1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт кибербезопасности и защиты информации

**ЛАБОРАТОРНАЯ РАБОТА № 6**

1. «**Защита программного обеспечения от нелегального использования**»
2. по дисциплине «Основы информационной безопасности»
3. Выполнил
4. студент гр. 4831001/20001 Маронова К.Д.

<*подпись*>

1. Преподаватель
2. асс. преподавателя Пахомов М.А.

<*подпись*>

1. Санкт-Петербург
2. 2023
3. **Цель работы**

Приобрести навыки по защите приложений от нелегального использования, по анализу исполняемых кодов в отсутствие исходных текстов и по применению способов защиты программы от дизассемблирования и отладки.

1. **Постановка задачи**

В данной работе необходимо:

1.Создать простую программу на языке С запрашивающую пароль.

2.Пользуясь исходным текстом программы и утилитой HIEW отыскать в исполняемом файле команду перехода при проверке правильности пароля и изменить ее таким образом, чтобы при вводе любого пароля программа принимала пользователя как авторизованного.

3.Получить и модифицировать код программы Nag.exe таким образом, чтобы сразу после появления всплывающего окна появлялась кнопка «Ок»

4.Распаковать исполняемый файл Guard.exe, модифицировать код таким образом, чтобы наличие отладчика не проверялось и при вводе любого пароля программа принимала пользователя как авторизованного.

5.Для разработанной в начале работы программы создать обфускатор на уровне исходных кодов.

1. **Теоретические исследования**

Простейшим способом проверки легальности копии программы является запрос у пользователя учетных данных (например, пароля, номера лицензии, ключа), выданных ему при покупке. Если пользователем введены корректные данные, функция проверки разрешает продолжить работу с программой.

В общем виде такая схема выглядит следующим образом:

If (!Password() )

{

pError (“Access denied!”);

abort();

}

Данный код транслируется в набор ассемблерных инструкций:

CALL Password

OR AX,AX

JZ continue

PUSH offset str\_access\_denied

CALL pError

…

Continue:

…

Программа продолжает полноценно работать после метки continue. Такой защитный механизм может быть легко обойден заменой одного байта. Заменив условный переход на безусловный, то есть инструкцию JZ на JMP, злоумышленник получает полностью работоспособную копию программы независимо от сложности алгоритма, реализованного в функции Password.

Вторым способом защиты программ от взлома является запрет на открытое хранение пароля во внешнем файле. Таким образом, например, реализовано хранение паролей в операционных системах UNIX. Доступ к файлам паролей не дает никакой информации о самих паролях. Пароли хранятся в зашифрованном виде или в виде соответствующих хэш-значений. Значение алгоритма шифрования или хэширования не дает предположений об исходном пароле.

Простейшая система сокрытия кода программ в целях затруднения ее анализа и модификации – использование логического “исключающего ИЛИ”(XOR). Таблица истинности функции XOR имеет вид:

0 XOR 0 = 1 XOR 1 = 0

0 XOR 1 = 1 XOR 0 = 1

Основное свойство функции: a XOR b XOR a = b, поэтому шифратор и дешифратор кода имеют одно строение.

На языке программирования очень сложно создать что-нибудь устойчивое ко взлому, поэтому в большинстве коммерческих приложений применяется механизм “психологического давления”, направленного на систематическое прерывание работы пользователя, не приобретающего лицензию на авторский продукт.

Пример такой защиты реализация всплывающих окон (nag screen),

которые часто активизируются в ходе сеанса пользовательской работы и напоминающего о необходимости зарегистрировать программу. Окно может содержать поле ввода регистрационных данных, рекламу или просьбу нажать некоторую клавишу. Слово "nag" переводится как "назойливый", что отражает характер воздействия данного прерывания и напоминания, препятствующие работе, подталкивающие к приобретению легальной копии программы, в которой данный механизм деактивирован.

Для обхода механизмов защиты и для анализа поведения программ с целью выявления нарушений применяются программы-дизассемблеры и отладчики (например, IDA, W32Dasm, ollyDBG, SoftICE, Periscope, Bubble Chamber, Sourcer). В простейшем случае, как и при проверке пароля, нарушителю необходимо исправить один байт, что свидетельствует о чрезвычайной ненадежности такого способа защиты программ.

Для того чтобы противодействовать модификации исполняемого файла, разработчики используют проверку целостности программ (например, с помощью контрольного суммирования). Кроме того, существует множество методов по усложнению дизассемблирования и анализа алгоритмов программ.

Простейшим методом от дизассемблирования является применение оптимизирующего компилятора. В оптимизированном исполняемом файле нарушается прямой порядок вызова команд и использования регистров. Это несколько усложняет процесс анализа функций и передачи управления внутри кода программы. В современных компиляторах оптимизация исполняемого кода включена по умолчанию.

Традиционный способ защиты программ применение шифрования или упаковки распространяемого программного обеспечения. В этом случае собственно исполняемый код не соответствует тексту программы в открытом виде. Однако если программа во время исполнения сама себя дешифрует или распаковывает, то она же содержит код, выполняющий действия по дешифрации или распаковке. Причем этот код доступен злоумышленнику, и, следовательно, выполнив его, можно получить в памяти исходный исполняемый код. Поэтому в программах, требующих хорошего уровня защиты от дизассемблирования, используется метод неполного динамического шифрования дешифрования, таким образом, программа никогда не содержится в памяти, расшифрованной целиком.

В большинстве случаев получение исходного ассемблерного кода не дает достаточно информации о механизме защиты. Тогда с помощью специальных утилит-отладчиков можно запустить анализируемую исполняемую программу в режиме отладки. Простейшим методом защиты от запуска в режиме отладки является проверка наличия отладчика (например, функция isDebugPresent).

Существует метод защиты программного обеспечения, основанный на запутывании программного кода, обфускация (оbfuscate сбивать с толку). При этом исходный текст или исполняемый код программы приводится к виду, сохраняющему ее функциональность, но затрудняющему анализ, понимание и модификацию алгоритма при дизассемблировании. Для создания обфусцированного кода могут применяться специализированные компиляторы, использующие неочевидные и недокументированные возможности, или специальные программы-обфускаторы.

При программировании на интерпретируемых скрипт-языках (например, JavaScript, VBScript) пользователю доступен исходный текст программы. Обфускация в таком случае сводится к форматированию текста и замене имен переменных и функций, что направлено на то, чтобы сделать текст менее читаемым, и, соответственно, менее понятным нарушителям.

Обфускация на уровне машинного кода заключается в добавлении блоков с необязательным исполнением, в перемешивании и внесении "мусорных" функциональных блоков программы с целью осложнения анализа код, но не нарушения логики работы программы. Код независимых функциональных блоков вместо тиражирования может, наоборот, объединяться. Таким образом, граф исполнения программы принимает новый вид, в котором передача управления выполняется через новую цепочку и не очевидным образом. В этой связи негативное следствие обфускации-замедление работы программы.

При обфускации машинного кода также используется скрытие констант и данных, когда константа формируется во время исполнения и не встречается в открытом виде. Константные данные могут скрываться при помощи шифрования.

В настоящее время ведутся разработки обфускаторов, использующих виртуальные процессоры. В этом случае создается случайная виртуальная машина со случайным набором инструкций, и весь код программы трансформируется под нее.

Обфускация помогает сделать программу более защищенной от отладки и анализа, но ни один из существующих обфускаторов не гарантирует абсолютной невозможности восстановления логики работы программы.

