

Proszę się nie bać. To tylko luźny materiał, który pokazuje co to jest inżynieria wsteczna i asembler.
A nóż kogoś zaciekawi 😊

x86 – nazwa architektury procesorów opracowanej przez firmę Intel. Mimo, że ma ona już bardzo dużo lat jest wciąż najpopularniejsza w rozwiązaniach przeznaczonych dla komputerów PC

Instrukcja procesora – procesor wykonuje instrukcje jedna po drugiej. Każda architektura ma swój zestaw instrukcji. Instrukcję może być np. liczba 0xABCD98 albo liczba 67 (tak, instrukcja std::cout<<"Hello World"; jest wcześniej czy później liczbą a w dalszej konsekwencji zbiorem 0 i 1).

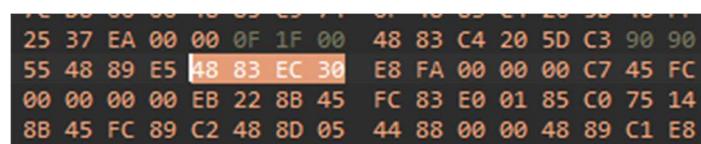
Kompilacja – proces zmiany kodu czytelnego dla człowieka zapisanego w konkretnym języku na instrukcje procesora.

Asembler – Jest to czytelny dla człowieka zapis instrukcji procesora.



```
push    rbp
mov     rbp, rsp
sub    rsp, 30h
call   __main
mov    [rbp+var_4], 0
jmp    short loc_140001818
```

Obrazek przedstawia asembler platformy x86



7E	33	00	00	10	03	C9	77	01	10	03	C7	20	30	10	11
25	37	EA	00	00	0F	1F	00	48	83	C4	20	5D	C3	90	90
55	48	89	E5	48	83	EC	30	E8	FA	00	00	00	C7	45	FC
00	00	00	00	EB	22	8B	45	FC	83	E0	01	85	C0	75	14
8B	45	FC	89	C2	48	8D	05	44	88	00	00	48	89	C1	E8

Instrukcje procesora – zaznaczony fragment

odpowiada fragmentowi zaznaczonemu

na obrazku wyżej

Można powiedzieć, że asembler jest pierwszym językiem, w którym powstały kolejne. Gdy te kolejne języki osiągnęły określony poziom sam czysty asembler stracił na wartości, ale wciąż wykorzystywany jest do np. inżynierii wstecznej. Dlaczego?

Kompilując program napisany np. w C++ uzyskujemy plik EXE, który zawiera instrukcje procesora. Odzyskanie oryginalnego kodu w 100% jest nie możliwe, ale da się za pomocą paru narzędzi uzyskać kod w C/C++ zbliżony do oryginału. Można też obrać inną ścieżkę i zapisane w EXE instrukcje procesora przetłumaczyć na instrukcje czytelne dla człowieka czyli na instrukcje tekstowe asemblera. Taki kod jest już jak najbardziej do przeanalizowania.

Do przeprowadzenia poniższego eksperymentu potrzebny nam kompilator C++ do komplikacji przykładowego kodu oraz program IDA Freeware dostępny za darmo w sieci.

Zakładam, że programy już mamy. Oto przykładowy program w C++:

```
#include <cstdio>

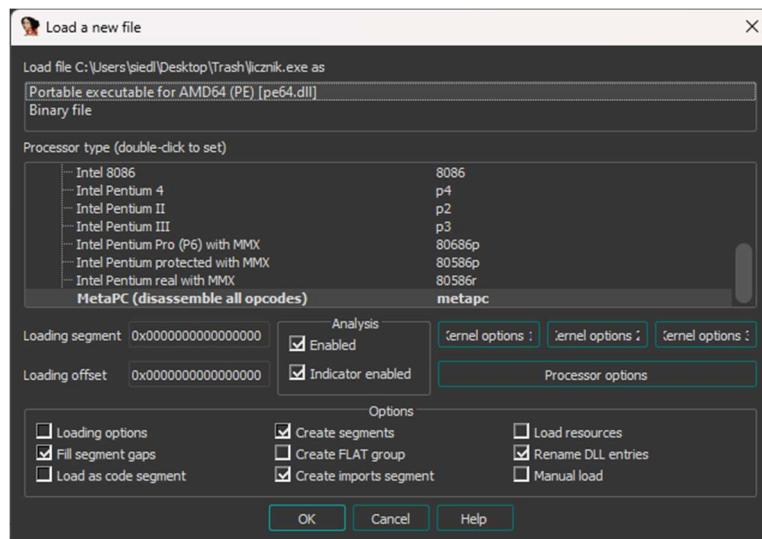
int main(){
    for(int i = 0;i<100;i++){
        if(i % 2 == 0){
            printf("%d is nice number\n", i);
        }
    }
}
```

Program wypisuje liczby parzyste w przedziale od 0 do 99. Po skompilowaniu i uruchomieniu dostajemy taki output:

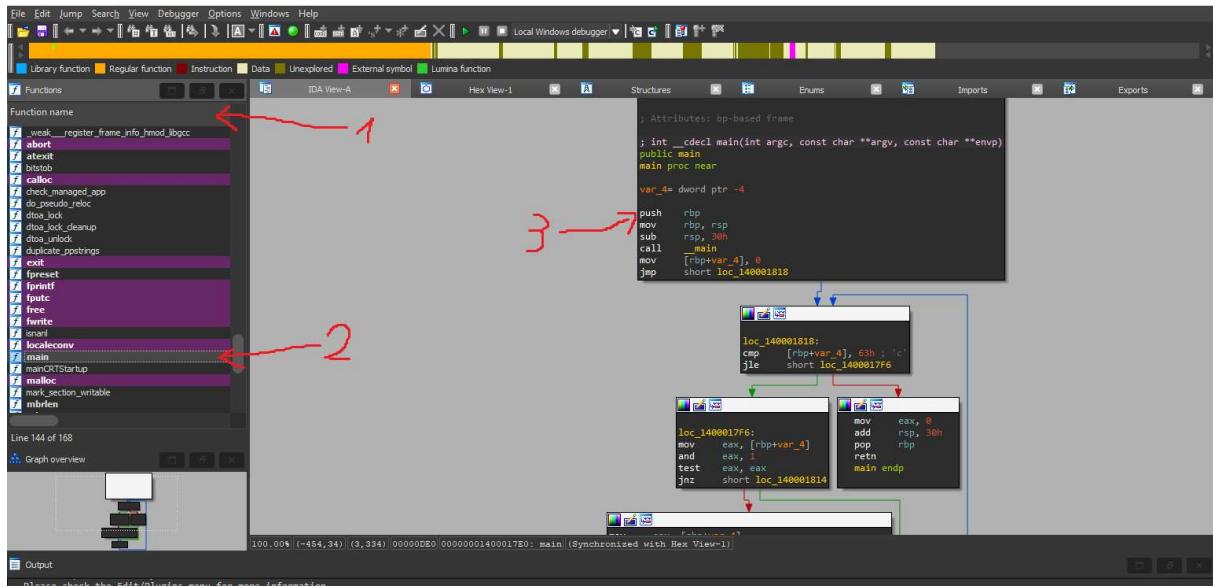
```
84 is nice number
86 is nice number
88 is nice number
90 is nice number
92 is nice number
94 is nice number
96 is nice number
98 is nice number
```

Wyobraźmy sobie, że nie mamy kodu źródłowego programu, a potrzebujemy by program ten wykonywał cały proces w przedziale od 0 do 120 (no do 119).

Z pomocą przychodzi nam IDA i inżynieria wsteczna. Uruchamiamy program. Wybieramy opcję **New** i wskazujemy nasz plik EXE z programem.



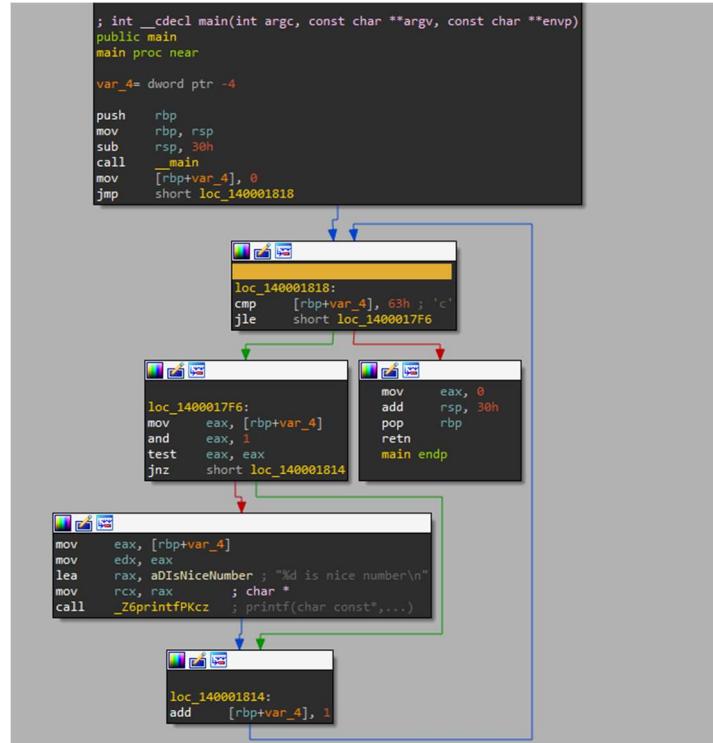
W kolejnym oknie zostawiamy wszystko tak jak jest i klikamy **OK**.



1. List funkcji w programie (tak – program by mógł działać np. w Windows potrzebuje wielu funkcji, nie tylko tych, które sami napisaliśmy)
2. Na liście funkcji znajduje się nasza funkcja **main**
3. Pierwsza instrukcja funkcji main.

Główna część okna to widok kodu w formie grafu (jeżeli widzimy inny widok np. tekstowy to możliwość przełączenia się na widok grafowy jest pod prawym przyciskiem myszy).

