Assembly - 2

Serinin ilk yazısında Assembly dili üzerine öğrenilen temel yapıları öğrenmiştik. Bu yazıda öğrenilen bilgileri pekiştirmek amacıyla C/C++ üzerinde hazırladığım birkaç kod parçasını Reverse Engineering sürecinin vazgeçilmez araçlarından olan x64 dbg ve IDA Free üzerinde incelemeye çalışılacaktır.

İlk adımda birçok kişinin programlamaya başlarken yazdığı ilk komut olan konsola "Hello World" yazdıran programı inceleyelim. C dili kullanılarak yazıldığında tek satırda "**printf();**" komutuyla konsola yazılması sağlanabiliyor.

```c
#include <stdio.h>


int main(){

    printf("Hello World");

    return 0;
}
```

Kaynak kod derlendikten sonra IDA aracıyla açıldığında program içeriği aşağıdaki gibi görülmektedir.

```
; Attributes: thunk

; int __fastcall main(int argc, const char **argv, const char **envp)
main proc near
jmp     main_0
main endp
```

- Programın başında bulunan "**proc near**" ve sonunda bulunan "**endp**" tanımları bir fonksiyonun (Burada Main fonksiyonu) başlangıç ve bitiş kısımlarını tanımamak için kullanılıyor.
- Programda "**jmp main_0**" komutunun bulunduğu satıra tıklanarak Main fonksiyonunun içeriğine giriş yapılıyor.

```
; Attributes: bp-based frame fpd=0D0h

; int __fastcall main_0(int argc, const char **argv, const char **envp)
main_0 proc near
push    rbp
push    rdi
sub     rsp, 0E8h
lea     rbp, [rsp+20h]
lea     rcx, unk_140021008
call    j___CheckForDebuggerJustMyCode
lea     rcx, aHelloWorld ; "Hello World"
call    sub_140011195
xor     eax, eax
lea     rsp, [rbp+0C8h]
pop     rdi
pop     rbp
retn
main_0 endp
```

- Main fonksiyonunun içerisine girildiğinde ilk dört ve son dört satırda bulunan komutlarla çok sık karşılaşacağız. Burada kullanılan komutlar **Calling Convention** olarak biliniyor. Calling Convention, **Prologue** ve **Epilogue** olmak üzere iki kısımdan oluşuyor.
    - **Prolouge**, program çalıştırılmaya başlamadan önce çalıştırma ortamını ayarlamak için kullanılan komutlar bütünü olarak tanımlanabilir. Bu ayarlamaya örnek olarak program içerisinde kullanılmak üzere (değişkenlerin değerlerini, adresleri depolamak için vs.) Stack üzerinde 232 (0xE8) baytlık alan ayrılıyor.
    - **Epilogue**, Prologue sürecinde program çalıştırılmadan önce oluşturulan çalıştırma ortamını geri düzenlemek/eski haline getirmek için kullanılan komutlar bütünü olarak tanımlanabilir.
        - Bu kısımda kullanılan "**[ ]**" semboller dikkat çekebilir. Bu semboller, içerisindeki değişkenin/Register'ın bellek adresi üzerinde işlem yapılırken kullanılıyor. Yani "lea rbp, [rsp+20h]" komutuyla aslında RBP adresi RSP adresinin 20h kadar uzağına taşındığı söylenebilir.
- Prologue aşamasından sonraki ilk komut satırında dallanma öncesinde RCX Register'ının adresi sonraki satırında bulunan dallanma komutuyla dallanılacak adreste kullanılacak olan bir değerin adresine set ediliyor. Zaten sonraki satırında da kodun kontrol edilmesi için kullanılan bir çağrı görülüyor.
    - Sonraki kısımlarda da dallanma komutu öncesinde RCX Register'ının adresinin değiştirildiğini sık görmeye başlayacağız.
- Ardından "**lea rcx, aHelloWorld**" komutuyla, RCX Register'ının adresi "Hello World" metninin yazılı olduğu adrese taşınıyor. Ardından "**call sub_1400111195**" komutuyla konsola yazdırılmak üzere alt fonksiyona dallanılıyor.