1. **Описание решения**
2. Работа с паролем

В ходе работы была реализована простая программа, запрашивающая у пользователя пароль. При вводе верного пароля (**parol123)** программа выводит "Password accept, WELCOME" и завершает работу, в противном случае на экран выводится "Password ERROR" и программа завершает работу.

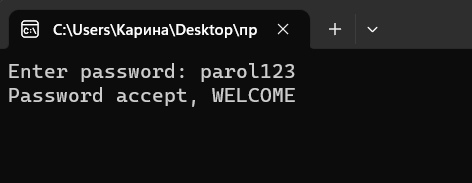


Рисунок 1 – Результат работы программы при вводе верного пароля

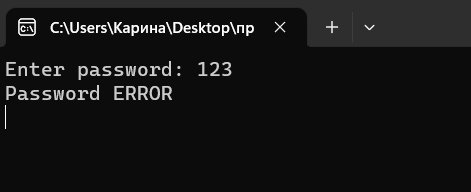


Рисунок 2 – Результат работы программы при вводе неверного пароля

Далее был найден участок дизассемблированного кода, который содержит команды переходов. JZ — условный переход, когда ZF равно 1. Если результат последнего выполненного арифметического или логического оператора равен нулю, то программа перейдет на указанную в инструкции метку.

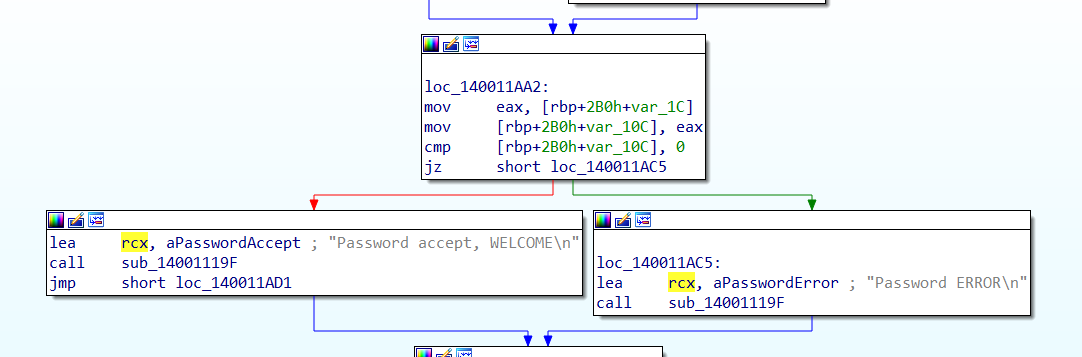


Рисунок 3 – Участок дизассемблированного кода, который содержит команду JZ

После этого заменяем условный переход JZ на безусловный JMP (те переход осуществляется в любом случае) После замены программа будет выводить "Password accept, WELCOME" вне зависимости от правильности пароля.

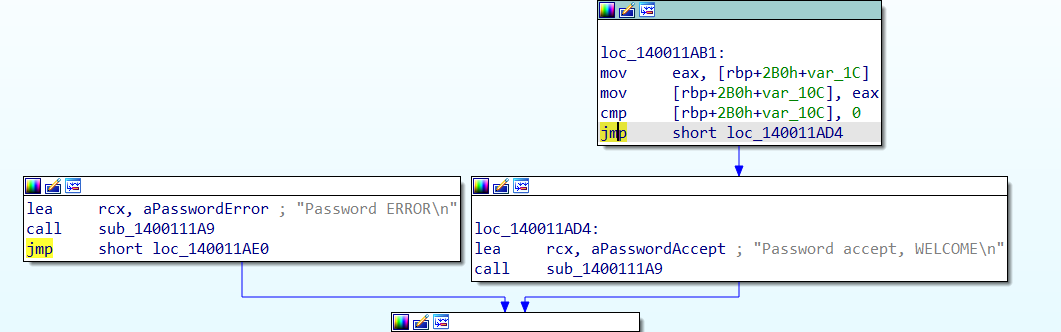


Рисунок 4 – Измененный участок дизассемблированного кода, содержащий команду JMP

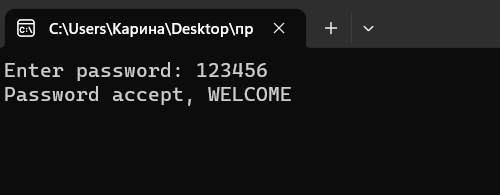


Рисунок 5 – Результаты работы программы после изменения участка дизассемблированного кода

Следующим действием исходная программа была модифицирована: было добавлено шифрование пароля при помощи функции XOR (листинг модифицированной программы в приложении В).

Проделав все вышеописанные действия, оказалось, что данная модификация не дала никаких результатов, при аналогичной замене команд любой набор символов рассматривался, как верный.

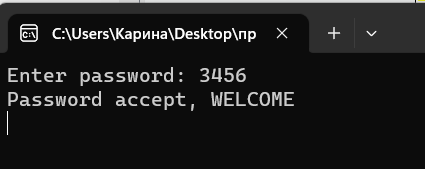


Рисунок 8 – Результаты работы модифицированной программы после замены ассемблерной инструкции JZ на JMP

1. Работа с nag2.exe

В файле Nag.exe после введения текста в течение примерно 10 секунд появляется окно, которое можно закрыть только после стечения счетчика(счетчик на 100 секунд). За это отвечает функция SetTimer.

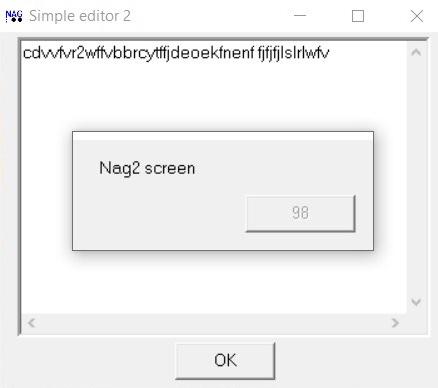


Рисунок 9 – Окно с таймером

Чтобы избежать назойливого появления окна, заменим функцию параметр, который подается в стек через первый push с 0 на 1. Таким образом, функция SetTimer больше не будет запускаться. Поскольку значение 1 не является допустимым адресом в адресном пространстве приложения, то при попытке выполнить функцию SetTimer с параметром TimerProc, равным 1, программа упадет с ошибкой.

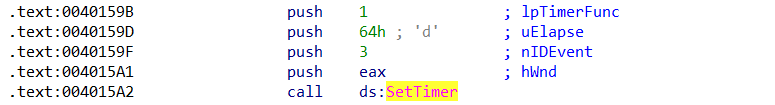


Рисунок 10 – Участок кода содержащий вызов SetTime, в который были внесены изменения

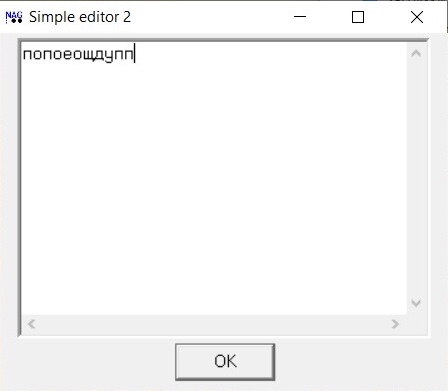


Рисунок 10 – Пример работы программы, не блокируемой счетчиком

1. Работа с Guard.exe

При вводе неверного пароля программа выдает ошибку.

