Raport Laboratorium 4

Organizacja i Architektura komputerów

Autor: Uladzimir Kaviaka (257276)

Cel laboratorium:

Zadanie polegało na napisaniu algorytmu steganografii, przyjęto następujące założenia:

- Wczytanie/zapisanie pliku BMP realizowane w C
- Algorytm kodowania/dekodowania napisany w asemblerze
- **Kodowanie odbywa się na dwóch najmłodszych bitach.**

Opis algorytmu:

Wczytujemy lokalizację pliku od użytkownika, następnie jeżeli użytkownik chce zakodować wiadomość to wczytujemy również wiadomość i kodujemy co tworzy nowy plik BMP z zakodowanym tekstem, natomiast jeżeli użytkownik chce dekodować to również najpierw wprowadza lokalizację pliku zaszyfrowanego po czym wybiera opcję deszyfrowania.

Implementacja programowa:

1. Nagłówki BMP

Każdy plik BMP ma nagłówek, to są 54 bajty, w tym nagłówku umieszczone rożne informacje odnośnie pliku BMP, pod adresem 0x000A jest umieszczony adres gdzie zaczynają się pixele, pod adresem 0x0012 jest przechowywana szerokość obrazku, pod adresem 0x001C przechowywany parametr jako ilość bitów na pixel.

Poszczególne informacje odnośnie danych w nagłówku są zawarte tutaj: https://pl.wikipedia.org/wiki/Windows_Bitmap

2. Wczytanie pliku BMP i tworzenie tablicy pixeli

Kolejnym etapem programu trzeba było zrobić tablice pixeli na której będziemy operować w przyszłości. Ale najpierw otwarcie pliku i wczytywanie nagłówka, do przemieszczania się po pliku BMP będą służyły funkcje **fseek** oraz **fread**

fseek – przechodzi do określonej pozycji w pliku

fread – odczytuje z pozycji określonej za pomocą fseek.

Więc najpierw wczytujemy z nagłówka parametry jako adres rozpoczęcia pixeli, długość, szerokość, bitów na pixel, następnie obliczamy ile bajtów jest na jeden pixel z założenia że 1 bajt = 8 bit. Następnie alokujemy pamięć gdzie będą umieszczone nasze pixele. Po czym w pętli odczytujemy zawartość pliku BMP i zapisujemy w naszej tablicy pixeli kolejne wiersze pod adresem początku pixeli. Po czym zamykamy nasz plik za pomocą funkcji **fclose.** Poniżej przedstawiono kod który wykonuje wczytywanie BMP.

```
FILE *image = fopen(filelocation,"rb");

int dataOffset;
fseek(image, DATA_OFFSET_OFFSET, SEEK_SET);
fread(&dataOffset, 4, 1, image);

fseek(image, WIDTH_OFFSET, SEEK_SET);
fread(&width, 4, 1, image);

fseek(image, HEIGHT_OFFSET, SEEK_SET);
fread(&height, 4, 1, image);

short bitsPerPixel;
fseek(image, BITS_PER_PIXEL_OFFSET, SEEK_SET);
fread(&bitsPerPixel, 2, 1, image);

int bytesPerPixel = ((int)bitsPerPixel) / 8;
int paddedRowSize = (int)(4 * (int)(((float)(width) / 4.0f) + 1))*bytesPerPixel;
int unpaddedRowSize = width*bytesPerPixel;

int totalSize = unpaddedRowSize*height;

byte* pixels = (byte*)malloc(totalSize);
byte* currentRowPointer = pixels + (height-1)*unpaddedRowSize;  //wskaznik na ost
for(int i=0;i<height;i++) {
    fseek(image, dataOffset+(i*paddedRowSize), SEEK_SET);
    fread(currentRowPointer, 1, unpaddedRowSize, image);
    currentRowPointer -= unpaddedRowSize;
}
fclose(image);
```

