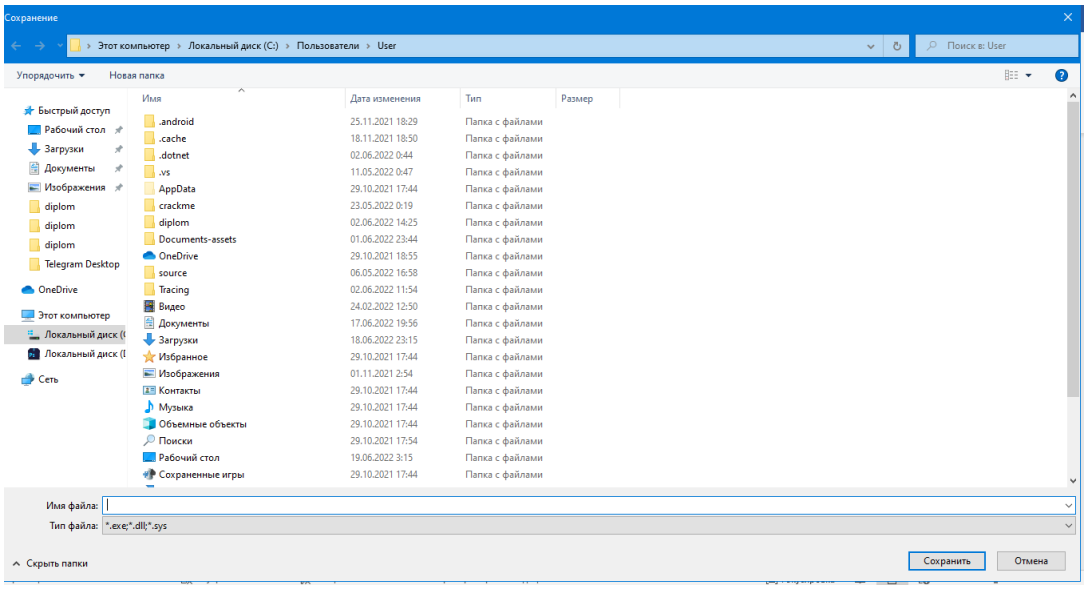


Рисунок 3.3 – Робота кнопки Protect

Щоб наглядно побачити обфускацію коду я пропущу файл до накриття через дебагер x64dbg та файл «накритий» розробленим протектором також пропущу через дебагер. Відкривши асемблерні інструкції можна побачити їх обфускацію (представлено на рисунку 3.4 та на рисунку 3.5):