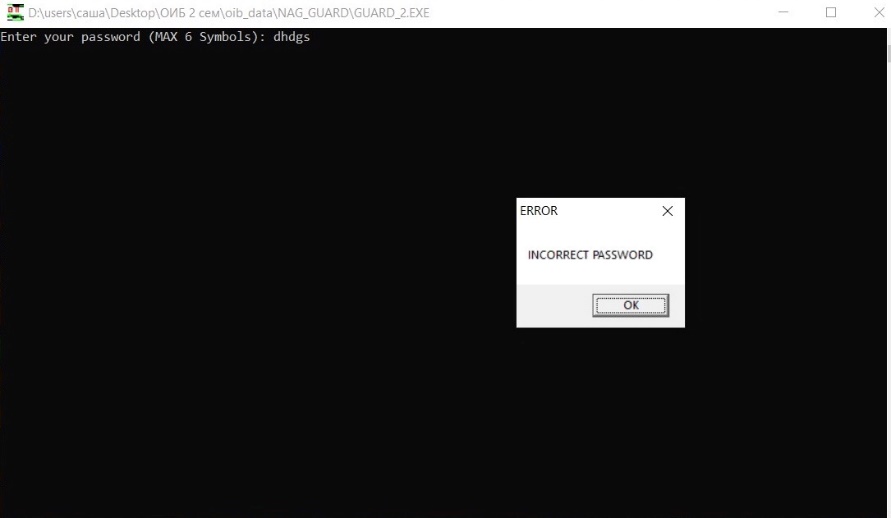


Рисунок 11 – Ошибка при вводе неверного пароля

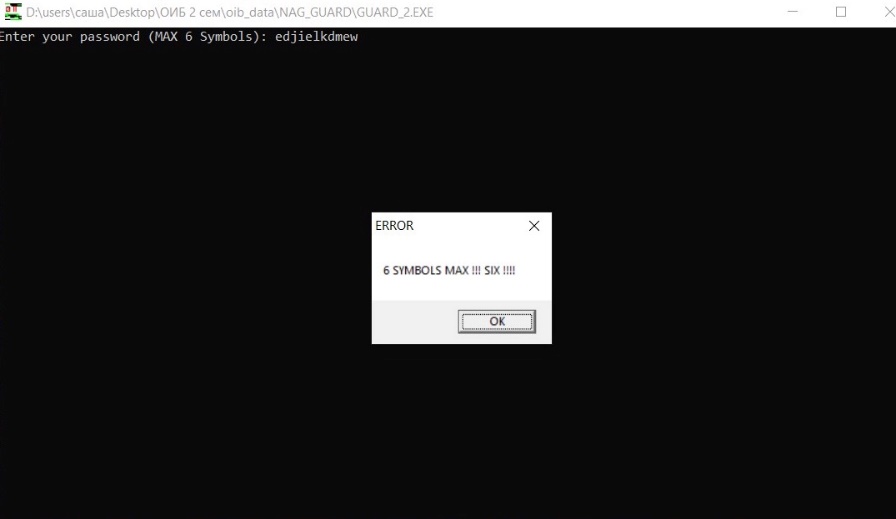


Рисунок 12– Ошибка при вводе более 6 символов

Далее при помощи утилиты Peid был определен упаковщик – UPX. Тогда используем сам UPX для ее распаковки.

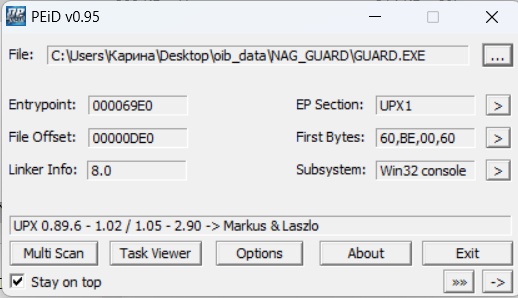


Рисунок 13– Определение упаковщика

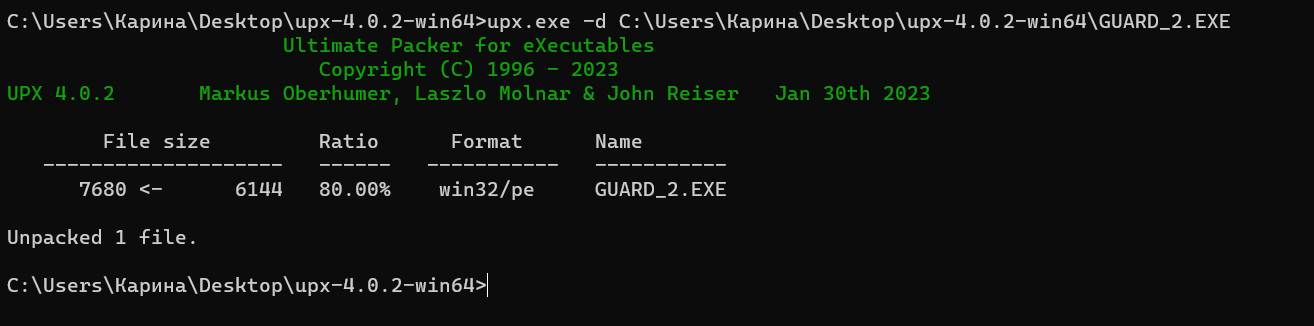


Рисунок 14– Процесс распаковки

Далее нужно заменить ряд ассемблерных команд для того чтобы программа принимала любой пароль. Заменяем переход JNZ(переход, если содержимое аккумулятора не равно нулю) на переход JZ(переход, если содержимое аккумулятора равно 0).

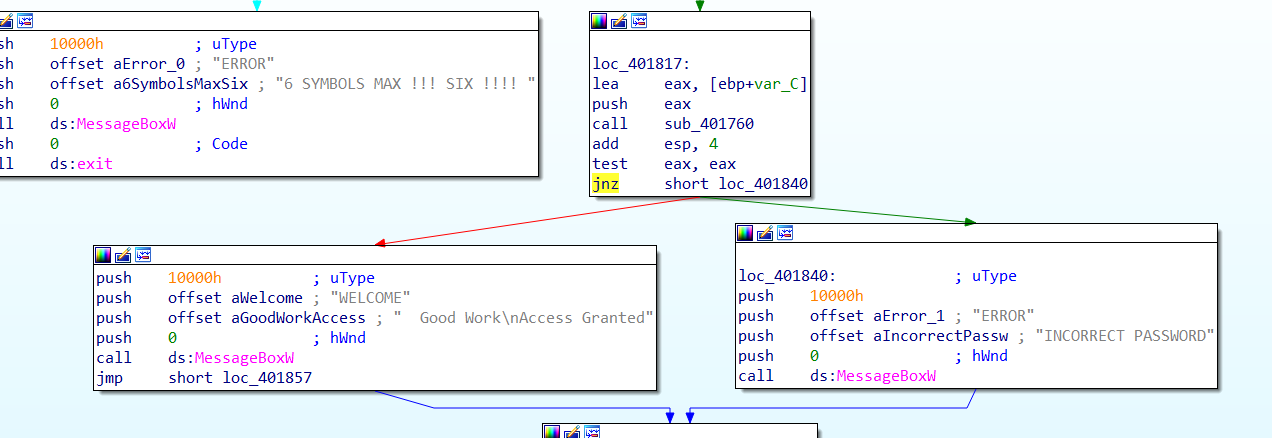


Рисунок 15– Участок модифицированного дизассемблированного кода содержащие команды JNZ