Memory Management in C Programming

Bir programı daha anlaşılır analiz edebilmek için program çalışmaya başladığında hafıza alanındaki depolanma düzeni hakkında bilgi sahibi olmak gerekiyor. Hafıza alanındaki depolama düzenine bakıldığında 5 farklı alandan bahsedilebilir. Bunlar;

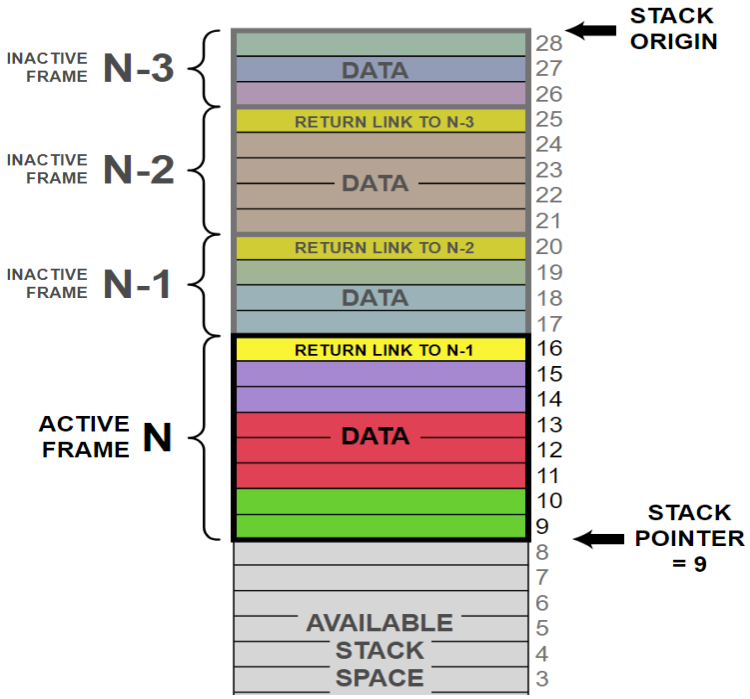
- **Heap**, hafıza alanı üzerinde yer tahsis etmek için kullanılan belirli bir bölümdür. Bu bölüm geliştiriciler tarafından kullanılmaktadır. Program çalıştırıldığında dinamik olarak yer tahsis etmek için kullanılır.

  Heap alanından yer tahsis etmek için "**malloc()**" fonksiyonu kullanılırken, işlem sonunda "**free()**" fonksiyonuyla farklı işlemlerde kullanılabilmesi adına tahsis edilen bu yer serbest bırakılır (stdlib.h kütüphanesiyle geliyor). Heap üzerinde yer tahsis etme operasyonuna "**dynamic memory allocation**" denilir.

- **Stack**, İşletim sistemlerinde ve programlama dillerinde Stack veri tutmak için kullanılan en temel yapıdır. Bu alan **derleyici tarafından** yönetilir. Örnek olarak C dilinde hazırladığımız programlarda da görüleceği üzere her fonksiyonun Local değişkenleri, bir dallanma işlemi gerçekleşeceğinde geri dönüş adresi gibi daha birçok veri Stack alanında tutulur.

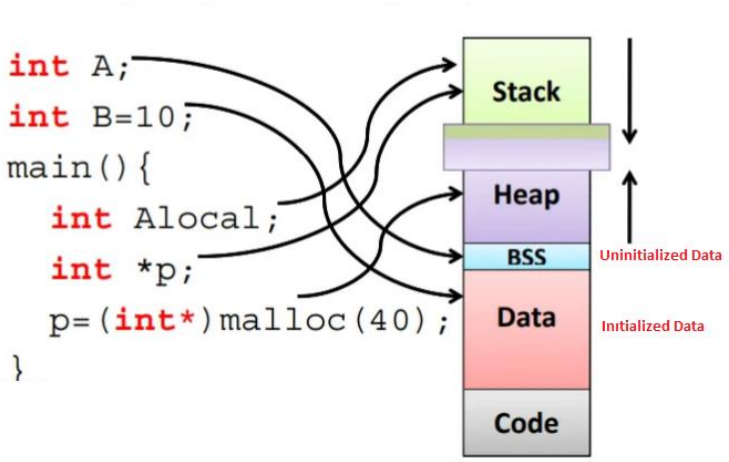
  Stack hafızaya veriler LIFO (Last İn First Out) mekanizmasıyla işlenir. Stack hafıza alanının tepesine veri eklemek için "**push <Variable>**", tepesinden veri çıkarmak için "**pop <Variable>**" komutu kullanılır. RSP Register'ı her zaman Stack yapısının tepesindeki veriyi temsil eder. Veri ekleme ve çıkarma işlemleri de yine RSP Register'ının aşağı ve yukarı hareket etmesiyle gerçekleştirilir.