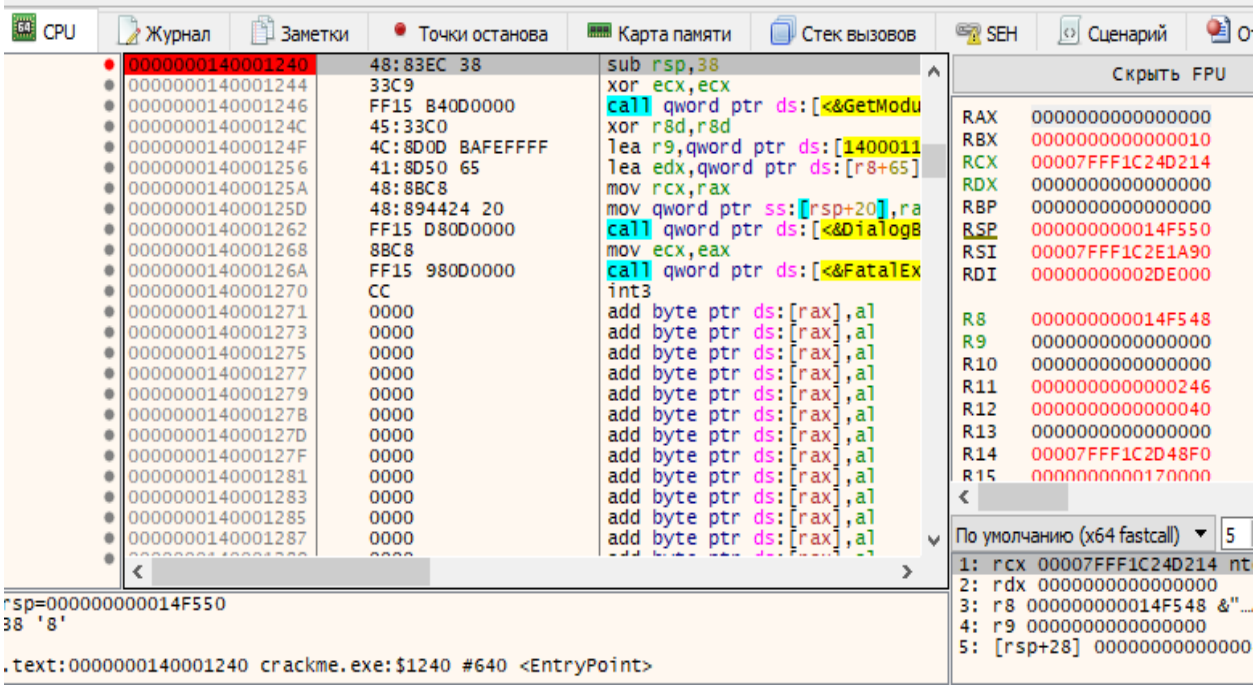


Рисунок 3.4 – Асемблерні інструкції файлу до накриття

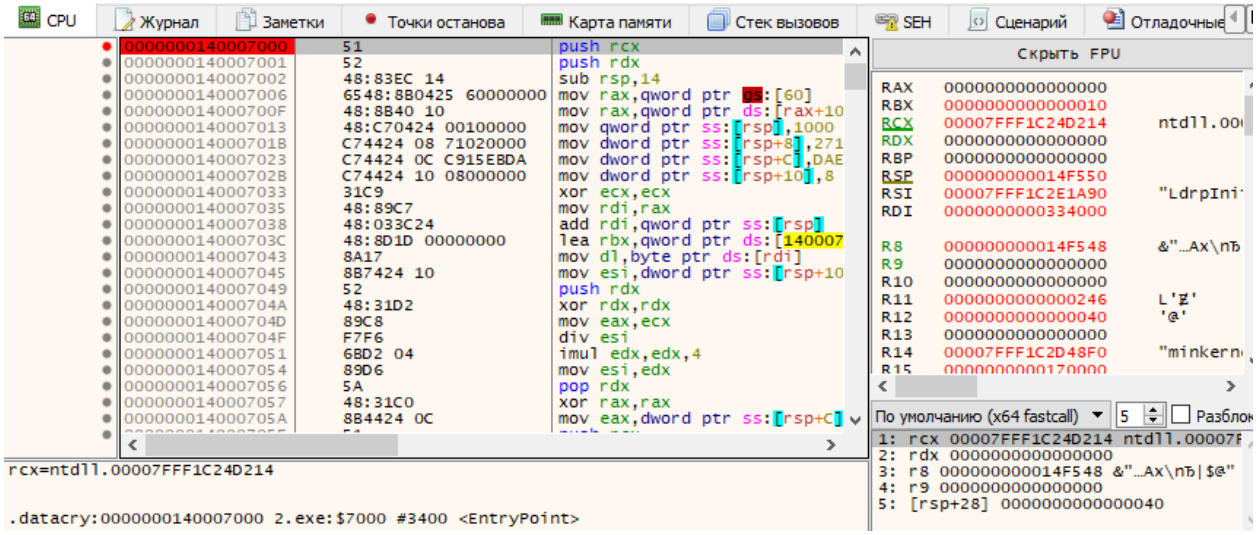


Рисунок 3.5 – Асемблерні інструкції файлу після накриття

3.2. Розробка патчера

Щоб продемонструвати роботу протектору, що він дійсно захищає виконувані файли від несанаційного використання було взято програму-активатор та написано патчер в якості зловмисника для її злому. В нашому випадку програма-патчер виступає в якості зловмисника, який буде робити несанаційне використання виконуваного файлу, спочатку вразливої програми, а потім вже захищеної програми. На рисунку 3.6 представлено написаний алгоритм роботи патчеру:

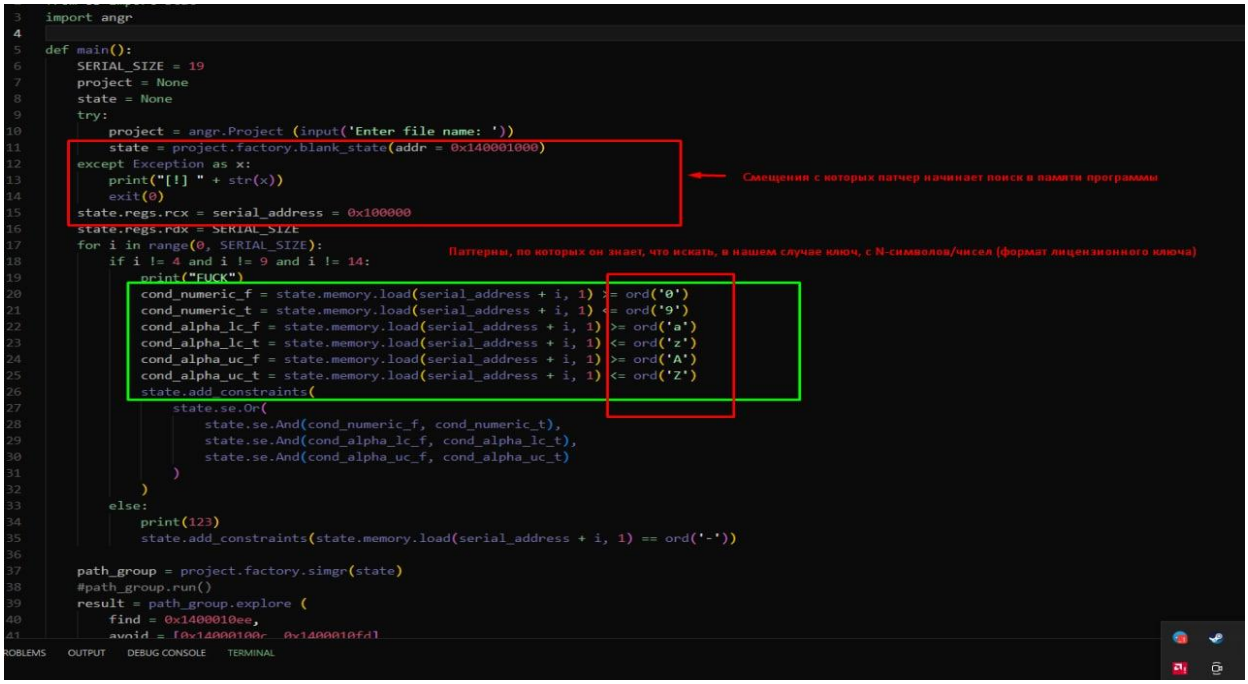


Рисунок 3.6 – Код патчера

3.3. Тестування розробленої програми

Для початку ми запускаємо програму-активатор через написаний патчер, тобто через нашого умовного зловмисника, який аналізує асемблерні інструкції виконуваного файлу та шукає в його пам'яті функцію, де відбувається порівняння ключа та витягує його значення з котрим вразлива програма порівнювала те, що вводив користувач. В результаті патчер (зловмисник) отримує ключ, який підходить для активації програми.

На рисунку 3.7 представлено роботу патчера з вразливим файлом та отриманий ним результат:

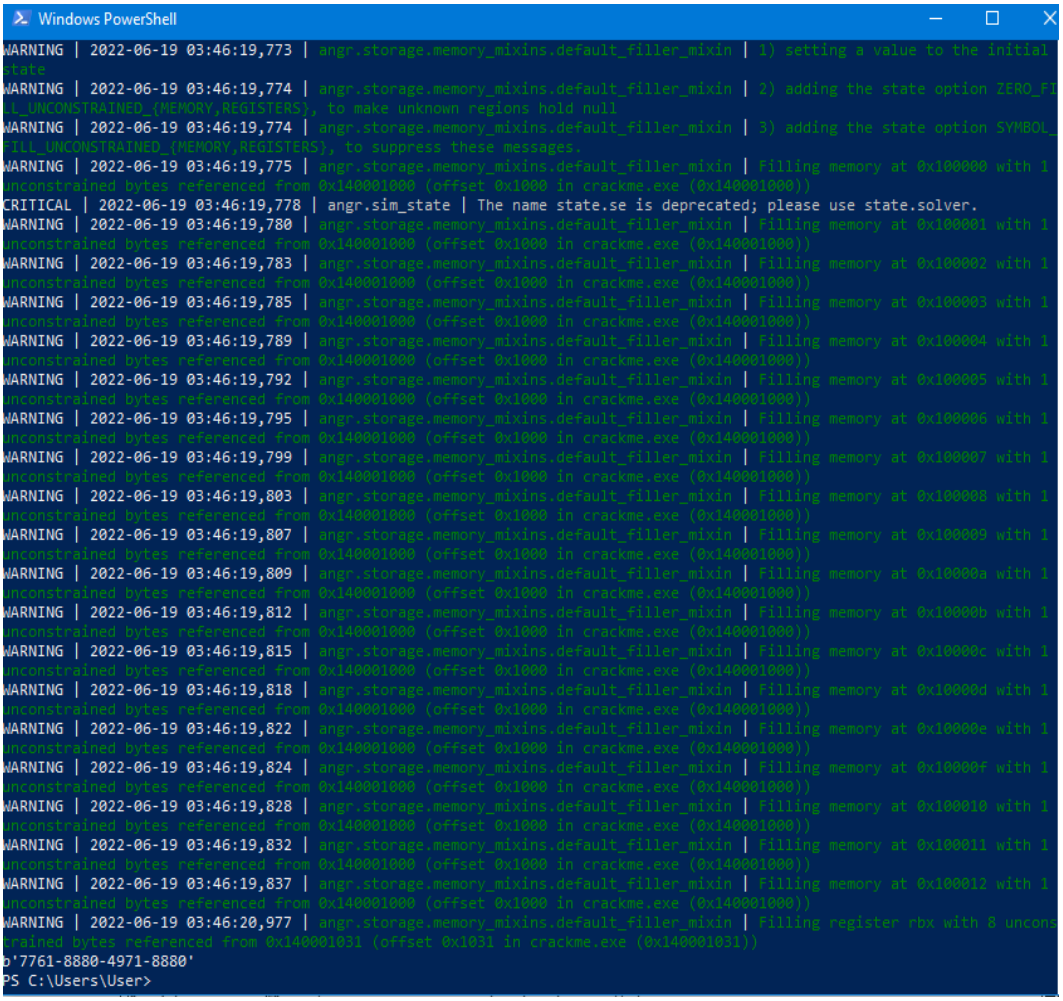


Рисунок 3.7 – Робота патчера з вразливим файлом

На рисунку 3.8 представлено, що отриманий ключ підходить для активації програми:

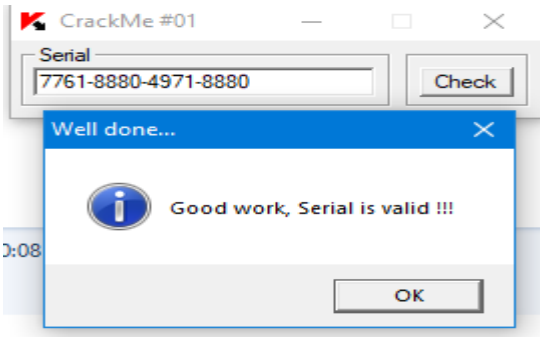


Рисунок 3.8 – Перевірка отриманого ключа

На рисунку 3.9 представлено роботу патчера з накритим нашим протектором файлом (тобто захищенним) та отриманний ним результат:

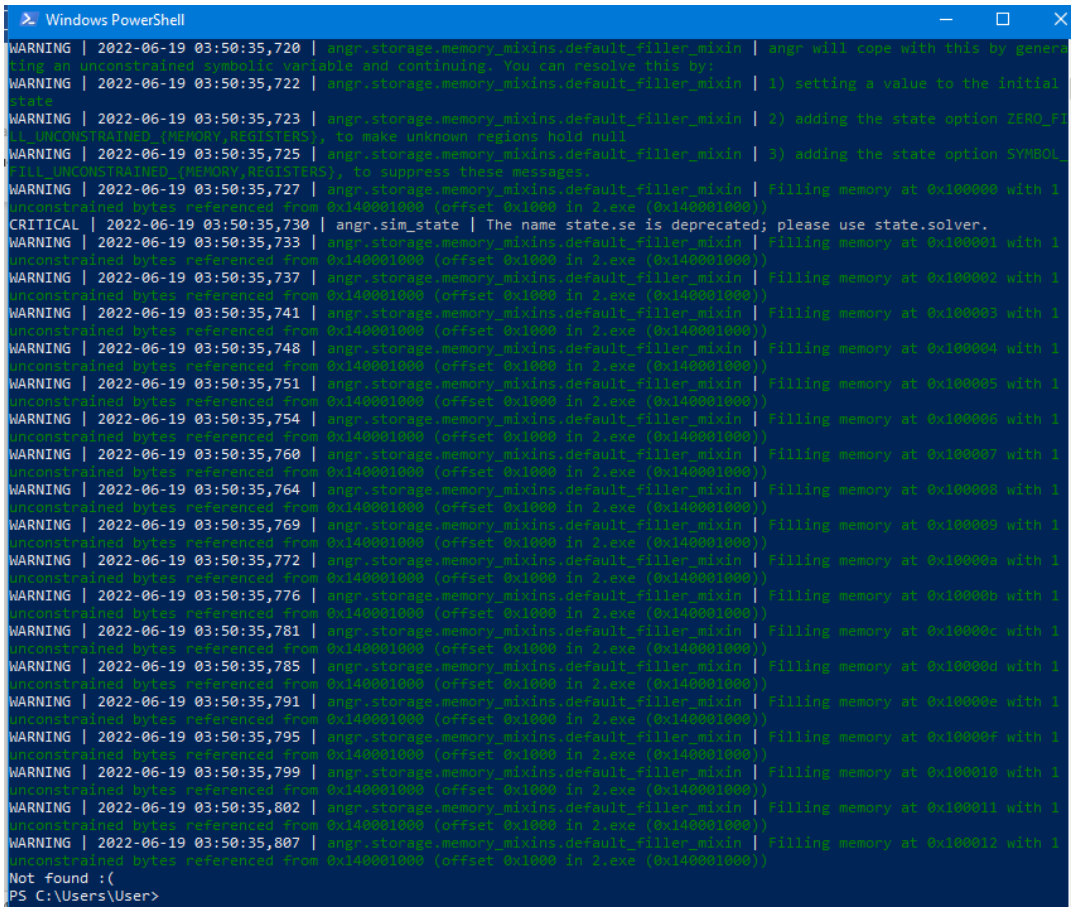


Рисунок 3.9 – Робота патчера з захищеним файлом

Як результат ми бачимо, що наш патчер не зміг дізнатись ключ. Через віртуальні функції він не знав за що зачіпитись та не знайшов адресу функції, де була реалізована логіка порівняння ключів, та де знаходився справжній ключ до програми-активатора. Тобто наш протектор зміг зробити виконуваний файл захищеним від несанкціонованого втручання.