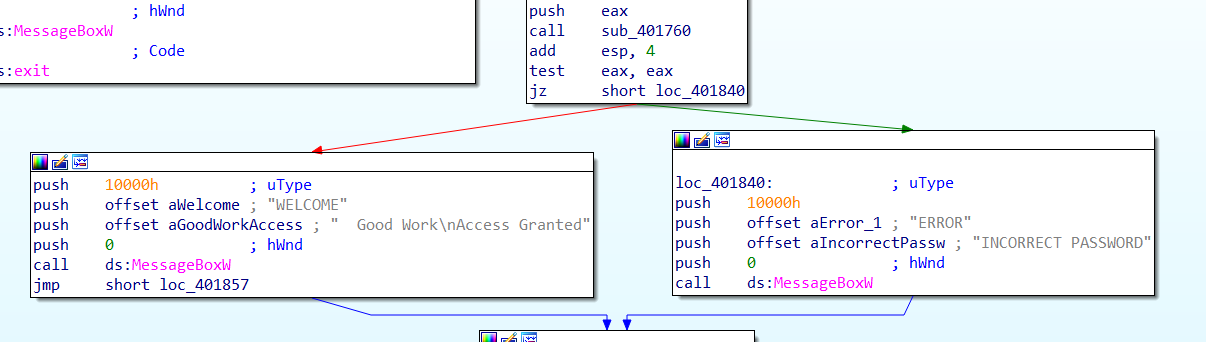


Рисунок 16– Участок дизассемблированного кода содержащие команды

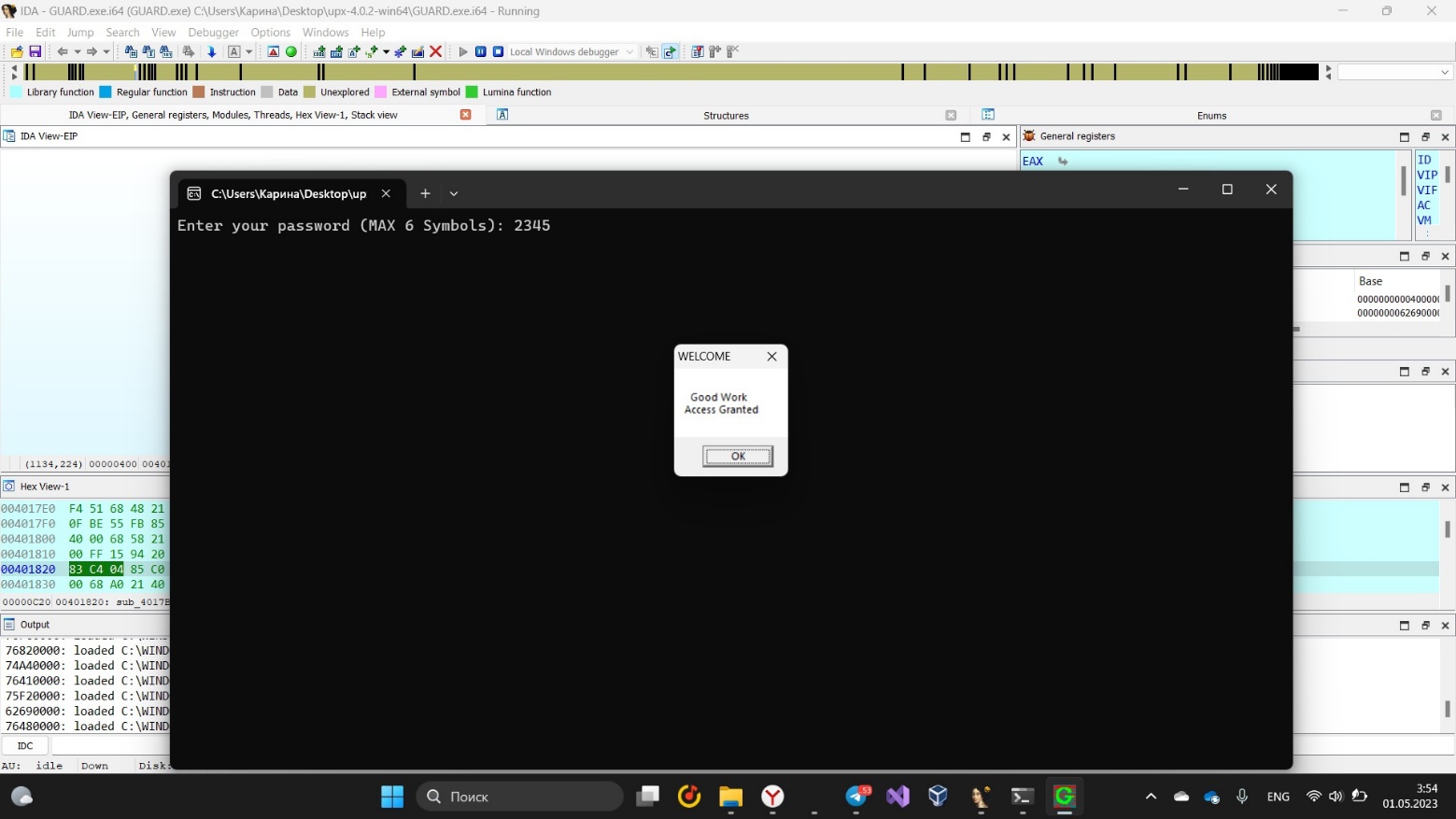


Рисунок 17– Результаты работы программы после изменений дизассемблированного кода

Далее нужно отключить проверку использования отладчика нужно в функции IsDebuggerPresent заменить инструкцию на JMP. Таким образом использование отладчика не будет проверятся.

