

DOKUMENTACJA SAMODZIELNEGO PROJEKTU Z PRZEDMIOTU PROGRAMOWANIE NISKOPOZIOMOWE

Temat: Szyfr One-Time Pad

Spis Treści:

1. Wyjaśnienie szyfru One-Time Pad z wykorzystaniem algorytmu Vigenère'a
2. Założenia projektowe,
3. Co umożliwia projekt,
4. Ograniczenia Projektu,
5. Rozwiązania implementacyjne, wykorzystane biblioteki, opis algorytmów,
-zrzuty ekranu i przykłady kodu źródłowego,
6. Testowanie aplikacji,
-Zrzuty ekranu i przykłady kodu źródłowego,
7. Podsumowanie.

1. Wyjaśnienie szyfru One-Time Pad z wykorzystaniem algorytmu Vigenère'a

Szyfr z kluczem jednorazowym (one-time pad) - szyfr zaproponowany w 1917 roku przez Gilberta Vernama, którego, przy poprawnym wykorzystaniu, nie można złamać. Szyfr z kluczem jednorazowym jest dużym zbiorem o niepowtarzalnych i przypadkowych sekwencjach znaków. Nadawca używa każdej litery z tego zbioru do zaszyfrowania jednego znaku tekstu jawnego.

Algorytm Vigenère'a - jest jednym z klasycznych algorytmów szyfrujących. Należy on do grupy tzw. polialfabetycznych szyfrów podstawieniowych.

Szyfr Vigenère'a może być szyfrem nie do złamania (zostało to udowodnione w 1949 przez Claude'a Elwooda Shannona) przy zachowaniu trzech reguł:

- klucz użyty do szyfrowania wiadomości musi być dłuższy lub równy szyfrowanej wiadomości,
- klucz musi być wygenerowany w sposób całkowicie losowy (nie może istnieć sposób na odtworzenie klucza na podstawie znajomości działania generatorów liczb pseudolosowych),
- klucz nie może być użyty do zaszyfrowania więcej niż jednej wiadomości.

Działanie szyfru oparte jest na takiej tablicy:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Tekst szyfrujemy na podstawie hasła. Szyfrowanie odbywa się w sposób następujący:

Przypuśćmy, że chcemy zaszyfrować prosty tekst, np.:

TO JEST BARDZO TAJNY TEKST

Do tego celu użyjemy znanego tylko nam słowa kluczowego, np. TAJNE

Na początku zauważamy, że użyte słowo kluczowe jest zbyt krótkie, by wystarczyło do zaszyfrowania całego tekstu, więc należy użyć jego wielokrotności. Będzie to miało następującą postać:

TO JEST BARDZO TAJNY TEKST

TA JNET AJNETA JNETA JNETA

Następnie wykonujemy szyfrowanie w następujący sposób: litera szyfrogramu odpowiada literze z tabeli znajdującej się na przecięciu wiersza, wyznaczanego przez literę tekstu jawnego i kolumny wyznaczonej przez literę słowa kluczowego, np. po kolei T i T daje M, O i A daje O itd. W efekcie otrzymujemy zaszyfrowany tekst:

MO SRWM BJEHSO CNNGY CROLT

2. Założenia projektowe

Głównymi założeniami projektu było zaimplementowanie skutecznego i niemożliwego do złamania szyfru typu One-Time Pad, w swoim projekcie wybrałam wersję znakową tego algorytmu, tj Algorytm Vigenere'a, korzystającego z przesunięcia Cezara do przesuwania wiadomości względem klucza. Implementacja spełnia założenia opisane powyżej,tj:

- Klucz jest jednorazowy, może być wygenerowany losowo, klucz może być dłuższy niż wiadomość do zaszyfrowania,

- co przy odpowiednim wykorzystaniu algorytmu czyni zaszyfrowany tekst niemożliwym do odczytania bez posiadania klucza szyfrującego.