Stack hafıza çalışma yapısı gereği ters durduğu için her yeni veri eklendiğinde RSP Register'ı negatif yöne doğru hareket edecektir. Bunu yukarıda incelediğimiz HelloWorld programında "**sub rsp, 0E8h**" komutuyla da görebiliriz.

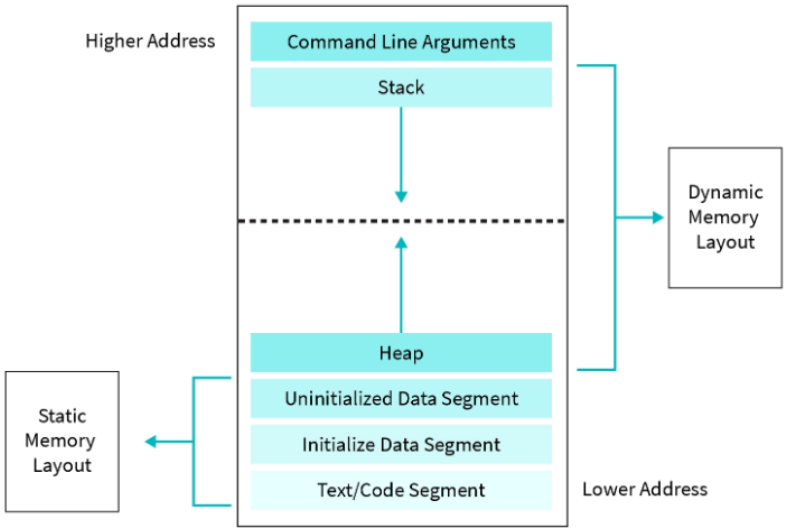


- **Inıtialized Data Segment**, program içerisinde başlangıç değerleri verilerek tanımlanan Static, Global, Constant veya harici değişkenlerin tutulduğu hafıza alanıdır. Bu alan Read-Write ve Read-Only olmak üzere iki kısımda sınıflandırılabilir. Program sürecinde üzerinde değişiklik yapılmasına izin verilecek değişkenler Read-Write alanına depolanırken, Constant anahtar kelimesi kullanılarak tanımlanan değişkenler Read-Only alanında depolanmaktadır.