Po tym jak wczytaliśmy plik, ja robię następną rzecz, najpierw zapisuje w pliku BMP długość tekstu jaką użytkownik chce zakodować, aby po deszyfrowaniu można było spod znanego adresu(pierwsze 4 bajty pixeli) odczytać długość i już było wiadomo ile bajtów musimy dekodować.

```
unsigned long arr = 0;
long encodedArr = 0;

int sizeMsg = strlen(message)-1;
int j = 1;

arr = pixels[0];        //pierwszy bajt pixela
arr *= 256;                 //rownowazne przesunieciu w HEX (00c9->c900)
arr += pixels[1];       //drugi bajt pixela
arr *= 256;
arr += pixels[2];       //trzeci bajt pixela
arr *= 256;
arr += pixels[3];       //czwarty bajt pixela

encodedArr = encoder(arr, sizeMsg); //kodujemy dlugosc ciagu
arr+=0;
```

Tutaj za pomocą funkcji **strlen** zapisujemy w sizeMsg długość tekstu użytkownika po czym musimy wybrać gdzie to chcemy zakodować. W moim przypadku koduję długość na początku tablicy pixeli. Musze pobrać teraz 4 bajty z tablicy pixeli bo przyjąłem kodowanie na dwóch najmłodszych bitach, oraz długość nie większa niż 255 (1 bajt).

Więc jeżeli kodujemy na dwóch najmłodszych bitach pixeli, to żeby zakodować 1 bajt(8 bit) potrzebujemy w jednym bajcie pixeli możemy zakodować 2 bity informacji, więc potrzebujemy 4 bajty. Bo 2 najmłodsze * 4 = 8 najmłodszych co wystarczy żeby zakodować 1 bajt. Więc pobieram kolejne bajty pixeli i wykonuje mnożenie razy 256 aby przesunąć o dwie pozycje w lewo. Po tych operacjach w **arr** znajdują się 4 bajty z tablicy pixeli. Po czym występuje wywołanie funkcji asemblera która koduje ale o tej funkcji później. Po zakodowaniu funkcja asemblera zwraca zakodowany ciąg więc teraz aktualizujemy naszą tablicę pixeli.

```
pixels[3] = encodedArr%256;
encodedArr = encodedArr>>8;
pixels[2]= encodedArr%256;
encodedArr = encodedArr>>8;
pixels[1] = encodedArr%256;
encodedArr = encodedArr>>8;
pixels[0] = encodedArr%256;
```

Tutaj wykonując % - reszta z dzielenia pobieramy kolejne bajty zakodowanej tablicy, przy pobraniu bajtów aby przejść do kolejnego wykonujemy przesunięcie w prawo (8 znaczy 1 bajt).