3.Co umożliwia projekt

- Wpisanie własnego tekstu z klawiatury do tablicy w programie, gdzie zostanie zaszyfrowany zgodnie z algorytmem szyfrowania,

- Wpisanie własnego klucza szyfrującego wiadomość,

- Możliwa implementacja losowania klucza szyfrującego za pomocą generatora liczb pseudolosowych wykorzystujących ziarno (seed),

- Wyniki programu, tj. zaszyfrowana wiadomość i klucz szyfrujący są zwracane do dwóch oddzielnych plików o rozszerzeniu txt, przez co są łatwe do odczytania i przesłania dalej ale trudne do odszyfrowania gdy odbiorca nie posiada dwóch plików, przez co wiadomość jest bezpieczna z dala od niepowołanych osób.

4. Przyjęte ograniczenia

Z uwagi na rodzaj szyfrowania i możliwości pamięci zostały przyjęte następujące ograniczenia:

- znaki wprowadzane z klawiatury są małymi literami alfabetu, bez znaków specjalnych,
- długość ciągu znaków nie przekracza 255

Ograniczenia te nie wpływają jednak na złożoność szyfrowania i jego niezawodność, przez co szyfr nie traci na swej wartości i użyteczności.

5. Rozwiązania Implementacyjne

Rozróżniamy dwa podstawowe rodzaje szyfru One-Time Pad:

- algorytm binarny XOR,
- algorytm znakowy z wykorzystaniem algorytmu Vigenere'a

W swoim projekcie skupiłam się na rozwiązaniu znakowym, ponieważ na poziomie liczb binarnych i kodach ASCII prezentuje się on ciekawiej niż dość prosty algorytm XOR.

Przy implementacji projektu wykorzystywałam biblioteki winapi:

- Kernel32.lib,
- masm32.lib.

Ważniejsze rodzaje zmiennych i rejestrów wykorzystywane w projekcie:

- tablice 16- i 32-bitowe,
- pomocnicze zmienne 16-bitowe,
- uchwyt 32-bitowe,
- rejestry EAX, EBX, AL, EDX, ECX, ESI i EDI

Wykorzystywane procedury:

```
GetStdHandle PROTO :DWORD
WriteConsoleA PROTO :DWORD, :DWORD, :DWORD, :DWORD, :DWORD
ReadConsoleA PROTO :DWORD, :DWORD, :DWORD, :DWORD, :DWORD
ExitProcess PROTO :DWORD
wsprintfA PROTO C :VARARG ; prototyp procedury w masm32
CreateFileA PROTO :DWORD, :DWORD, :DWORD, :DWORD, :DWORD, :DWORD, :DWORD
WriteFile PROTO :DWORD, :DWORD, :DWORD, :DWORD, :DWORD
```

Główny algorytm szyfrujący STEP by STEP:

```
;SZYFROWANIE MOTODA VIGENERE 'A

mov EBX, rin
sub EBX, 2
mov pom, EBX
mov EBX, 0
mov EDX, 0
mov EAX, 0
.WHILE EBX < pom

    mov EAX, 0
    mov EAX, wiadomosc[EBX]
    mov EDX, 0d
    mov EDX, klucz2[EBX]
    sub EDX, 97d
    add EAX, EDX
    .IF AL > 7Ah

        sub EAX, 7Ah
        add EAX, 61h
        mov pom2, EAX ; w pom mam zakodowany kod ascii
        mov zakodowane[EBX], AL

    .ENDIF

    inc EBX
    inc EDX
    inc pom

.ENDIF
```

1. Do rejestru EBX kopiowana jest ilość znaków wiadomości do zaszyfrowania, odejmowane są ostatnie dwa znaki, a otrzymana realna wartość jest kopiowana do pomocniczej zmiennej pom
2. zerowane są inne potrzebne rejestry,
3. pętla .WHILE wykonuje się tyle razy ile znaków ma wiadomość,
4. z tablicy wiadomości brane po kolei wartości znaków ASCII wiadomości,
5. odejmowana jest wartość 97d, ponieważ od tej liczby zaczynają się pożądane przez nas znaki w tablicy ASCII,
6. do wartości znaku z tablicy wiadomości dodawana jest wartość znaku z tablicy klucz, wynik zapisywany w rejestrze EAX,
7. jeśli wartość w tym rejestrze jest większa niż 122d wchodzimy w instrukcję .IF, musimy tak zrobić, aby znaki przesuwali się zgodnie z algorytmem który jest zaimplementowany,
8. odejmujemy wartość 7Ah od rejestru EAX, jest to w systemie dziesiętnym wartość 122d, dzięki czemu mamy w rejestrze wartość przesunięcia względem wartości klucza,
9. dodajemy do wartości przesunięcia liczbę 97d -61h- co odpowiada kodowi ASCII pierwszego interesującego nas znaku czyli literki 'a'. Dzięki temu mamy w rejestrze EAX kod ASCII zakodowanej litery
10. przenosimy ją na wolne miejsce w tablicy zakodowane, gdzie mamy gotową zakodowaną wiadomość po ostatnim obiegu pętli.