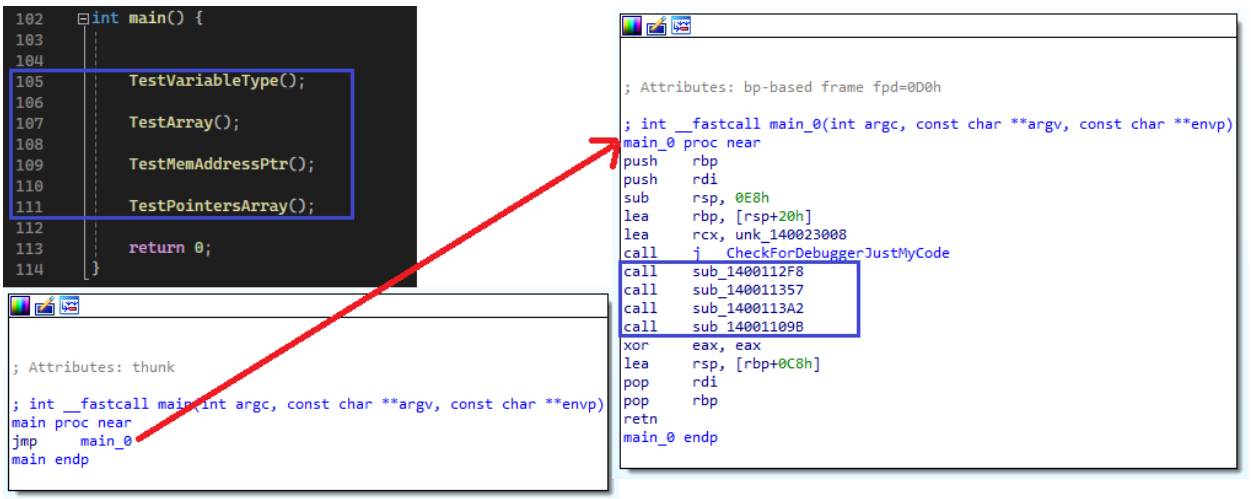
- **Uninitialized Data Segment**, program içerisinde başlangıç değerleri verilmeden tanımlanan Static, Global veya harici değişkenlerin tutulduğu hafıza alanıdır.



- **Code/Text Segment**, program derlendikten sonra oluşan ikili dosyaların depolandığı hafıza alanıdır. Bu alan programın yanlışlıkla değiştirilmesini önleyen salt okunur izne sahiptir.