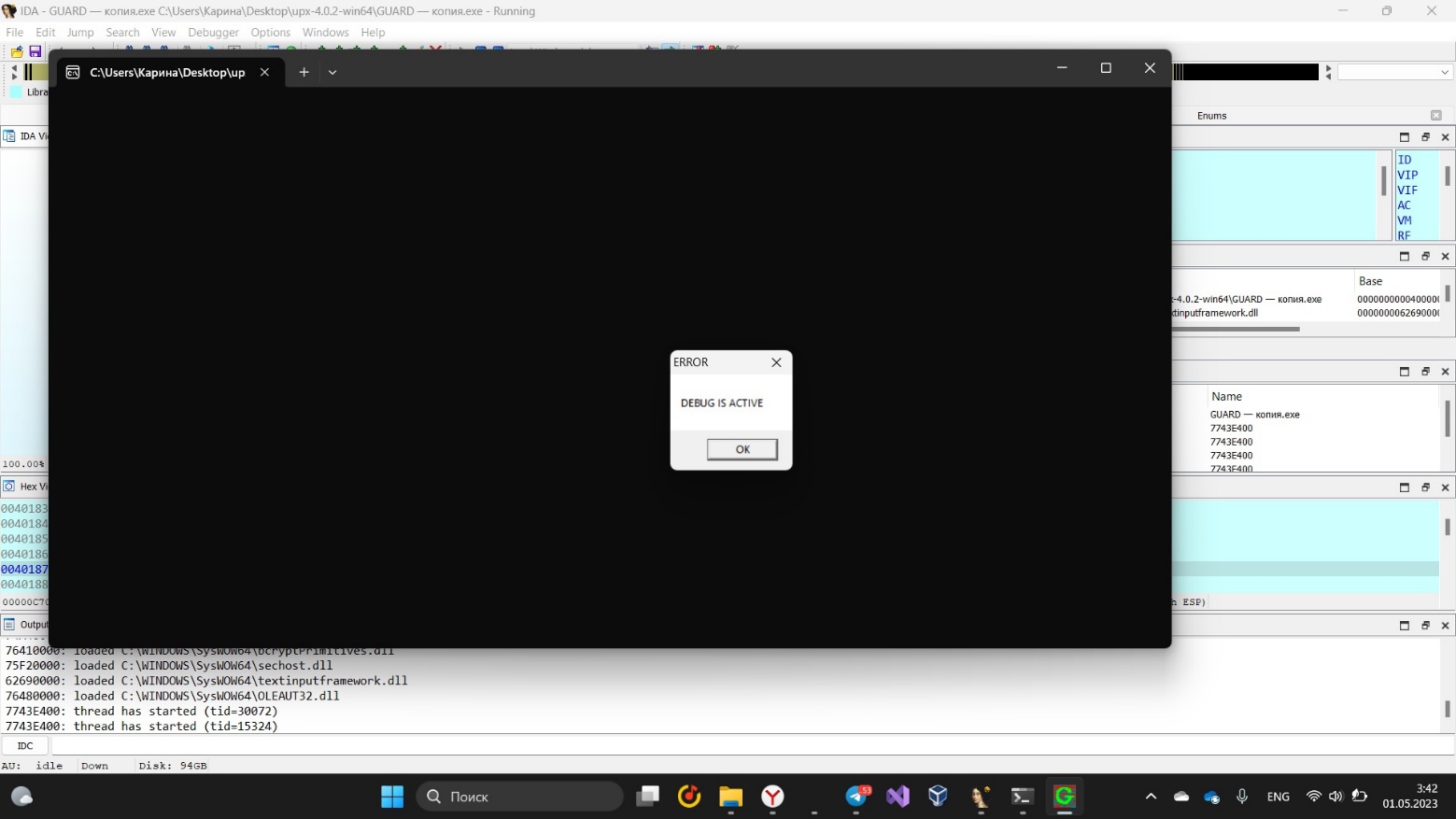


Рисунок 18– Результаты работы программы до изменений дизассемблированного кода

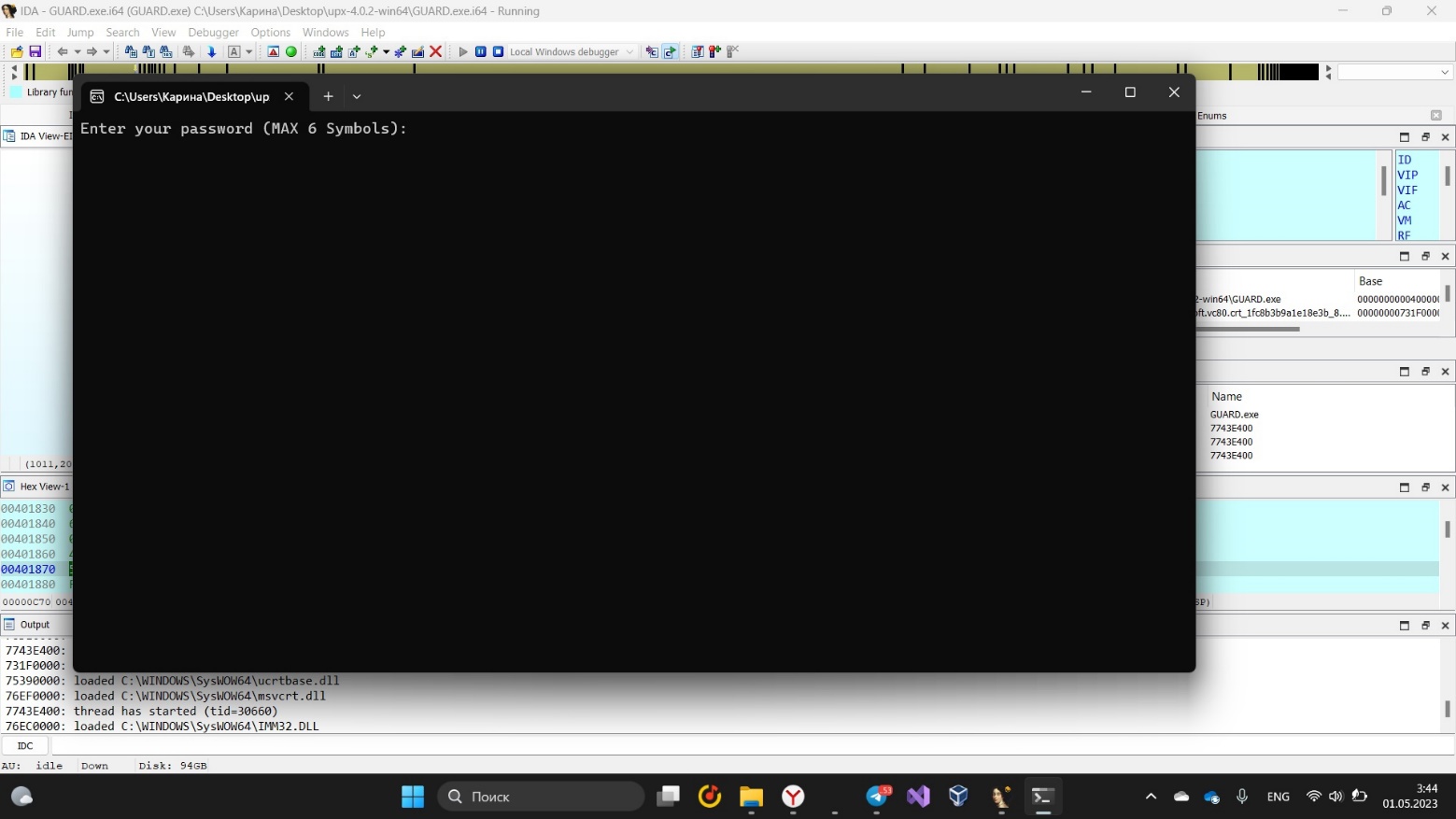


Рисунок 19– Результаты работы программы после изменений дизассемблированного кода

3)Обфускатор

Текст программы до работы обфускатора:

#define \_CRT\_SECURE\_NO\_WARNINGS

#define key 5

#include <stdio.h>

#include <string.h>

void XOR(char\* pass)

{

for (int i = 0; i < 128; i++)

{

if (pass[i] != '\0') pass[i] ^= key;

}

}

int proverka(char\* parol, char\* consol) {

int i = 0; int lenp = 0, lenc = 0;

while (parol[i] != '\0' && consol[i] != '\0' && parol[i] == consol[i]) {

i++;

}

int j = 0;

while (parol[j] != '\0') {

lenp++; j++;

}

if (i == lenp) {

return 1;

}

else return 0;

}

int main() {

FILE\* pasfile;

pasfile = fopen("password.txt", "r");

char text[128] = { 0 };

char parol[128] = { 0 };

int c; int i = 0;

while ((c = fgetc(pasfile)) != EOF) {

parol[i] = c;

i++;