6. Testowanie

Zacznijmy od początku:

```
C:\Users\IEUser\Documents\Visual Studio 2015\Projects\Projekt\Debug\Projekt.exe

Marta Kisielinska - projekt z Programowania Niskopoziomowego
Szyfrowanie One-Time Pad - wersja znakowa
Szyfr Vigenerea-jest jednym z klasycznych algorytmow szyfrujacych. Nalezy on do grupy tzw. polialfabetu
Podaj wiadomosc do zakodowania, max 255 malych liter bez znakow specjalnych
```

Pierwsze uruchomienie Aplikacji, nazwisko autora, krótki opis i zachęta do wpisania swojej wiadomości do zaszyfrowania razem z ostrzeżeniem o ograniczeniach.

```
Marta Kisielinska - projekt z Programowania Niskopoziomowego
Szyfrowanie One-Time Pad - wersja znakowa
Szyfr Vigenerea-jest jednym z klasycznych algorytmow szyfrujacych. Nalezy on do grupy tzw. polialfabetu
Podaj wiadomosc do zakodowania, max 255 malych liter bez znakow specjalnych
mojatajnawiadomosc
```

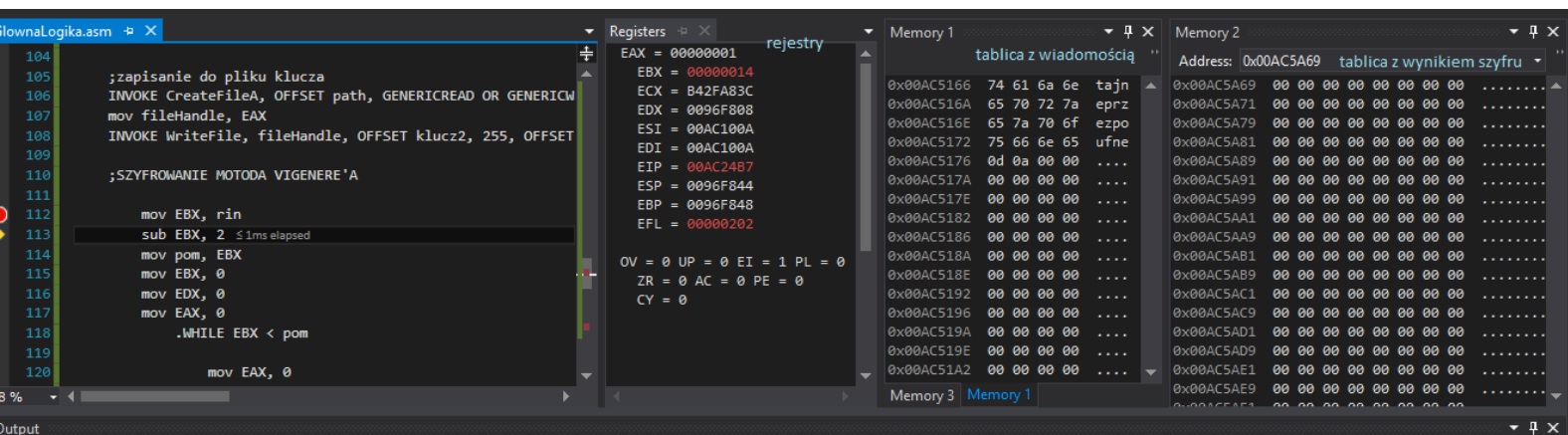
Moja „tajna wiadomość”,

po zatwierdzeniu enterem:

```
Podaj wiadomosc do zakodowania
mojatajnawiadomosc
wpisz klucz, minimum 20 znakow
```

Dzięki procedurze `wsprintfA` kod „wie” ile muszę wpisać znaków aby wiadomość była odpowiednio zakodowana, użytkownik nie musi liczyć ile znaków podać.