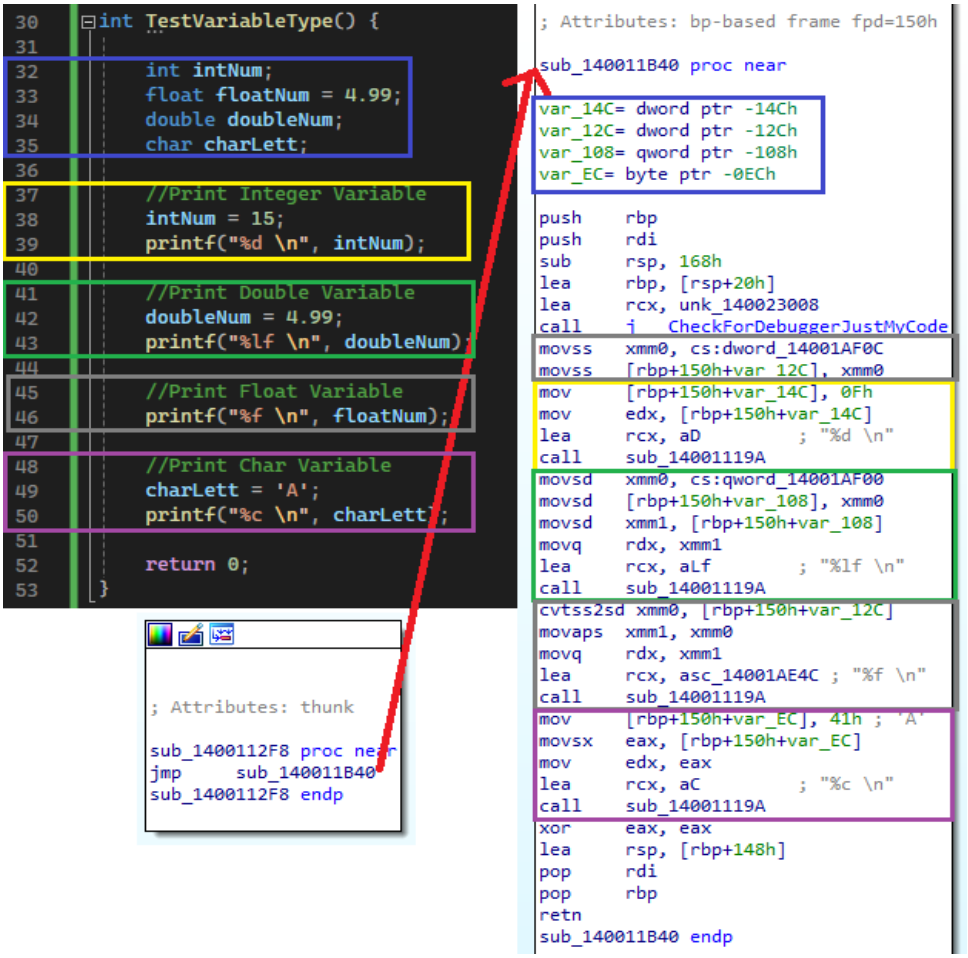


Bir program çalıştırıldığında hafıza alanındaki düzeni hakkında bilgi edindikten sonra programın kaynak kodunda kullanılan temel yapıların nasıl göründüğünü analiz edebiliriz. Bunun için Assembly kodunda nasıl göründüğünü merak ettiğim yapılardan oluşan bir program hazırlayarak derledim. Derlenen dosya bir Disassembler üzerinde açıldığında aşağıdaki gibi görünüyordu. Program çalıştırıldığında Prologue süreci tamamlandıktan sonra tanımladığım 4 fonksiyon için dallanma komutları görülebiliyor. Dallanma komutlarında sonra da klasik Epilogue süreci başlıyordu.



Oluşturduğum ilk fonksiyon içerisinde tanımlanan değişken tiplerinin nasıl görüldüğünü incelemek üzere oluşturuldu. Disassemble edilmiş haline bakıldığında fonksiyon içerisinde tanımlanan değişkenler için bellek alanında RBP'den ne kadar uzakta depolanacağını gösteren tanımlamalar görülüyor (ptr -> Pointer tipinde oluşturulmuş). Örnek olarak var_12C değişkenine bakıldığında -12Ch değeri atanmış. Bu değer Stack üzerindeki Base Pointer (RBP)'dan ne kadar uzağa yazılacağını belirtiyor. Bu tanımlamalar program çalıştırıldığında değişkenin Stack hafıza üzerindeki konumunu tespit edebilmek adına önemlidir. Burada tanımlanan alanları büyüklüğüne bakıldığında;

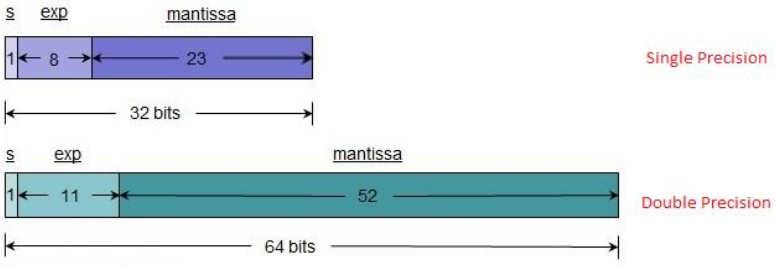
- Integer -> DWORD (32 bit)
- Float -> DWORD (32 bit)
- Double -> QWORD (64 bit)
- Char -> BYTE (8 bit) alan kaplamaktadır.



Prologue süreci sonrasında karşılaşılan ilk komut satırı dikkat çekiyor. Burada Float veya Double tipinde değerlerle işlemler yapıldığı anlaşılıyor. Bu tip değişkenler kayan noktalı sayılar olarak biliniyor. X86-66 mimarisinde kayan noktalı sayılarla işlem yapabilmek için SSE (Streaming SIMD (Single Instruction, Multiple Data) Extentions) Registers olarak bilinen XMM0-XMM15 arasındaki Register'ların kullanılması gerekiyor. Bu Regiter'lar üzerinde sürecin genel olarak nasıl gerçekleştirildiğine bakıldığında;