3. Kodowanie tekstu

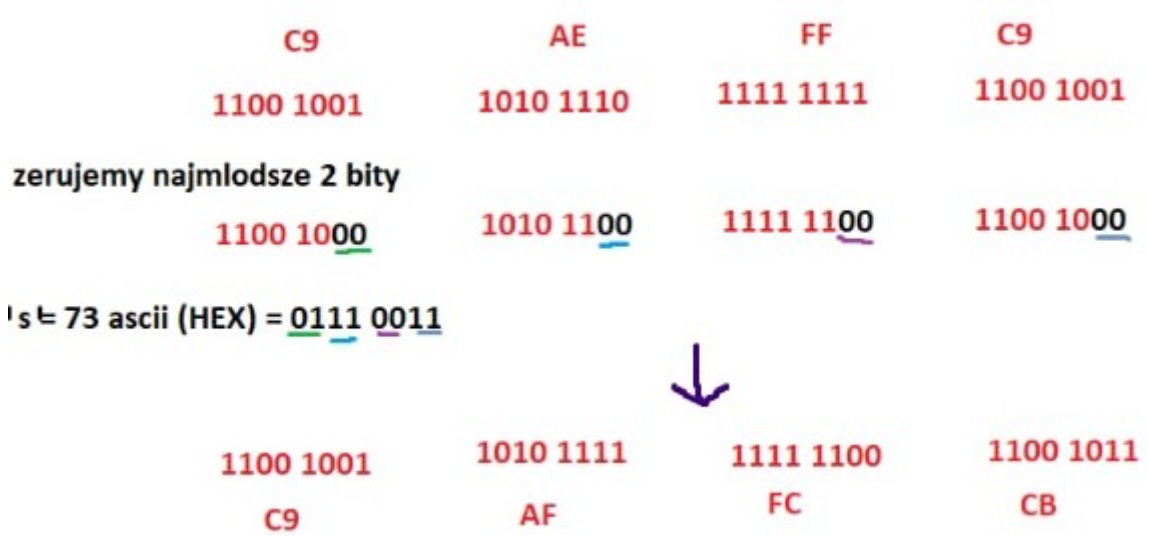
Algorytm ten sam co przy kodowaniu długości.

```
for(int i=10368;i<height*unpaddedRowSize;i+=4) {
    if(j-1>sizeMsg) break;

    arr = pixels[i];        //pierwszy bajt pixela
    arr *= 256;                 //rownowazne przes
    arr += pixels[i+1];     //drugi bajt pixela
    arr *= 256;
    arr += pixels[i+2];     //trzeci bajt pixela
    arr *= 256;
    arr += pixels[i+3];     //czwarty bajt pixela

    encodedArr = encoder(arr, message[j-1]); //fun
    j++;

    pixels[i+3] = encodedArr%256;   //dzielenie pr
    encodedArr = encodedArr>>8;     //przesuwamy o
    pixels[i+2]= encodedArr%256;   //kolejne dwie
    encodedArr = encodedArr>>8;
    pixels[i+1] = encodedArr%256;
    encodedArr = encodedArr>>8;
    pixels[i] = encodedArr%256;

    encodedArr = 0;
}

WriteFile(pixels,width,height,bytesPerPixel);
```

Tutaj początek zapisywania tekstu do tablicy pixeli zaczyna się od pozycji 10368, bo sprawdziłem w GDB że do tej pozycji elementy

tablicy pixeli są zerami. Po czym pobieramy 4 bajty pixeli, wywołujemy funkcje asemblera która koduje oraz zwraca zakodowany ciąg, po czym zapisujemy do naszej tablicy pixeli zakodowany ciąg bo na podstawie tej tablicy będę tworzył zakodowany BMP.

Algorytm kodowania jest następujacy:



Na górze mamy nasze 4 bajty pixeli. Po czym zerujemy dwa najmłosze bity na których będziemy zapisywali naszą informację. Następnie chcemy zakodować powiedzmy literkę ‘s’ (73 HEX). Jej kod dwójkowy jest przedstawiony na obrazku, po czym umieszczamy na kolejne pozycje bajtow pixeli kod naszej liczby i otrzymujemy zakodowany ciąg który jest na dole. Ten ciąg wpisujemy do tablicy pixeli(nadpisując) która poźniej będzie użyta do utworzenia zakodowanego BMP.

Po czym wywołujemy funkcję **WriteFile** która stworzy na podstawie tablicy pixeli oraz wczytanych nagłówków do tej pory, zakodowany obrazek BMP. Funkcję **WriteFile** wziąłem jako gotową w internecie więc nie będę umieszczał w raporcie.

4. Wywołanie funkcji asemblera do kodowania

Najpierw jeżeli chcemy wywołać funkcje z asemblera musimy umieścić w kodzie C nagłówek tej funkcji:

```
extern unsigned long encoder(unsigned long n,unsigned long msglength);
```

Następnie gdy wywołujemy funkcje asemblera wszystkie parametry są przekazywane przez stos.