Po wpisaniu „hasła” dzieje się algorytm:



Możemy podejrzeć co dzieje się w rejestrach, przygotowywane są na operowanie danymi z ich pomocą, tj czyszczone.

ownaLogika.asm

```
104 ;zapisanie do pliku klucza
105 INVOKE CreateFileA, OFFSET path, GENERIC_READ OR GENERIC_WRITE, 0, NULL, 0, FILE_ATTRIBUTE_NORMAL, 0
106 mov fileHandle, EAX
107 INVOKE WriteFile, fileHandle, OFFSET klucz2, 255, OFFSET 0, NULL
108 ;SZYFROWANIE MOTODA VIGENERE'A
109
110 mov EBX, rin
111 sub EBX, 2
112 mov pom, EBX
113 mov EBX, 0
114 mov EDX, 0
115 mov EAX, 0
116 .WHILE EBX < pom ≤ 1ms elapsed
117
118 mov EAX, 0
119
120 mov EAX, 0
```

Registers

EAX = 00000000
EBX = 00000000
ECX = B42FA83C
EDX = 00000000
ESI = 00AC100A
EDI = 00AC100A
EIP = 00AC24CF
ESP = 0096F844
EBP = 0096F848
EFL = 0000206

OV = 0 UP = 0 EI = 1 PL = 0
ZR = 0 AC = 0 PE = 1
CY = 0

Memory 1

0x00AC5166 74 61 6a 6e tajn
0x00AC516A 65 70 72 7a eprz
0x00AC516E 65 7a 70 6f eprz
0x00AC5172 75 66 6e 65 ufne
0x00AC5176 0d 0a 00 00
0x00AC517A 00 00 00 00
0x00AC517E 00 00 00 00
0x00AC5182 00 00 00 00
0x00AC5186 00 00 00 00
0x00AC518A 00 00 00 00
0x00AC518E 00 00 00 00
0x00AC5192 00 00 00 00
0x00AC5196 00 00 00 00
0x00AC519A 00 00 00 00
0x00AC519E 00 00 00 00
0x00AC51A2 00 00 00 00

Memory 2

Address: 0x00AC5A69

0x00AC5A69 00 00 00 00 00 00 00 00
0x00AC5A71 00 00 00 00 00 00 00 00
0x00AC5A79 00 00 00 00 00 00 00 00
0x00AC5A81 00 00 00 00 00 00 00 00
0x00AC5A89 00 00 00 00 00 00 00 00
0x00AC5A91 00 00 00 00 00 00 00 00
0x00AC5A99 00 00 00 00 00 00 00 00
0x00AC5AA1 00 00 00 00 00 00 00 00
0x00AC5AA9 00 00 00 00 00 00 00 00
0x00AC5AB1 00 00 00 00 00 00 00 00
0x00AC5AB9 00 00 00 00 00 00 00 00
0x00AC5AC1 00 00 00 00 00 00 00 00
0x00AC5AC9 00 00 00 00 00 00 00 00
0x00AC5AD1 00 00 00 00 00 00 00 00
0x00AC5AD9 00 00 00 00 00 00 00 00
0x00AC5AE1 00 00 00 00 00 00 00 00
0x00AC5AE9 00 00 00 00 00 00 00 00
0x00AC5AF1 00 00 00 00 00 00 00 00

Po przygotowaniu rejestrów możemy przejść do właściwego algorytmu

ownaLogika.asm

```
112 mov EBX, rin
113 sub EBX, 2
114 mov pom, EBX
115 mov EBX, 0
116 mov EDX, 0
117 mov EAX, 0
118 .WHILE EBX < pom
119
120 mov EAX, 0
121 mov EAX, wiadomosc[EBX]
122 mov EDX, 0d ≤ 1ms elapsed
123 mov EDX, klucz2[EBX]
124 sub EDX, 97d
125 add EAX, EDX
126 .IF AL > 7Ah
127
128 sub EAX, 7Ah
```