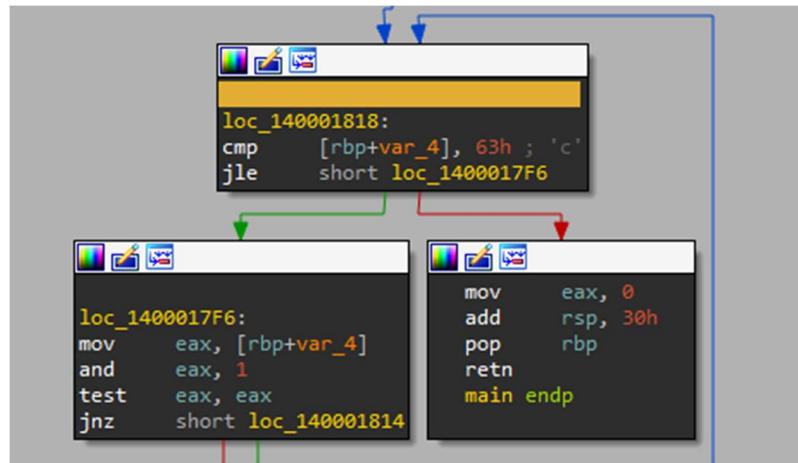
Zobaczmy całą funkcję main:



Widok grafu jest o tyle fajny, że pokazuje w bardzo czytelny sposób przebieg działania programu:

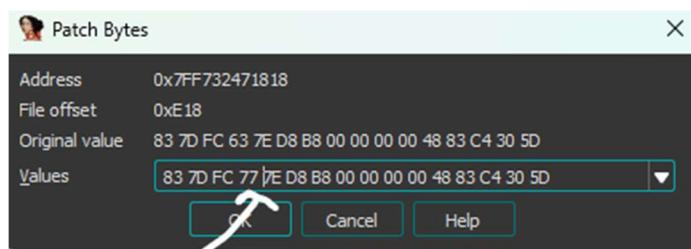
- Spodziewamy się w programie pętli – na grafie widzimy ścieżkę biegnącą w kółko oraz jeden blok, z którego nie wychodzi strzałka
- Blok, z którego nie wychodzi strzałka to ostatni blok w funkcji – instrukcja **ret** też sugeruje powrót (jest to odpowiednik returna)
- Na grafie widzimy dosyć wyraźnie kod wypisujący tekst na ekranie – istnieje ścieżka omijająca ten blok co sugeruje, że jest to blok reprezentujący ciało **ifa**

Pętla for tak jak i instrukcja warunkowa opiera się na pewnym warunku logicznym. W asemblerze testy logiczne można wykonać na wiele sposobów. Zobaczmy drugi bloczek od góry:

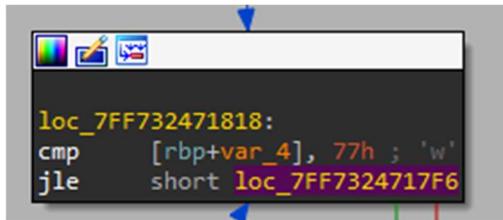


cmp brzmi jak skrót od **compare** i faktycznie nim jest. Następuje tu porównanie zmiennej znajdującej się pod adresem **rbp+var_4** ze stałą **0x63**. Stała ta to nasze 99 tylko zapisana w systemie szesnastkowym. Instrukcja **cmp** w praktyce wykonuje odejmowanie (od pierwszego argumentu odejmuje drugi). Następna instrukcja **jle** oznacza **jump if less or equal**. Jeżeli wartość naszego **i**, które znajduje się pod adresem **rbp+var_4** pomniejszona o 99 jest mniejsza bądź równa 0 (**jle**) to skocz do etykiety **loc_1400017F6**. W przeciwnym razie idź dalej. Należy pamiętać, że kod w rzeczywistości nie jest podzielony na bloczki (tutaj korzystamy z takiej reprezentacji dla wygody). Instrukcja **mov eax, 0** znajduje się bezpośrednio po **jle short loc_1400017F6**. Skoki są podobne do instrukcji **goto** z C++.

Z całej tej analizy wynika, że do osiągnięcia celu (pętla do wartości 119) należy podmienić wartość **63h** na wartość **77h** (czyli 119 dziesiętnie). Zaznaczamy całą linię z instrukcją **cmp**. Następnie klikamy menu **Edit > Patch program > Change byte**



Zmieniamy wartość 63 na 77 i klikamy ok.



Bloczek z warunkiem zmienił się.

W menu **Edit > Patch program** wybieramy **Apple patches to input file** i zatwierdzamy okienko. Teraz uruchamiamy nasz plik exe.

```
100 is nice number
102 is nice number
104 is nice number
106 is nice number
108 is nice number
110 is nice number
112 is nice number
114 is nice number
116 is nice number
118 is nice number
```

Efekt

Oczywiście cała sprawa była prosta, ponieważ modyfikowaliśmy tylko stałą, którą łatwo było wypatrzyć w kodzie. Gorzej sprawa miała by się gdybyśmy chcieli modyfikować dużo większy program, którego kompletnie nie znamy (jak widać poza **main** nie zachowały się tutaj, żadne nazwy własne z oryginalnego kodu), albo chcielibyśmy zmieniać same instrukcje.

Może to kogoś zaciekawi 😊