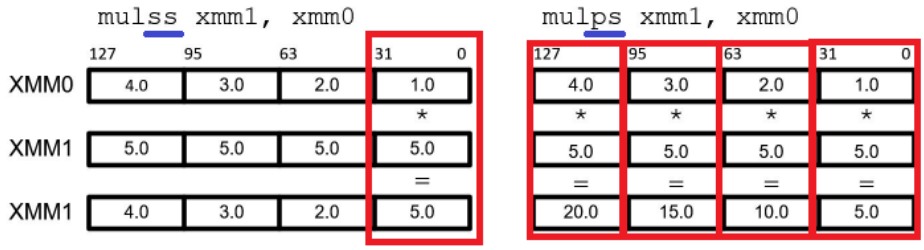
SSE (Streaming SIMD (Single Instruction, Multiple Data) Extentions) Registers, tek komutla birden fazla işlemi aynı anda yapabilmeyi sağlıyor. Bu Regsiter'lar 128 bitlik boyuta sahiptir ve sadece veri hesaplama işlemlerinde kullanılırlar. Bir Floating-Point (Single Precision) değer 32 bit alan kapladığı için tek bir SSE Register üzerinde aynı anda 4 farklı Floating-Point değer saklanabiliyor (IDA uygulamasında XMM Register üzerine fare getirildiğinde görülebiliyor).

- Bir Single Precision Floating-Point sayı -> 1bit işaret, 8 bit sayının tam kısmını, 23 bit küsürat kısmını ifade etmek için kullanılıyor.
- Bir Single Precision Floating-Point sayı -> 1bit işaret, 11 bit sayının tam kısmını, 52 bit küsürat kısmını ifade etmek için kullanılıyor.



SSE Register'lar üzerinde bulunan değerleri işlemek için Single Scalar ve Packed Scalar olmak üzere iki tip komut yapısı bulunuyor. Bu komutlar;

- Single Scalar, SSE Register üzerinde bulunan verilerden sadece en anlamsız 32 bitinde bulunanlar arasında işlem yapılmasını sağlayan koutlardır. Single Scalar komutlar sonunda "ss" takısı barındırır.
- Packed Scalar, SSE Register üzerinde bulunan verileri aynı anda ve paralel olarak işlenmesini sağlar. Packed Scalar komutlar sonunda "ps" takısı (Packed Scalar) barındırır.



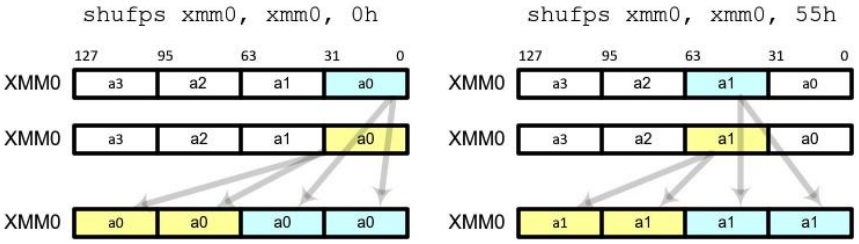
|→ Bu doğrultuda SSE Regsiter'lar üzerinde veri taşıma işlemlerinde Single Scalar işlem için "**movss**" komutu kullanılırken, Packed Scalar için Register üzerinde veri aktarılacak kısımları belirlemek üzere çeşitlenebiliyor. Birkaç örnek verilecek olursa;

- **Movlps**, en anlamsız 64 bit üzerindeki (2 Floating-Point değer) kısımda veri taşımak için kullanılır.
- **Movhps**, en anlamlı 64 bit üzerindeki (2 Floating-Point değer) kısımda veri taşımak için kullanılır.
- **Movaps**, hizalanmış şekilde 128 bitin (4 Floating-Point değer) tamamı üzerinde veri taşımak için kullanılır.
- **Movups**, hizalanmamış şekilde 128 bitin (4 Floating-Point değer) tamamı üzerinde veri taşımak için kullanılır.
- **Movhlps**, kaynak operandın en anlamlı 64 biti (2 Floating-Point değer) hedef operandın en anlamsız 64 bitine (2 Floating-Point değer) taşımak için kullanılır.
- **Movlhps**, kaynak operandın en anlamsız 64 biti (2 Floating-Point değer) hedef operandın en anlamlı 64 bitine (2 Floating-Point değer) taşımak için kullanılır.