Registers

EAX = 6E6A6174
EBX = 00000000
ECX = B42FA83C
EDX = 00000000
ESI = 00AC100A
EDI = 00AC100A
EIP = 00AC24DC
ESP = 0096F844
EBP = 0096F848
EFL = 0000297

OV = 0 UP = 0 EI = 1 PL = 1
ZR = 0 AC = 1 PE = 1
CY = 1

Memory 1

0x00AC5166 74 61 6a 6e tajn
0x00AC516A 65 70 72 7a eprz
0x00AC516E 65 7a 70 6f eprz
0x00AC5172 75 66 6e 65 ufne
0x00AC5176 0d 0a 00 00
0x00AC517A 00 00 00 00
0x00AC517E 00 00 00 00
0x00AC5182 00 00 00 00
0x00AC5186 00 00 00 00
0x00AC518A 00 00 00 00
0x00AC518E 00 00 00 00
0x00AC5192 00 00 00 00
0x00AC5196 00 00 00 00
0x00AC519A 00 00 00 00
0x00AC519E 00 00 00 00
0x00AC51A2 00 00 00 00

Memory 2

Address: 0x00AC5A69

0x00AC5A69 00 00 00 00 00 00 00 00
0x00AC5A71 00 00 00 00 00 00 00 00
0x00AC5A79 00 00 00 00 00 00 00 00
0x00AC5A81 00 00 00 00 00 00 00 00
0x00AC5A89 00 00 00 00 00 00 00 00
0x00AC5A91 00 00 00 00 00 00 00 00
0x00AC5A99 00 00 00 00 00 00 00 00
0x00AC5AA1 00 00 00 00 00 00 00 00
0x00AC5AA9 00 00 00 00 00 00 00 00
0x00AC5AB1 00 00 00 00 00 00 00 00
0x00AC5AB9 00 00 00 00 00 00 00 00
0x00AC5AC1 00 00 00 00 00 00 00 00
0x00AC5AC9 00 00 00 00 00 00 00 00
0x00AC5AD1 00 00 00 00 00 00 00 00
0x00AC5AD9 00 00 00 00 00 00 00 00
0x00AC5AE1 00 00 00 00 00 00 00 00
0x00AC5AE9 00 00 00 00 00 00 00 00
0x00AC5AF1 00 00 00 00 00 00 00 00

Algorytm wykonuje się zgodnie z opisem powyżej, krok po kroku, mamy wgląd we wszystko co dzieje się w pamięci i rejestrach

ownaLogika.asm

```
112 mov EBX, rin
113 sub EBX, 2
114 mov pom, EBX
115 mov EBX, 0
116 mov EDX, 0
117 mov EAX, 0
118 .WHILE EBX < pom
119
120 mov EAX, 0
121 mov EAX, wiadomosc[EBX]
122 mov EDX, 0d
123 mov EDX, klucz2[EBX]
124 sub EDX, 97d
125 add EAX, EDX
126 .IF AL > 7Ah ≤ 1ms elapsed
127
128 sub EAX, 7Ah
```

Registers

EAX = D4E3DB86
EBX = 00000000
ECX = B42FA83C
EDX = 66797A12
ESI = 00AC100A
EDI = 00AC100A
EIP = 00AC24EC
ESP = 0096F844
EBP = 0096F848
EFL = 0000A82

OV = 1 UP = 0 EI = 1 PL = 1
ZR = 0 AC = 0 PE = 0
CY = 0

Memory 1

0x00AC5166 74 61 6a 6e tajn
0x00AC516A 65 70 72 7a eprz
0x00AC516E 65 7a 70 6f eprz
0x00AC5172 75 66 6e 65 ufne
0x00AC5176 0d 0a 00 00
0x00AC517A 00 00 00 00
0x00AC517E 00 00 00 00
0x00AC5182 00 00 00 00
0x00AC5186 00 00 00 00
0x00AC518A 00 00 00 00
0x00AC518E 00 00 00 00
0x00AC5192 00 00 00 00
0x00AC5196 00 00 00 00
0x00AC519A 00 00 00 00
0x00AC519E 00 00 00 00
0x00AC51A2 00 00 00 00