Funkcja kodująca jeden bajt:

Najpierw wywołując funkcje asemblera z poziomu języka C trzeba zrobić kolejne kroki:

Tworzenie ramki stosu: najpierw wrzucamy wskaźnik na stos, po czym przypisujemy mu wartość wskaźnika na obecny szczyt stosu.

```
encoder:

push %ebp
mov %esp, %ebp

#pobieramy ze stosu nasze

mov 8(%ebp), %eax   #4 ba
mov 12(%ebp), %ecx  #zapi

#pierwszy bajt kodujemy
mov %al, %bl
and $0b11111100, %bl    #
mov %cl, %dl
and $0b00000011, %dl    #
add %dl, %bl            #
mov %bl, %al
```

Następnie znając gdzie jest podstawa stosu możemy odczytywać przekazane parametry, tutaj do **eax** wczytuje 4 bajty pixeli zaś do **ecx** wczytuje jedna literkę którą trzeba zakodować.

Następnie patrząc na rysunek który tłumaczy kodowanie robię to w asemblerze, czyli przemieszczamy **al** (najmłodszy bajt rejestru **eax**) do **bl** po czym zerujemy dwa najmłodsze używając and, następnie pobieramy **cl**(najmłodszy bajt **ecx**) do **dl** i zerujemy wszystko w **dl** oprócz dwóch ostatnich bo kodujemy dwa bity na dwóch bitach. Po czym dodajemy do siebie **dl** i **bl** i już jest zakodowany pierwszy bajt więc wrzucamy go z powrotem do **al.** W analogiczny sposób kodujemy pozostałe używając przesuwań (**shr/shl**). Ale po zakodowaniu dwóch bajtów (rejestry **al, ah**) musimy przejść do kolejnych, więc ja rozwiązałem w ten sposób że zamieniam starszą i młodszą cześć rejestru **eax**.

```
mov %ax, %bx
shr $16, %eax
mov %ax, %dx
mov %bx , %ax
shl $16, %eax
mov %dx , %ax
```

Tutaj nie ma nic skomplikowanego zapisujemy młodszą część rejestru **eax** po czym przesuwamy **eax** 16 pozycji w prawo co daje że teraz starsza część jest na pozycji młodszej po czy zapisujemy starszą część w innym rejestrze, następnie wstawiamy młodszą cześć przesuwamy na pozycję starszej po czym na pozycję młodszej części wstawiamy zapisaną wartość starszej po tej operacje mamy zamienione części LOW i HIGH. Po czym możemy zakodować następne 2 bajty. Następnie w analogiczny sposób mając zakodowane 4 bajty, powracamy na swoje miejsca LOW i HIGH. Po czym usuwamy ramkę stosu. Wynik mamy całkowity, więc jest zwracany w rejestrze **eax** gdzie właśnie znajduje się nasz zakodowany ciąg.

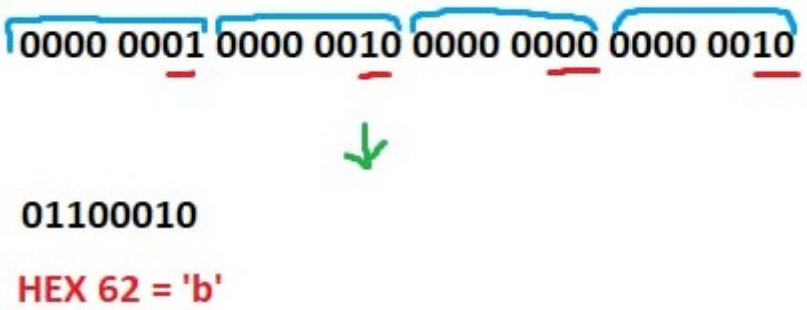
5. Deszyfrowanie

Dla deszyfrowania najpierw odczytujemy plik zgodnie z tym jak opisano w punkcie 2. Po czym pobieramy nasze pierwsze 4 bajty pixeli gdzie jest zakodowana długość i następnie mając długość w pętli pobieramy kolejne 4 bajty pixeli i dekodujemy jeden symbol i dodajemy do tablicy symboli, po pętli będziemy mieli już dekodowany ciąg który spokojnie wypisujemy do konsoli.

```
arr = pixels[0];        //pierwsz
arr *= 256;                 //row
arr += pixels[1];       //drugi b
arr *= 256;
arr += pixels[2];       //trzeci
arr *= 256;
arr += pixels[3];       //czwarty


long x = 0;
long length = decoder(arr);
// Konwertowanie z postaci szesna
int factLength = 1*length % 256;
length /= 256;
factLength += 4* length%256;
length /= 256;
factLength += 16*length%256;
length /= 256;
factLength += 64*length%256;
```