}

fclose(pasfile);

parol[i] = '\0';

printf("Enter password: ");

scanf("%s", &text);

XOR(text);

int p;

if (p = proverka(parol, text) == 0) {

printf("Password ERROR\n");

}

else {

printf("Password accept, WELCOME\n");

}

getch();

}

Текст программы после работы обфускатора:

#define \_CRT\_SECURE\_NO\_WARNINGS

#define key 5

#include <stdio.h>

#include <string.h>

int ghidroegl(int kd, int tpegopsbg) { int dhshh = 0; kd = 0; tpegopsbg = 765; while (kd < tpegopsbg) kd++; return 0; } void jei(int lons) { for (int mew = 1; mew < 30; mew++) lons++; int kr = 55; } int zhyfkmopu(int nu, int poiuytrew) { int qwert = 0; nu = 0; poiuytrew = 123; while (nu < poiuytrew) nu++; return 0; } char fghnjmhgfd = '//';int fgfghvv=4; void XOR(char\* pass) { for (int kjghtryofn = 0; kjghtryofn < 128; kjghtryofn++) { if (pass[kjghtryofn] != '\0') pass[kjghtryofn] ^= key;int jdjfjt=76; } } int sgfdgshgsh(char\* parol, char\* consol) { int gththfdhsw = 0; int jtuyjyjfgr = 0, lenc = 0; while (parol[gththfdhsw] != '\0' && consol[gththfdhsw] != '\0' && parol[gththfdhsw] == consol[gththfdhsw]) { gththfdhsw++;int lKFJFjvo3; } int ergtbfdsds = 0; while (parol[ergtbfdsds] != '\0') { jtuyjyjfgr++; ergtbfdsds++;int jjksooooo; } if (gththfdhsw == jtuyjyjfgr) { return 1; } else return 0; } int main() { FILE\* pasfile;int hgfdjfjdf; pasfile = fopen("password.txt", "r");int kasdatvsf; char hehtjytjhr[128] = { 0 };int cnt\_\_\_\_=0; char kwegfgyntn[128] = { 0 };int rrr[5000]; int leoprepoir; int qoiwewieii = 0; while ((leoprepoir = fgetc(pasfile)) != EOF) { kwegfgyntn[qoiwewieii] = leoprepoir;int m[500000]; qoiwewieii++;int ghfshgh=3; } fclose(pasfile);int m[238882]; kwegfgyntn[qoiwewieii] = '\0';int tttttwttw; printf("Enter password: ");int hgfdjrftg; scanf("%s", &hehtjytjhr);int memper=23; XOR(hehtjytjhr);int lubludimu; int oropolgddd;int sobaka=34; char mjajsjdhak = '//';int drug[673]; if (oropolgddd = sgfdgshgsh(kwegfgyntn, hehtjytjhr) == 0) { printf("Password ERROR /\* COmment 2 \*/\n");int rhehrueur; } else { printf("Password accept, // Comment 1 WELCOME\n"); } getch(); }

Конфигурационные параметры обфускатора:

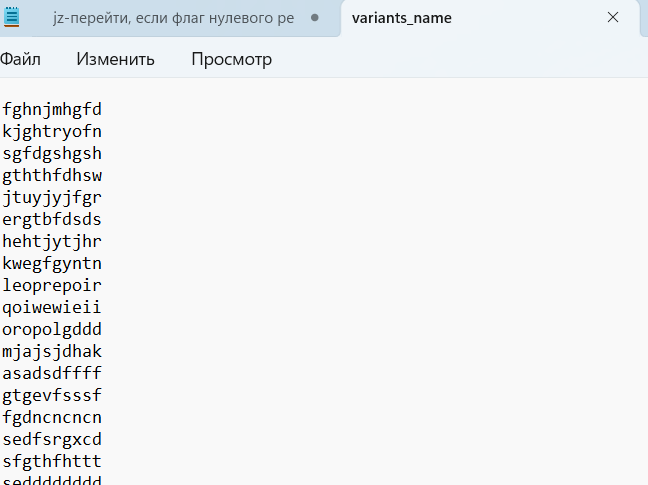


Рисунок 20– Имена переменных для замены

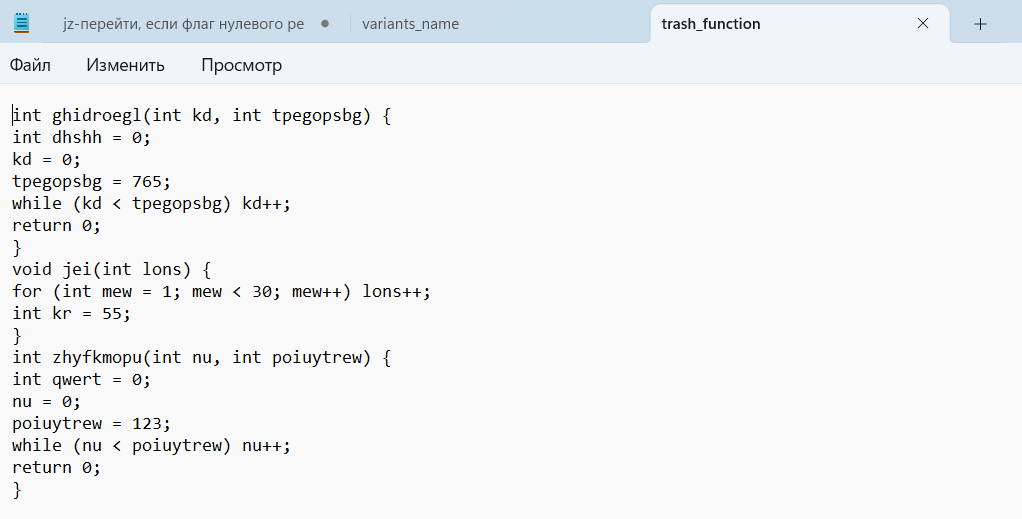


Рисунок 21– Мусорные функции

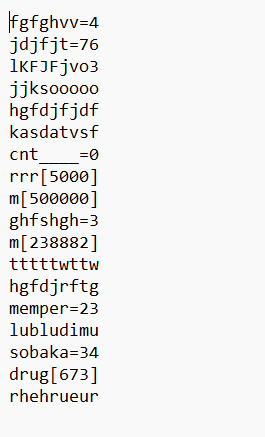


Рисунок 21– Мусорные переменные

1. **Контрольные вопросы**

*1)Какие методы применяются для защиты коммерческих программ от ввода некорректных данных?*

* запрос пароля у пользователя
* запрет на открытое хранение пароля во внешнем файле
* появление всплывающего окна, мешающего использованию программы

*2)По каким признакам можно найти интересующий код сопоставления с паролем в программе?*

Сопоставлению пароля в программе будет соответствовать команда CMP или команда условного перехода.

*3)Какие существуют основные методы защиты от дизассемблирования и отладки программ?*

* проверка целостности программ
* применение оптимизирующего компилятора
* применение упаковки и шифрования распространяемого программного обеспечения;
* проверка наличия отладчика.

*4)Как реализуется обфускация с помощью виртуальных машин?*   
Обфускация с помощью виртуальных машин (VM) заключается в написании кода на языке VM, со случайным набором инструкций, его трансляции в инструкции виртуальной машины, обфускации кода на языке VM, трансляции инструкций виртуальной машины в машинный код, и, возможно, в дополнительной обфускации машинного кода. Этот метод обеспечивает защиту кода от декомпиляции и обратного инжиниринга, и широко используется в системах защиты программного обеспечения.