Memory 2

Address: 0x00AC5A69

0x00AC5A69 00 00 00 00 00 00 00 00
0x00AC5A71 00 00 00 00 00 00 00 00
0x00AC5A79 00 00 00 00 00 00 00 00
0x00AC5A81 00 00 00 00 00 00 00 00
0x00AC5A89 00 00 00 00 00 00 00 00
0x00AC5A91 00 00 00 00 00 00 00 00
0x00AC5A99 00 00 00 00 00 00 00 00
0x00AC5AA1 00 00 00 00 00 00 00 00
0x00AC5AA9 00 00 00 00 00 00 00 00
0x00AC5AB1 00 00 00 00 00 00 00 00
0x00AC5AB9 00 00 00 00 00 00 00 00
0x00AC5AC1 00 00 00 00 00 00 00 00
0x00AC5AC9 00 00 00 00 00 00 00 00
0x00AC5AD1 00 00 00 00 00 00 00 00
0x00AC5AD9 00 00 00 00 00 00 00 00
0x00AC5AE1 00 00 00 00 00 00 00 00
0x00AC5AE9 00 00 00 00 00 00 00 00
0x00AC5AF1 00 00 00 00 00 00 00 00

Widzimy tutaj, że wartość sumy kodów ASCII przekroczy 122d, więc wykonujemy instrukcję .IF


```
mov EAX, 0
mov EAX, wiadomosc[EBX]
mov EDX, 0d
mov EDX, klucz2[EBX]
sub EDX, 97d
add EAX, EDX
    .IF AL > 7Ah
        sub EAX, 7Ah
        add EAX, 61h ≤ 1ms elapsed
        mov pom2, EAX ; w pom mam zakodowany
        mov zakodowane[EBX], AL
    .ENDIF
mov zakodowane[EBX], AL
```

EAX = D4E3DB0C
EBX = 00000000
ECX = B42FA83C
EDX = 66797A12
ESI = 00AC100A
EDI = 00AC100A
EIP = 00AC24F3
ESP = 0096F844
EBP = 0096F848
EFL = 00000296

OV = 0 UP = 0 EI = 1 PL = 1
ZR = 0 AC = 1 PE = 1
CY = 0

W instrukcji .IF odejmujemy od rejestru EAX 122d aby otrzymać przesunięcie względem pierwszego znaku w alfabecie, potem dodajmy 97d i otrzymujemy w rejestrze EAX kod ASCII odpowiadający zakodowanej już literce

Mamy zakodowany pierwszy znak zgodnie z wcześniejszymi założeniami

```
mov EAX, wiadomosc[EBX]
mov EDX, 0d
mov EDX, klucz2[EBX]
sub EDX, 97d
add EAX, EDX
    .IF AL > 7Ah
        sub EAX, 7Ah
        add EAX, 61h
        mov pom2, EAX ; w pom mam zakodowany
        mov zakodowane[EBX], AL
    .ENDIF
mov zakodowane[EBX], AL
inc EBX ≤ 1ms elapsed
.ENDW
```

EAX = D7D4E37A
EBX = 00000001
ECX = B42FA83C
EDX = 72667919
ESI = 00AC100A
EDI = 00AC100A
EIP = 00AC2507
ESP = 0096F844
EBP = 0096F848
EFL = 00000246