4 ОХОРОНА ПРАЦІ

Питання охорони праці людини необхідно вирішувати на всіх стадіях трудового процесу незалежно від виду професійної діяльності.

Забезпечення безпечних і здорових умов праці в значній мірі залежить від правильної оцінки небезпечних, шкідливих виробничих факторів. Однакові за складністю зміни в організмі людини можуть бути викликані рядом різних причин. Це можуть бути фактори виробничого середовища, або ж надмірне фізичне та розумове навантаження, нервово-емоційна напруга, або ж сполучення цих факторів.

У даному розділі вирішується питання охорони праці програмного розробника на стадії розробки ним програми захисту власного продукті від несанкціонованого доступу.

Об'єктом проведення досліджень було взяте приміщення, в якому працює програмний розробник, має загальну площу 12 м^2, висоту стелі 3 м. На стелі розташовано одну люмінесцентну лампу білого світла зі значеннями світлового потоку по 1200 лм. У приміщенні знаходиться 1 робоче місце з ноутбуком. Робоче місце обладнане робочим столом площею 1 м^2, комп’ютерним стільцем та ноутбуком, зарядним пристроєм, миші.

Головними елементами робочого місця програмного розробника виступають стіл та крісло. Основне робоче положення – положення сидячі.

Робоче місце програмного розробника складається з: ноутбуку, зарядного пристрою для ноутбуку, бездротової миші, комп’ютерного стільця. Біля робочого місця знаходиться мережевий фільтр з захистом від перевантажень на 6 розеток по 220В кожна. В приміщенні знаходився вогнегасник та план евакуації на випадок пожежі.

Під час аналізу умов праці необхідно визначити небезпечність шкідливих факторів за нормативним документом ГН 3.3.5-8.6.6.1 2002 «Гігієнічна класифікація праці за показниками шкідливості та небезпечності факторів виробничого середовища, важкості та напруженості трудового

процесу». Було проведено аналіз на наступні фізичні фактори: мікроклімат робочої зони, шум на робочому місці, освітленість робочої зони, електробезпека, ергономічні аспекти під час роботи.

Мікроклімат. Фактична температура повітря 23 - 24°С. Відносна вологість у приміщенні 45 – 50%. Швидкість руху повітря 0.2 м/с. Дані параметри повністю задовольняють вимоги для категорії легких робіт Іб, до якої належить робота програміста, згідно з ДСН 3.3.6.042-99.

Фактичне освітленість робочого місця становить 600лк - задовольняє мінімальним умовам. Нормованим параметром природного освітлення згідно ДБН В.2.5–28 – 2006 є коефіцієнт природного освітлення (КПО). КПО встановлюється в залежності від розряду виконуваних зорових робіт. Робота програміста відноситься до робіт середньої точності (IV розряд зорових робіт, мінімальний розмір об'єкту розрізнення складає 0,5-1,0мм), для яких при використанні бокового освітлення КПО=1,5%. Для штучного освітлення нормованим параметром виступає $E_{мін}$ – мінімальний рівень освітленості, та Кп – коефіцієнт пульсації світлового потоку, який не повинний бути більшим ніж 20%. Мінімальна освітленість встановлюється в залежності від розряду виконуваних зорових робіт. Для IV розряду зорових робіт вона дорівнює 300-500 лк.

Вплив шуму на програмного розробника. Через вплив шуму знижується концентрація уваги та порушуються фізіологічні функції. Через це з'являється втома у зв'язку з підвищеними енергетичними витратами та нервово-психічним напруженням. Також погіршується мовна комутація. Якщо ж мова йде про вплив шуму, то в такому випадку основну увагу приділяють стану органу слуху, тому що слуховий аналізатор у першу чергу сприймає звукові коливання, його подразнення від дії шуму. Навіть невеликої інтенсивності шуми, які присутні протягом усього робочого дня впливають негативно на організм людини. Допустимий рівень шуму для роботи програміста має не перевищувати 50 дБ за ДСН 3.3.6.037-99 «Санітарні норми виробничого шуму, ультразвуку та інфразвуку». Нормовані

рівні шуму забезпечуються шляхом використання малошумного устаткування, застосуванням звуковбирних матеріалів для облицювання приміщень. Також задля захисту від небажаного шуму використовують навушники.

Ергономічні аспекти. Ергономічними аспектами проектування відеотермінальних робочих місць є: висота робочої поверхні, розміри простору для ніг, вимоги до розташування документів на робочому місці (наявність і розміри підставки для документів, можливість різного розміщення документів, відстань від очей користувача до екрану, документа, клавіатури і т.д.), характеристики робочого крісла, регульованість елементів робочого місця.

Робоча поза сидячи у програміста викликає мінімальне стомлення. Робоче місце передбачає чіткий порядок і сталість розміщення предметів, засобів праці і документації. Раціональне планування на робочому місці: те, що потрібно для виконання робіт частіше, розташоване в зоні легкої досяжності робочого простору.

Робочий стіл повинен мати стабільну та зручну конструкцію. Його розміри 160 х 95 см. Крісло й площа стола повинні регулюватися по висоті на 40 – 50 см і 60 – 80 см відповідно. Тип робочого крісла вибрано залежно від тривалості роботи, для тривалої – масивне комп’ютерне крісло.

При роботі з персональним комп'ютером дуже важливу роль грає дотримання правильного режиму праці та відпочинку. Тому що в іншому випадку у працівника наростає значна напруга зорового апарату та з’являються скарги на незадоволеність роботою, головні болі, дратівливість, порушення сну, втому та хворобливі відчуття в очах, в поясниці, в області шиї та руках. Робочий час та перерви на відпочинок виконувались відповідно до СанПіН 2.2.2 542-96 «Гігієнічні вимоги до відеодисплейних терміналів, персональних електронно-обчислювальних машин і організації робіт». Денний робочий час становив 6 годин, 70 хвилин – загальний час перерв. Під

час перерв виконувалась виробнича гімнастика з двох-трьох вправ: потягування з глибоким диханням, обертання тулуба, присідання тощо.

Електробезпека. Приміщення за небезпекою ураження електричним струмом можна віднести до 1 класу, тобто це приміщення без підвищеної небезпеки (сухе, без пилу, з нормальною температурою повітря, ізольованими підлогами і малим числом заземлених приладів).

На робочому місці програмного розробника немає металевих деталей, корпус ноутбуку виконаний із пластику, деталі самого ноутбуку щільно закриті всередині та не мають прямого доступу до них.