*5)Предложите методы усиления механизмов защиты программ от нелегального использования, исследованные в данной работе.*

* шифровать данные учетной записи пользователя;
* использовать большое количество ссылок на разные участки кода;
* использовать проверку на присутствие отладчика, и всплывающие окна.

1. **Выводы**

В ходе работы мы знакомились с простейшими методами защиты программ от использования их незарегистрированными пользователями, приобрели практические навыки по анализу исполняемого кода без исходных текстов, исследовали способы защиты программ от дизассемблирования и отладки. В данном виде защита обеспечивает очень низкую степень защиты, достаточно замены одной ассемблерной команды. Изучили обфусцирование и написали программу обфускатор для приложения запрашивающего пароль.

ПРИЛОЖЕНИЕ А

Листинг программы «Ввод пароля» до модификации

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <string.h>

int proverka(char\* parol, char\* consol) {

int i = 0; int lenp = 0, lenc = 0;

while (parol[i] != '\0' && consol[i] != '\0' && parol[i] == consol[i]) {

i++;

}

int j = 0;

while (parol[j] != '\0') {

lenp++; j++;

}

if (i == lenp) {

return 1;

}

else return 0;

}

int main() {

FILE\* pasfile;

pasfile = fopen("password.txt", "r");

char text[128] = { 0 };

char parol[128] = { 0 };

int c; int i = 0;

while ((c = fgetc(pasfile)) != EOF) {

parol[i] = c;

i++;

}

fclose(pasfile);

parol[i] = '\0';

printf("Enter password: ");

scanf("%s", &text);

int p;

if (p = proverka(parol, text) == 0) {

printf("Password ERROR\n");

}

else {

printf("Password accept, WELCOME\n");

}

getch();

}

ПРИЛОЖЕНИЕ B

Листинг программы «Ввод пароля» после модификации

#define \_CRT\_SECURE\_NO\_WARNINGS

#define key 5

#include <stdio.h>

#include <string.h>

void XOR(char\* pass)

{

for (int i = 0; i < 128; i++)

{

if (pass[i] != '\0') pass[i] ^= key;

}

}

int proverka(char\* parol, char\* consol) {

int i = 0; int lenp = 0, lenc = 0;

while (parol[i] != '\0' && consol[i] != '\0' && parol[i] == consol[i]) {

i++;

}

int j = 0;

while (parol[j] != '\0') {

lenp++; j++;

}

if (i == lenp) {

return 1;

}

else return 0;

}

int main() {

FILE\* pasfile;

pasfile = fopen("password.txt", "r");

char text[128] = { 0 };

char parol[128] = { 0 };

int c; int i = 0;

while ((c = fgetc(pasfile)) != EOF) {

parol[i] = c;

i++;

}

fclose(pasfile);

parol[i] = '\0';

printf("Enter password: ");

scanf("%s", &text);

XOR(text);

int p;

if (p = proverka(parol, text) == 0) {

printf("Password ERROR\n");

}

else {

printf("Password accept, WELCOME\n");

}

getch();

}

ПРИЛОЖЕНИЕ B

Листинг программы «Обфускатор»

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#define MAX\_SIZE 10000

FILE\* input;

FILE\* output;

char\* code;

int l2 = 0;

int l = 0;

void zamena(char\* code) {

l = strlen(code);

int check;//указатель на то, нашли ли инициализатор для переменной

FILE\* name = fopen("variants\_name.txt", "r");

char\* nemas = (char\*)malloc(l \* sizeof(char));

int flag = 0;

char newperemen[15];

int k = 0; //счётчик для новых имен

int raznica; //разница между старой и новой длиной имени переменной

l2 = l;

for (int i = 0; i < l;)

{

/\* ЕСЛИ ВСТРЕТИЛАСЬ MAIN, ТО КОПИРУЕМ ЕЕ В НОВЫЙ МАССИВ И УВЕЛИЧИВАЕМ ЗНАЧЕНИЕ УКАЗАТЕЛЯ И FLAG НА 12 СИМВОЛОВ(ДЛИНА СТРОКИ int main()) \*/

if (code[i] == 'i' && code[i + 1] == 'n' && code[i + 2] == 't' && code[i + 3] == ' ' && code[i + 4] == 'm' && code[i + 5] == 'a' && code[i + 6] == 'i' && code[i + 7] == 'n' && code[i + 8] == '(' && code[i + 9] == ')')

{

for (int j = 0; j <= 10; j++)

{

nemas[flag + j] = code[i + j];

}

i = i + 12;

flag = flag + 12;

check = 1;

}

/\* ТА ЖЕ ТЕМА С ОСТАЛЬНЫМИ ТИПАМИ В ФАЙЛЕ \*/

if (code[i] == 'i' && code[i + 1] == 'n' && code[i + 2] == 't' && code[i + 3] == ' ')

{

for (int j = 0; j <= 3; j++)

{

nemas[flag + j] = code[i + j];

}

i = i + 4;

flag = flag + 4;

if (code[i] == '\*')

{

nemas[flag] = code[i];

i++;

flag++;

}

check = 1;

}

else if (code[i] == 'c' && code[i + 1] == 'h' && code[i + 2] == 'a' && code[i + 3] == 'r' && code[i + 4] == ' ' && (code[i + 6] != '='))

{

for (int j = 0; j <= 4; j++)

{

nemas[flag + j] = code[i + j];

}

i = i + 5;

flag = flag + 5;

if (code[i] == '\*')

{

nemas[flag] = code[i];

i++;

flag++;

}

check = 1;

}

else if (code[i] == 'F' && code[i + 1] == 'I' && code[i + 2] == 'L' && code[i + 3] == 'E' && code[i + 4] == ' ')

{

for (int j = 0; j <= 4; j++)

{

nemas[flag + j] = code[i + j];

}

i = i + 5;

flag = flag + 5;

if (code[i] == '\*')

{

nemas[flag] = code[i];

i++;

flag++;

}

check = 1;

}

/\* иначе все переписываем в тупую \*/

else

{

nemas[flag] = code[i];

flag++;

i++;

check = 0;

}

int savei = i;

int saveflag = flag;

int razmer = 0;

char\* peremen = (char\*)malloc(razmer \* sizeof(char));

if (check == 1) //нашли инициализатор

{

k = 0;

razmer = 0;

// Перебираем символы в строке до тех пор, пока не встретим символ, означающий конец имени переменной

while (code[i] != ' ' && code[i] != '[' && code[i] != '(' && code[i] != ',' && code[i] != ';' && code[i] != ')')

{

peremen = (char\*)realloc(peremen, (razmer + 1) \* sizeof(char));

peremen[razmer] = code[i];

i++; //перемещаем указатель

razmer++;

}

fgets(newperemen, 15, name); //считываем новое имя переменной из файла

raznica = 10 - razmer;

int kontrol;

nemas = (char\*)realloc(nemas, (l + raznica) \* sizeof(char)); //новый массив для хранения

for (int a = 0; a < 10; a++)

{

nemas[flag] = newperemen[a]; //замена сама

flag++;

}

k++;

// Перебираем оставшуюся часть кода и заменяем все вхождения старого имени переменной на новое

for (int j = i; j <= l; j++)