OV = 0 UP = 0 EI = 1 PL = 0
ZR = 1 AC = 0 PE = 1
CY = 0

Address: 0x00AC5A69

0x00AC5166	74	61	6a	6e	tajn	0x00AC5A69	6d	7a	00	00	00	00	00	00	mz.....
0x00AC516A	65	70	72	7a	eprz	0x00AC5A71	00	00	00	00	00	00	00	00
0x00AC516E	65	7a	70	6f	ezpo	0x00AC5A79	00	00	00	00	00	00	00	00
0x00AC5172	75	66	6e	65	ufne	0x00AC5A81	00	00	00	00	00	00	00	00
0x00AC5176	0d	0a	00	00	0x00AC5A89	00	00	00	00	00	00	00	00
0x00AC517A	00	00	00	00	0x00AC5A91	00	00	00	00	00	00	00	00
0x00AC517E	00	00	00	00	0x00AC5A99	00	00	00	00	00	00	00	00
0x00AC5182	00	00	00	00	0x00AC5AA1	00	00	00	00	00	00	00	00
0x00AC5186	00	00	00	00	0x00AC5AA9	00	00	00	00	00	00	00	00
0x00AC518A	00	00	00	00	0x00AC5AB1	00	00	00	00	00	00	00	00
0x00AC518E	00	00	00	00	0x00AC5AB9	00	00	00	00	00	00	00	00
0x00AC5192	00	00	00	00	0x00AC5AC1	00	00	00	00	00	00	00	00
0x00AC5196	00	00	00	00	0x00AC5AC9	00	00	00	00	00	00	00	00
0x00AC519A	00	00	00	00	0x00AC5AD1	00	00	00	00	00	00	00	00
0x00AC519E	00	00	00	00	0x00AC5AD9	00	00	00	00	00	00	00	00
0x00AC51A2	00	00	00	00	0x00AC5AE1	00	00	00	00	00	00	00	00

Memory 3 Memory 1

```
ownaLogika.asm
122 mov EDX, 0d
123 mov EDX, klucz2[EBX]
124 sub EDX, 97d
125 add EAX, EDX
126 .IF AL > 7Ah
127     sub EAX, 7Ah
128     add EAX, 61h
129     mov pom2, EAX ; w pom mam zakodowany
130     mov zakodowane[EBX], AL
131 .ENDIF
132
133     mov zakodowane[EBX], AL ≤ 1ms elapsed
134     inc EBX
135 .ENDW
```

EAX = D4E3DB6D
EBX = 00000000
ECX = B42FA83C
EDX = 66797A12
ESI = 00AC100A
EDI = 00AC100A
EIP = 00AC2501
ESP = 0096F844
EBP = 0096F848
EFL = 00000282

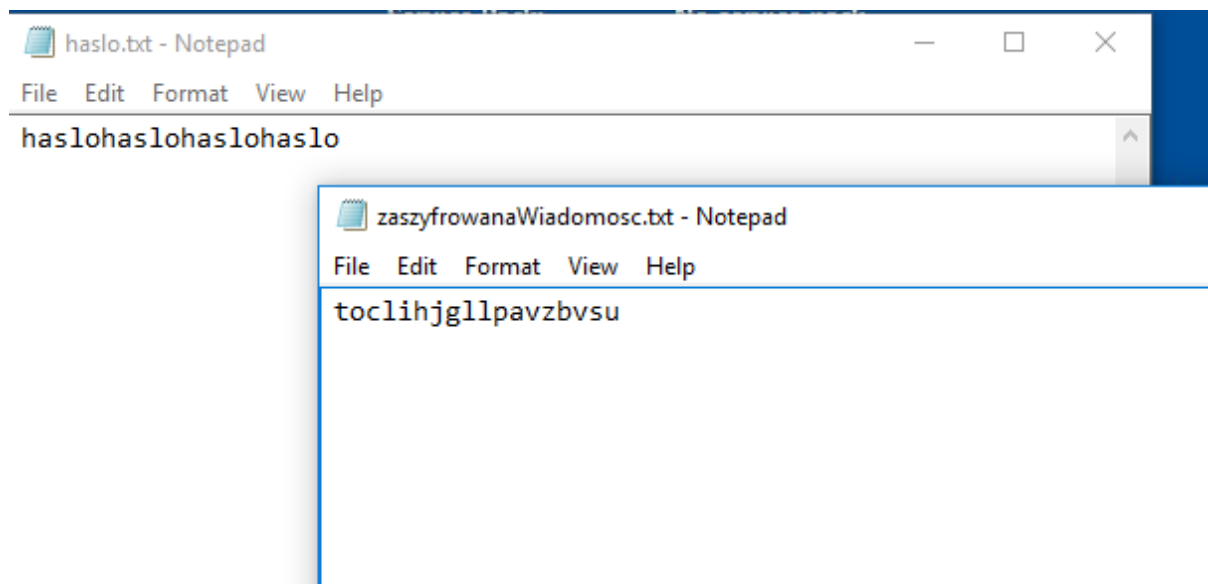
OV = 0 UP = 0 EI = 1 PL = 1
ZR = 0 AC = 0 PE = 0
CY = 0