Основні причини ураження людини електричним струмом на робочому місці:

- дотик до металевих неструмоведучих частин (внутрішніх деталей ноутбуку), що можуть виявитися під напругою в результаті ушкодження ізоляції та у разі пошкодження корпусу ноутбука;
- нерегламентоване використання електричних приладів.

Пожежна безпека являє собою комплекс організаційних заходів щодо забезпечення нормального протипожежного стану об’єкта нерухомості — житлового, виробничого, складського, офісного або торгово-розважальної споруди, приміщення або ж комплексу приміщень.

Основним законодавчим документом, що регламентує вимоги щодо пожежної безпеки, є Закон України "Про пожежну безпеку", згідно з яким необхідно виконувати огляд приміщень, профілактики пожежі, правильну оцінку небезпеки виникнення пожежі в будинку, визначення небезпечних факторів і обґрунтування способів і засобів пожежопопередження і захисту.

До виникнення пожеж, в приміщеннях пов’язаних із програмною розробкою, найчастіше призводять:

- куріння;
- порушення правил користування електроприладами;
- перенавантаження електромережі;
- необережне поводження з вогнем в приміщенні.

Комп’ютерне обладнання повинне підключатися до електромережі лише за допомогою справних штепсельних з’єднань і електророзеток заводського виготовлення.

У штепсельних з’єднаннях та електророзетках, крім контактів фазового та нульового робочого провідників, мають бути спеціальні контакти для підключення нульового захисного провідника. Їх конструкція має бути такою, щоб приєднання нульового захисного провідника відбувалося раніше, ніж приєднання фазового та нульового робочого провідників. Порядок роз’єднання при відключенні має бути зворотним для уникнення порушень правил пожежної безпеки, ліквідації пожеж і загорянь.

Основними завданнями пожежної безпеки є: контроль за дотриманням протипожежних вимог, запобігання пожеж і нещасних випадків від них, гасіння пожеж, рятування людей і надання допомоги в ліквідуванні наслідків аварій, катастроф і стихійного лиха.

Для протидії пожежі треба: наявність засобів пожежогасіння та пожежної сигналізації; створити план та шляхи евакуації, фарбувати стіни приміщення вогнезахисною фарбою світлого кольору, яка у випадку пожежі витримує горіння полум'я протягом 1 - 1,5 годин.

У разі виникнення пожежі необхідно:

- Повідомити про пожежу директора організації або іншу компетентну посадову особу;
- Повідомити пожежно-рятувальну службу, вказати адресу організації, кількість поверхів будівлі, поверх, на якому виникла пожежа та своє прізвище.
- Завершити усю робочу діяльність, крім тієї, що пов’язана із заходами щодо ліквідування пожежі.
- Вжити заходи щодо евакуації людей та збереження матеріальних цінностей, гасіння пожежі з використанням вогнегасників та інших наявних засобів пожежогасіння.
- Здійснити відключення електроенергії (за винятком систем

протипожежного захисту).

- Організувати зустріч пожежно-рятувальних підрозділів, забезпечити безперешкодний доступ їх до місця виникнення пожежі та надати їм допомогу під час локалізації та ліквідації пожежі.

У даному розділі кваліфікаційної роботи були викладені вимоги до робочого місця програмного розробника. Створені умови повинні забезпечувати комфортну роботу. Були зазначені оптимальні розміри робочого столу та робочої поверхні. Дотримання умов, що визначають оптимальну організацію робочого місця програмного розробника, дозволить зберегти гарну працездатність протягом усього робочого дня, а й підвищить продуктивність праці програмного розробника , що в свою чергу впине на швидкість розробки продукту і його налагодженню.

ВИСНОВКИ

Піратство програмного забезпечення є однією з основних проблем для розробників. Для захисту програмного забезпечення від незаконного розповсюдження недостатньо ліцензії, необхідно захистити програму від зворотної інженерії. Існують сотні способів захисту виконуваних файлів, однак вони також мають свій обхід. Ми створили програму-протектор мовою C++ під операційну систему Windows. Переважно цей протектор було розроблено для захисту власного програмного продукту, щоб захистити його від неасоційованого використання. Цей протектор є конкурентоспроможною програмою, так як у відкритому доступі не має подібних програм, тільки у платному сегменті. Перевагою розробленого протектору є те, що він має власну віртуальну машину, метод обфускації та унікальну ентропію, яка не схожа навіть на платні аналоги. Також перевагою є те, що для вже давно існуючих аналогів – зловмисники навчились відновлювати інструкції, а щоб виконати те саме, після накриття новим розробленим протектором з нуля – потрібно дуже багато часу.