Tutaj pobieramy pierwsze 4 bajty, wywołujemy funkcję asemblera do dekodowania przekształcamy z postaci jak pokazano niżej do HEX:

0x01020002 **wynik po deszyfrowaniu**



Czyli dekodowanie polega na tym, że z tego powodu że koduję na dwóch najmłodszych bitach, to żeby deszyfrować zerujemy w każdym bajcie wszystkie bity oprócz dwóch ostatnich, następnie te bity dają naszą zakodowaną literkę. Następnie pamiętamy że ja zacząłem kodować tekst od 10368 pozycji w tablicy pixeli więc zaczynamy odczytywać pixeli też z tej pozycji.

```
for(int i = 10368; i< (factLength*4+10368) + 4;i+=4) {
    arr = pixels[i];        //pierwszy bajt pixela
    arr *= 256;                 //rownowazne przesunieci
    arr += pixels[i+1];     //drugi bajt pixela
    arr *= 256;
    arr += pixels[i+2];     //trzeci bajt pixela
    arr *= 256;
    arr += pixels[i+3];     //czwarty bajt pixela

    decoded = decoder(arr);
```

Po czym znów przekształcam na HEX, jak pokazano na rysunku wyżej.

6. Deszyfrowanie w asemblerze

Konwencja wywołania jest taka sama, jak było opisano w punkcie 4. Więc tutaj wytłumaczę tylko dekodowanie. Jak było wspomniano wyżej musimy dla każdego z czterech bajtów wyzerować wszystkie bity oprócz dwóch ostatnich:

```
decoder:
push %ebp
mov %esp, %ebp

mov 8(%ebp), %eax   #4 baj

#pierwszy bajt dekodujemy
mov %al, %bl
and $0b00000011, %bl    #z
mov %bl, %al
```

Znów pobieramy najmłodszy bajt **eax** czyli **al,** zerujemy bity oprócz dwóch ostatnich i wstawiamy z powrotem, analogicznie robimy pozostałe bajty, oraz pamiętając żeby zakodować drugą część czyli 2 ostatnie bajty musimy zamienić znów część HIGH i LOW rejestru **eax** dokonać deszyfrowania tych dwóch bajtów po czym powrócić na swoje miejsca HIGH i LOW (Analogicznie jak w kodowaniu). Na koniec usuwamy ramkę stosu.

Wnioski:

Najpierw zadanie pokazało się bardzo trudnym, ale próbując coś robić zawsze okazywało się że jestem co krok bliżej, nie wyobrażam jak zrobiłbym to zadanie nie korzystając z debuggera GDB bo dostawałem bardzo dużo błędów podczas pisania tego programu. Spędziłem szczerze mówiąc z tym zadaniem co najmniej 20 godzin, ale nauczyłem się sporo nowych rzeczy, najważniejszym i najciekawszym dla mnie było wywołanie funkcji ASM z poziomu C oraz przekazywanie/zwracanie parametrów. Nagłówki BMP też były bardzo ciekawą rzeczą ale na wikipedii jest to wytłumaczone w bardzo zrozumiały sposób, podczas pisania szyfratora już gdy stworzyłem plik zaszyfrowany sprawdzałem za pomocą programu na windows (**Frhed)** czy faktycznie zakodowałem to co jest potrzebne. Szyfrowanie/deszyfrowanie nie było trudno napisać w asemblerze więc najtrudniejszym w tym zadaniu, moim zdaniem, jest uwierzyć że to nie jest trudne.