Memory 1

0x00AC5166	74	61	6a	6e	tajn	0x00AC5A69	6d	7a	00	00	00	00	00	00	m.....
0x00AC516A	65	70	72	7a	eprz	0x00AC5A71	00	00	00	00	00	00	00	00
0x00AC516E	65	7a	70	6f	ezpo	0x00AC5A79	00	00	00	00	00	00	00	00
0x00AC5172	75	66	6e	65	ufne	0x00AC5A81	00	00	00	00	00	00	00	00
0x00AC5176	0d	0a	00	00	0x00AC5A89	00	00	00	00	00	00	00	00
0x00AC517A	00	00	00	00	0x00AC5A91	00	00	00	00	00	00	00	00
0x00AC517E	00	00	00	00	0x00AC5A99	00	00	00	00	00	00	00	00
0x00AC5182	00	00	00	00	0x00AC5AA1	00	00	00	00	00	00	00	00
0x00AC5186	00	00	00	00	0x00AC5AA9	00	00	00	00	00	00	00	00
0x00AC518A	00	00	00	00	0x00AC5AB1	00	00	00	00	00	00	00	00
0x00AC518E	00	00	00	00	0x00AC5AB9	00	00	00	00	00	00	00	00
0x00AC5192	00	00	00	00	0x00AC5AC1	00	00	00	00	00	00	00	00
0x00AC5196	00	00	00	00	0x00AC5AC9	00	00	00	00	00	00	00	00
0x00AC519A	00	00	00	00	0x00AC5AD1	00	00	00	00	00	00	00	00
0x00AC519E	00	00	00	00	0x00AC5AD9	00	00	00	00	00	00	00	00
0x00AC51A2	00	00	00	00	0x00AC5AE1	00	00	00	00	00	00	00	00

Memory 3 Memory 1

Kolejny obieg pętli- kolejny znak w naszej tablicy wynikowej



Na końcu dostajemy dwa pliki – jeden z kluczem, drugi z zaszyfrowaną wiadomością.

7. Podsumowanie

Udało się zrealizować wszystkie założenia projektu biorąc pod uwagę ograniczenia nałożone przy wstępnym projektowaniu funkcjonalności aplikacji.

Możliwe dalsze modyfikacje:

-dodanie modułu losowania klucza za pomocą ziarna, dołączając procedurę losującą, w praktyce taki kod:

```
mov EAX, rin
sub EAX, 2
INVOKE wsprintfA, OFFSET solutionBuffer, OFFSET solutionText, EAX
add ESP, 12
mov rinp, EAX

mov EBX, rin
sub EBX, 2
mov pom3, EBX
mov EBX, 0

;moduł losowania

.WHILE EBX < pom3 ; losowanie tyle liter ile ma wiadomość do zakodowania

    call GetTickCount
    push EAX
    call nseed
    push zakres
    call nrandom
    mov wylosowanaLiczba, EAX
    add EAX, 97
    mov klucz[EBX], EAX

    INC EBX

.ENDW
```

Wraz ze zmiennymi i procedurami:

```
GetTickCount PROTO ;random
nseed PROTO :DWORD ;random
nrandom PROTO :DWORD ;random
.data
    solutionText      BYTE "Wylosuje klucz, minimum %i znaków", 0Ah, 0Dh ,0

    rozmiar4 BYTE $-textWybor
    wybor DD 0
    zakres DWORD 25
    wylosowanaLiczba DWORD 100
    rozmiar3 DD $ - solutionText
    solutionBuffer BYTE 255 dup ( 0 )
```

- dodanie procedury pozwalającej zaszyfrować duże litery i znaki specjalne,
- zapisywanie wiadomości w formacie binarnym, trudniejszym dla odczytu
- graficzny interfejs użytkownika, który jest bardziej estetyczny przy spełnianiu funkcjonalności aplikacji,
- dodanie modułu pozwalającego wybrać metodę szyfrowania np. metodę XOR czy innego szyfrowania.