ЛІТЕРАТУРА

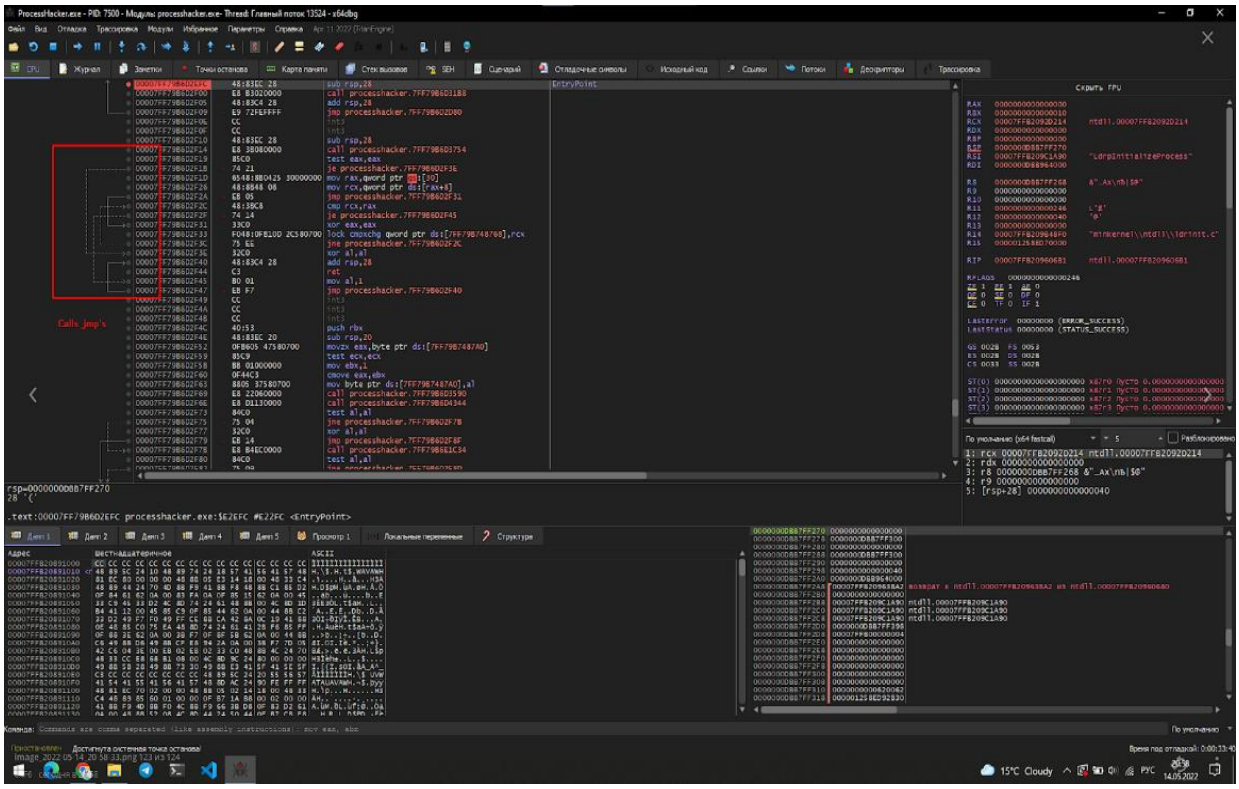
1. Обфускація. *Матеріал з вікіпедії – вільної енциклопедії.* URL: https://uk.wikipedia.org/wiki/Обфускація
2. Agrachiv. Обфускація як метод захисту програмного забезпечення. URL: https://habr.com/ru/post/533954/
3. Web-proger. C++. URL: http://web.spt42.ru/index.php/chto-takoe-c-plus-plus
4. Виконуюча диз'юнкція. *Матеріал з вікіпедії – вільної енциклопедії.* URL: https://uk.wikipedia.org/wiki/Виключна_диз%27юнкція
5. Корисні поради. Виконувані файли мають розширення якого типу? URL: http://xn--80aimveh.pp.ua/kompyuter-internet/5274-vikonuvan-fayli-mayut-rozshirennya-yakogo-tipu-nayblsh-poshiren.html
6. Виконуваний файл. *Матеріал з вікіпедії – вільної енциклопедії.* URL: https://uk.wikipedia.org/wiki/ Виконуваний_файл

Додаток А. Таблиця істинності функції XOR

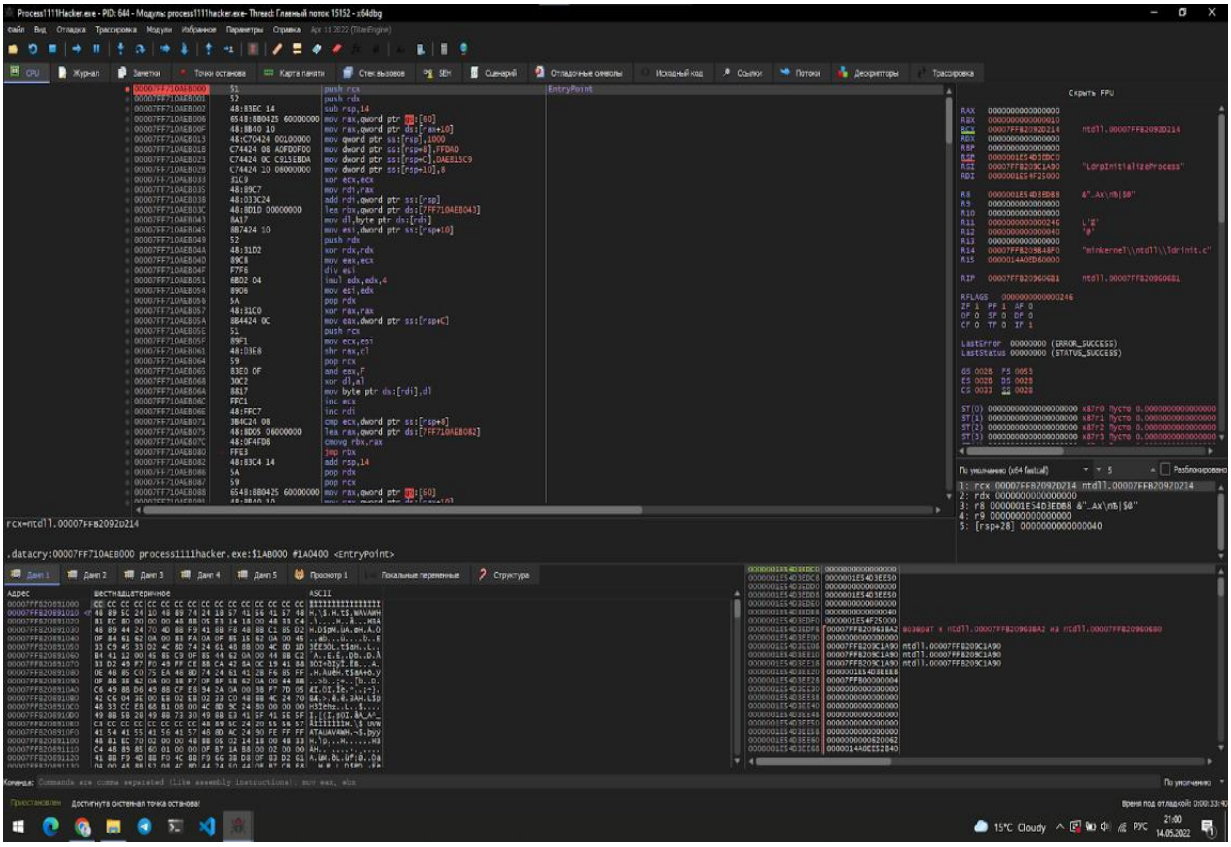
A	B	$A \oplus B$
хибність	хибність	**хибність**
хибність	істина	**істина**
істина	хибність	**істина**
істина	істина	**хибність**

Додаток Б. Точка входу виконуваного файлу та точка входу з застосуванням мутації коду

Точка входу без мутації:



Точка входу з мутацією:



Додаток В. Програмний код інтерфейсу протектору

```
#include <iostream>
#include <nfd.h>
#include "window.h"
#include "pe.h"

std::string GetPath(const char* filter, bool is_open);

int main() {
  std::locale::global(std::locale{"ru"});

  constexpr uint32_t width = 450;
  constexpr uint32_t height = 195;

  Window wnd{{width, height}, "PE Protector"};

  const char* none_name = "unknown";
  const char* pe_xtensions = "exe,dll,sys";

  tgui::Color info_color{0, 255, 72};
  tgui::Color name_color{180, 180, 180};
```

```
std::unique_ptr<PE> current_pe;

auto chbtn = wnd.CreateButton({0, 0}, {(width / 2 - 15), 30}, "Choose file");
auto svbtn =
    wnd.CreateButton({(width / 2 - 15), 0}, {width / 2, 30}, "Protect");
auto fp = wnd.CreateLabel({100, 40}, none_name, info_color);
auto is = wnd.CreateLabel({100, 60}, none_name, info_color);
auto it = wnd.CreateLabel({100, 80}, none_name, info_color);
auto ss = wnd.CreateLabel({100, 100}, none_name, info_color);
auto hc = wnd.CreateLabel({100, 120}, none_name, info_color);

wnd.CreateLabel({0, 40}, "File path:", name_color);
wnd.CreateLabel({0, 60}, "Image size:", name_color);
wnd.CreateLabel({0, 80}, "Image type:", name_color);
wnd.CreateLabel({0, 100}, "Subsystem:", name_color);
wnd.CreateLabel({0, 120}, "Has certificate:", name_color);

svbtn->setEnabled(false);

chbtn->onPress([&]() {
  if (auto path = GetPath(pe_xtensions, true); !path.empty()) {
    if (PE pe{path}; pe.IsOpen()) {
```

```
        current_pe = std::make_unique<PE>(path);


        fp->setText(path);
        is->setText(std::to_string(current_pe->GetImageSize()) + " bytes");
        it->setText(std::string{current_pe->IsImage64() ? "64" : "32"} + "bit");
        ss->setText(current_pe->GetSubsystemName());
        hc->setText(current_pe->HasCert() ? "True" : "False");


        svbtn->setEnabled(true);
      } else {
        wnd.DrawMessageBox("Error", pe.open_error(), true);
      }
    }
  });
  svbtn->onPress([&]() {
    if (auto path = GetPath(pe_xtensions, false); !path.empty()) {
      current_pe->ProtectAndSave(path);
    }
  });
  wnd.Display();
  return 0;
}
```

```
std::string GetPath(const char* filter, bool is_open) {
  std::string spath{};

  nfdchar_t* path;
  nfdresult_t result;

  result = is_open ? NFD_OpenDialog(filter, nullptr, &path)
                   : NFD_SaveDialog(filter, nullptr, &path);
  if (result == NFD_OKAY) {
    spath = path;
    free(path);
  }
  return spath;
}
```

Додаток Г. Програмний код алгоритму роботи протектора та патчера

```
#include "pe.h"
#include <fstream>
using namespace asmjit::x86;
PE::PE(std::string_view path) {
  if (std::ifstream file{path.data(), std::ios::binary}; file.is_open()) {
    try {
      pe_base_ = std::make_unique<pe_bliss::pe_base>(
          pe_bliss::pe_factory::create_pe(file));
      file.close();
      if (pe_base_->is_dotnet()) {
        open_error_ = "File is not native";
        return;
      }
      if (pe_bliss::entropy_calculator::calculate_entropy(*pe_base_) > 6.8) {
        open_error_ = "File is packed";
        return;
      }
    } catch (const pe_bliss::pe_exception& e) {
      open_error_ = e.what();
      file.close();
    }
```

```
    return;
  }
  open_error_ = "Could not read file";
}
bool PE::IsOpen() const noexcept { return open_error_.empty(); }

bool PE::IsImage64() const {
  return pe_base_->get_pe_type() == pe_bliss::pe_type_64;
}
bool PE::IsDll() const {
  return pe_base_->get_characteristics() & pe_bliss::pe_win::image_file_dll;
}
bool PE::HasCert() const { return pe_base_->has_security(); }
std::size_t PE::GetImageSize() const {
  return pe_base_->get_size_of_image(); }
std::string PE::GetSubsystemName() const {
  return pe_base_->is_gui() ? "Windows" : "Console";
}
std::string PE::open_error() const { return open_error_; }
#include <iostream>
void PE::ProtectAndSave(std::string_view path) {
  asmjit::JitRuntime rt{};
```

```
asmjit::CodeHolder code{};
code.init(rt.environment());
Assembler shell{&code};
if (std::ofstream file{path.data(), std::ios::binary}; file.is_open()) {
  for (auto& section : pe_base_->get_image_sections()) {
    if (section.executable()) {
      section.writeable(true);
      CryptSection(section);
      AddDecryptCode(section, shell);
    }
  }
  pe_bliss::section sect{};
  sect.set_name(".datacry");
  sect.executable(true);
  if (IsDll() && pe_base_->has_tls()) {
    auto tls_info = pe_bliss::get_tls_info(*pe_base_);
    auto first_tls = tls_info.get_tls_callbacks().front();
    AddJumpToOriginalEP(shell, first_tls);
    auto& datacry = pe_base_->add_section(sect);
  } else {
    AddJumpToOriginalEP(shell, pe_base_->get_ep());
    sect.set_raw_data(GetRawCode(code));
```

```
      pe_base_->set_ep(pe_base_->add_section(sect).get_virtual_address());
    }
    pe_bliss::rebuild_pe(*pe_base_, file);
    file.close();
  }
}
void PE::CryptSection(pe_bliss::section& section) {
  auto& data = section.get_raw_data();
  for (size_t i = 0; i < data.size(); ++i)
    data[i] ^= (GetKey() >> ((i % GetKeySize()) * 4)) & 0xF;
}
void PE::PushImageBase(Assembler& shell) {
  if (IsDll()) {
    shell.mov(rax, rcx);
  } else {
    auto mem = qword_ptr(0x60);
    mem.setSegment(gs);
    shell.mov(rax, mem);
    shell.mov(rax, qword_ptr(rax, 0x10));
  }
}
void PE::AddDecryptCode(const pe_bliss::section& sect, Assembler& shell) {
```

```
if (IsImage64()) {
  shell.push(rcx);
  shell.push(rdx);
  shell.sub(rsp, 20);
  PushImageBase(shell);
  shell.mov(qword_ptr(rsp), sect.get_virtual_address());
  shell.mov(dword_ptr(rsp, 8), sect.get_virtual_size());
  shell.mov(dword_ptr(rsp, 12), GetKey());
  shell.mov(dword_ptr(rsp, 16), GetKeySize());
  shell.xor_(ecx, ecx);
  shell.mov(rdi, rax);
  shell.add(rdi, qword_ptr(rsp));
  shell.lea(rbx, qword_ptr(rip));
  shell.mov(dl, byte_ptr(rdi));
  shell.mov(esi, dword_ptr(rsp, 16));
  shell.push(rdx);
  shell.xor_(rdx, rdx);
  shell.mov(eax, ecx);
  shell.div(esi);
  shell.imul(edx, 4);
  shell.mov(esi, edx);
  shell.pop(rdx);
```

```
    shell.xor_(rax, rax);
    shell.mov(eax, dword_ptr(rsp, 12));
    shell.push(ecx);
    shell.mov(ecx, esi);
    shell.shr(rax, cl);
    shell.pop(ecx);
    shell.and_(eax, 0x0F);
    shell.xor_(dl, al);
    shell.mov(byte_ptr(rdi), dl);
    shell.inc(ecx);
    shell.inc(rdi);
    shell.cmp(ecx, dword_ptr(rsp, 8));
    shell.lea(rax, qword_ptr(rip, 6));
    shell.cmovnle(rbx, rax);
    shell.jmp(rbx);
    shell.add(rsp, 20);
    shell.pop(rdx);
    shell.pop(rcx);
  }
}
void PE::AddJumpToOriginalEP(Assembler& shell, uint32_t ep_rva) {
  PushImageBase(shell);
```

```
  shell.add(rax, ep_rva);
  shell.jmp(rax);
}
std::string PE::GetRawCode(asmjit::CodeHolder& code) {
  std::string shellcode{};
  shellcode.resize(code.codeSize());
  code.copyFlattenedData(shellcode.data(), shellcode.size());
  return shellcode;
}
constexpr uint32_t PE::GetKey() noexcept { return 0xDAEB15C9; }
constexpr uint32_t PE::GetKeySize() noexcept { return 8; }
```