{

kontrol = 0;

for (int a = 0; a < razmer; a++)

{

if (code[j + a] == peremen[a])

{

kontrol = 1; //нашли вхождение

}

else

{

kontrol = 0;

break;

}

}

if (kontrol == 1 && (code[j + razmer] == ' ' || code[j + razmer] == '[' || code[j + razmer] == '(' || code[j + razmer] == ')' || code[j + razmer] == '+' || code[j + razmer] == ',' || code[j + razmer] == ';' || code[j + razmer] == ']'))

{

k++;

nemas = (char\*)realloc(nemas, (l + raznica \* k) \* sizeof(char));

for (int a = 0; a < 10; a++)

{

nemas[flag] = newperemen[a];

flag++;

}

j = j + razmer - 1;

}

else //если не нашли вхождение просто все принтуем дальше

{

nemas[flag] = code[j];

flag++;

}

}

l2 = l + raznica \* k;

memcpy(code, nemas, l2); //копируем новый массив в изначальный code

l = l2;

}

flag = saveflag + razmer;

i = flag;

}

nemas[flag++] = '\0';

code[flag++] = '\0';

fclose(name);

}

void musor(char\* code) {

FILE\* f = fopen("trash\_peremennie.txt", "r"); //имена переменных

int o;

int cnt\_trash\_perem = 1;

while ((o = fgetc(f)) != EOF) {

if (o == 10) {

cnt\_trash\_perem++;

}

}

rewind(f);

int ind = 0, i = 1;

while (i < strlen(code) - 4) {

if (code[i] == ';' && code[i + 1] == '\n' && code[i - 1] != '1' && code[i - 1] != '0' && cnt\_trash\_perem > 0) {

ind = i + 1;

char per0[12] = { 0 };

fgets(per0, 12, f);

per0[9] = ';'; per0[10] = '\0'; //считываем из файла имя переменной и добавляем ей ; и завершающий ноль

char per[15] = { 0 };

//в массив для создания мусорной перменной закидываем в начало тип (int)

per[0] = 'i'; per[1] = 'n'; per[2] = 't'; per[3] = ' '; per[4] = '\0';

//склеиваем

strcat(per, per0);

char\* code0 = (char\*)calloc(sizeof(char), (strlen(code)));

int len = strlen(code);

for (int j = 0; j < len; j++) {

// Копируем все символы после точки с запятой из code в новую строку code0, а также добавляем имя мусорной переменной вместо точки с запятой в строку code

code0[j] = code[ind + j];

if (j < strlen(per) + 1) code[ind + j] = per[j];

}

strcat(code, code0);

// Изменяем переменную i на количество добавленных символов, чтобы продолжить перебор со следующей точки с запятой

i = i + strlen(per) + 1;

cnt\_trash\_perem--;

}

i++;

}

}

void musorfunction() {

FILE\* fp\_in, \* fp\_out, \* fp\_junk;

char buffer[MAX\_SIZE]; // буфер для считывания строк из файлов

fp\_in = fopen("f\_out.txt", "r");

fp\_out = fopen("fppp\_out.txt", "w");

fp\_junk = fopen("trash\_function.txt", "r");

while (fgets(buffer, MAX\_SIZE, fp\_in)) {

if (strstr(buffer, "#include") || strstr(buffer, "#define")) {

fprintf(fp\_out, "%s", buffer); // записываем строку с include или define в файл вывода

}

else {

while (fgets(buffer, MAX\_SIZE, fp\_junk)) {

fprintf(fp\_out, "%s", buffer); // добавляем мусорные функции в файл вывода

}

}

}

fclose(fp\_in);

fp\_in = fopen("f\_out.txt", "r");

char b;

b = fgets(buffer, MAX\_SIZE, fp\_in);

while (!feof(fp\_in)) {

if ((strstr(buffer, "#include")) || (strstr(buffer, "#define"))) {

b = fgets(buffer, MAX\_SIZE, fp\_in);

}

else {

fprintf(fp\_out, "%s", buffer); //записываем оставшуюся часть кода в файл

b = fgets(buffer, MAX\_SIZE, fp\_in);

}

}

fclose(fp\_in);

fclose(fp\_out);

fclose(fp\_junk);

fp\_out = fopen("fppp\_out.txt", "r");

int e; int i = 0;

char\* txt = (char\*)calloc(100000, sizeof(char));

while (!feof(fp\_out)) {

e = fgetc(fp\_out);

if (e == -1) {

txt[i] = '}';

i++;

continue;

}

txt[i] = e;

i++;

}

txt[i] = '\0';

code = txt;

}

void tabul(char\* code) {

for (int i = 0; i < strlen(code); i++)

{

if (code[i] == '\t')

code[i] = '?';

}

}

void new(char\* code){

for (int i = 0; i < strlen(code); i++)

{

if (code[i] == '\n')

code[i] = '?';

}

}

void comment(char\* code)

{

int l = strlen(code);

int flag = 0;

for (int i = 0; i < l - 1; i++)

{

if (flag < 2 && code[i] == '\"') flag++;

if (flag < 2 && code[i] == '\'') flag++;

if (flag == 2) flag = 0; // Если нашли закрывающую кавычку или апостроф, сбрасываем счетчик flag

if (flag == 0) { // Если счетчик flag равен 0, значит мы находимся вне строковых литералов

if (code[i] == '/' && code[i + 1] == '/') // Если нашли комментарий "//", то удаляем его и все символы до конца строки

{

int counter = i;

while (code[counter] != '\n')

{

counter++;

}

int b = i;

while (counter < l) {

code[b] = code[counter];

b++;

counter++;

}

while (b < l)

code[b++] = 0;

}

if (code[i] == '/' && code[i + 1] == '\*') // Если нашли комментарий "/\*", то удаляем его и все символы до закрывающего комментария "\*/"

{

int counter = i;

while (!(code[counter - 1] == '\*' && code[counter] == '/'))

{

counter++;

}

counter++;

int buf = i;

while (counter < l) {

code[buf] = code[counter];

buf++;

counter++;

}

while (buf < l)

{

code[buf] = 0;

buf++;

}

}

}

}

}

void result(char\* code) {

int len = strlen(code);

char\* res = (char\*)malloc(sizeof(char) \* len+10);

int j = 0;

char b = ' ';

for (int i = 0; i <= len; i++) {

if (code[i] != '?') res[j++] = code[i];

if (code[i] == '?') res[j++] = b;

if ((code[i - 2] == 'y' && code[i - 1] == ' ' && code[i] == '5') || (code[i] == 'S' && code[i - 1] == 'G') || code[i] == '>') res[j++] = '\n';

}

res[j++] = '\0';

puts(res);

}

void main() {

input = fopen("password\_orig.txt", "r");

char c;

code = (char\*)calloc(100000, sizeof(char)); int i = 0;

while ((c = fgetc(input)) != EOF) {

code[i] = c;

i++;

}

i++;

code[i] = '\0';

l = i;

zamena(code);

l = l2;

musor(code);

FILE\* f\_out;

f\_out = fopen("f\_out.txt", "w");

int v = 0;

while (code[v] != '\0') {

fprintf(f\_out, "%c", code[v]);

v++;

}

v++;

if (code[v] == '\0') {

fprintf(f\_out, "%c", code[v]);

}

fclose(f\_out);

musorfunction();

comment(code);

tabul(code);

new(code);

result(code);

}