pe.h:

```
#define PE_H_
#include <pe_lib/pe_bliss.h>
#include <asmjit/asmjit.h>
class PE {
 public:
  PE(std::string_view path);
 public:
  bool IsOpen() const noexcept;
  bool IsImage64() const;
  bool IsDll() const;
  bool HasCert() const;
  std::size_t GetImageSize() const;
  std::string GetSubsystemName() const;
  std::string open_error() const;
```

```
 void ProtectAndSave(std::string_view path);
private:
 void CryptSection(pe_bliss::section& section);
 void PushImageBase(asmjit::x86::Assembler& shell);
 void AddDecryptCode(const pe_bliss::section& sect,
                     asmjit::x86::Assembler& shell);
 void AddJumpToOriginalEP(asmjit::x86::Assembler& shell, uint32_t ep_rva);
 std::string GetRawCode(asmjit::CodeHolder& code);
 static constexpr uint32_t GetKey() noexcept;
 static constexpr uint32_t GetKeySize() noexcept;
private:
 std::unique_ptr<pe_bliss::pe_base> pe_base_;
 std::string open_error_;
};
#endif  // !PE_H_
#ifndef WINDOW_H_
#define WINDOW_H_
#include <SFML/Graphics.hpp>
#include <TGUI/TGUI.hpp>
class Window {
public:
 Window(const sf::VideoMode& size, const sf::String& tittle)
     : wnd_size_{size},
       wnd_name_{tittle},
       window_{size, "", sf::Style::None},
       gui_{window_} {}
public:
 void Display();
 tgui::Button::Ptr CreateButton(const sf::Vector2i& pos,
                                const sf::Vector2i& size,
```

```
                        const tgui::String& text);
 tgui::Label::Ptr CreateLabel(const sf::Vector2i& pos,
                        const tgui::String& text,
                        const tgui::Color& color);
 void DrawMessageBox(std::string_view tittle,
                 std::string_view caption,
                 bool is_error);
private:
 void DrawGradientRect(const sf::Vector2f& pos,
                  const sf::Vector2f& size,
                  sf::Color first_color,
                  sf::Color second_color,
                  bool vertical = true);
 void DrawForm(tgui::Button::Ptr& close,
             tgui::Button::Ptr& hide,
             tgui::Panel::Ptr& drag);
 void SetWidgetSizesByRatio(tgui::Widget::Ptr widget,
                      const sf::Vector2i& pos,
                      const sf::Vector2i& size);
 void HandleDragging();
 tgui::Button::Ptr CreateCloseButton();
 tgui::Button::Ptr CreateHideButton();
 tgui::Panel::Ptr CreateDragPanel();
private:
 sf::VideoMode wnd_size_;
 sf::String wnd_name_;
 sf::RenderWindow window_;
 tgui::Gui gui_;
 sf::Vector2i cursor_pos_;
 bool is_dragging_ = false;
```

```
  inline static sf::Vector2f border_size_{8.f, 30.f};
};
#endif  // WINDOW_H_
```

Програмний код патчера

```
from importlib.resources import path
from os import stat
import angr

def main():
	SERIAL_SIZE = 19
	project = None
	state = None
	try:
		project = angr.Project (input('Enter file name: '))
		state = project.factory.blank_state(addr = 0x140001000)
	except Exception as x:
		print("[!] " + str(x))
		exit(0)
	state.regs.rcx = serial_address = 0x100000
	state.regs.rdx = SERIAL_SIZE
	for i in range(0, SERIAL_SIZE):
		if i != 4 and i != 9 and i != 14:
			cond_numeric_f = state.memory.load(serial_address + i, 1) >=
ord('0')
			cond_numeric_t = state.memory.load(serial_address + i, 1) <=
ord('9')
			cond_alpha_lc_f = state.memory.load(serial_address + i, 1) >=
ord('a')
```

```
                cond_alpha_lc_t = state.memory.load(serial_address + i, 1) <=
ord('z')
                cond_alpha_uc_f = state.memory.load(serial_address + i, 1) >=
ord('A')
                cond_alpha_uc_t = state.memory.load(serial_address + i, 1) <=
ord('Z')
                state.add_constraints(
                        state.se.Or(
                                state.se.And(cond_numeric_f, cond_numeric_t),
                                state.se.And(cond_alpha_lc_f, cond_alpha_lc_t),
                                state.se.And(cond_alpha_uc_f, cond_alpha_uc_t)
                        )
                )
        else:
                state.add_constraints(state.memory.load(serial_address + i, 1)
== ord('-'))
    path_group = project.factory.simgr(state)
    #path_group.run()
    result = path_group.explore (
            find = 0x1400010ee,
            avoid = [0x14000100c, 0x1400010fd]
    )
    if result.found:
            solution = path_group.found[0]
            return solution.solver.eval(solution.memory.load(serial_address,
SERIAL_SIZE), cast_to=bytes)
    else:
            return 'Not found :('
if __name__ == "__main__":
   print(main())
```