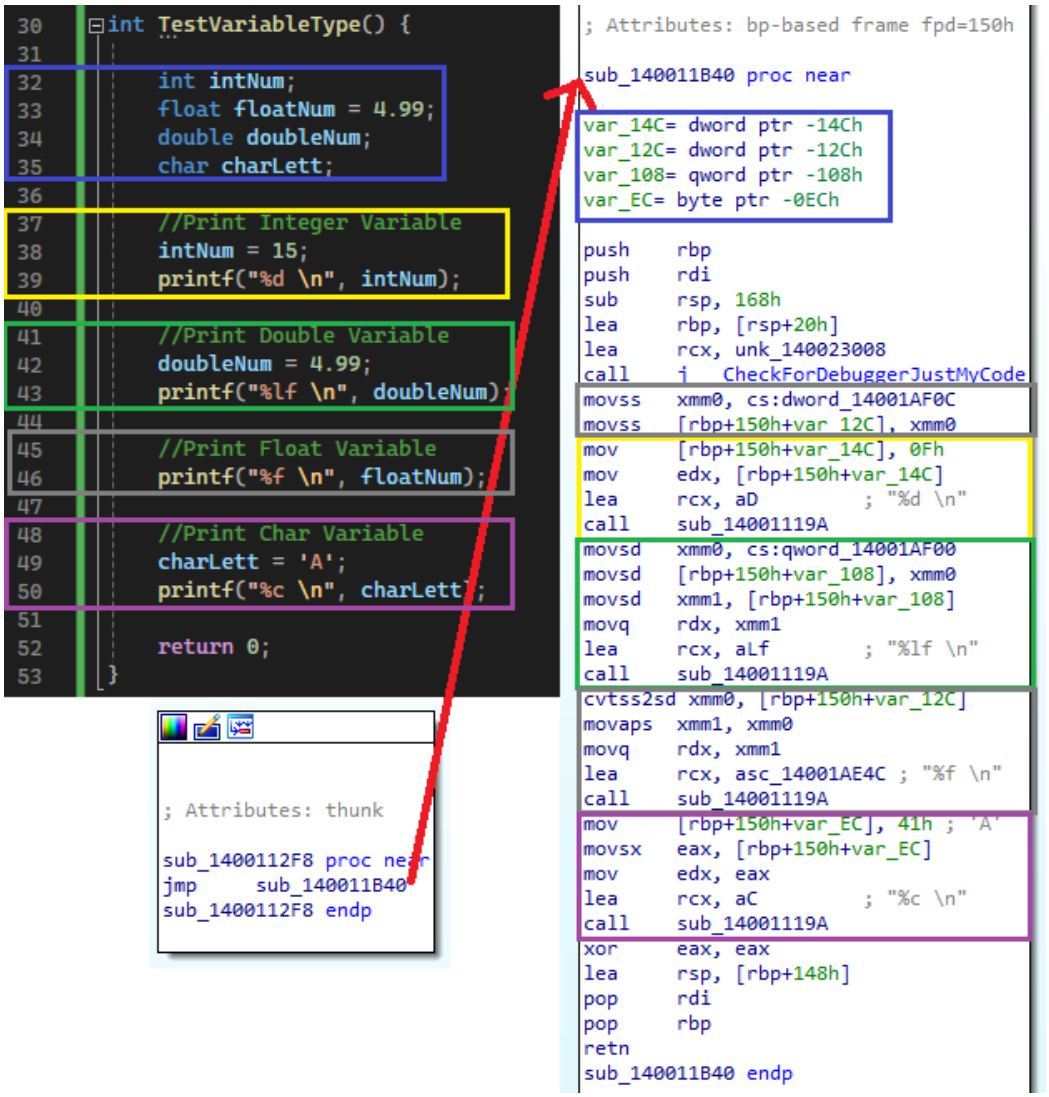
Aritmetik işlemlerde kullanılan komutlar da Single Scalar ve Packed Scalar olacak şekilde iki farklı yapıda kullanılabiliyor (addss, addps, subss, subps, mulss, mulps, dicss, divps …).

Shuffle Instruction olarak bilinen komutlar da bulunuyor. Bu komutlar XMM Regsiter üzerindeki depolanabilecek 4 Float değerden birisinin kendi veya farklı bir XMM Regsiter üzerinde dağılmasını sağlamak için kullanılıyor. XMM Regsiter üzerindeki hangi Float değerin dağıtılacağını belirlemek için;

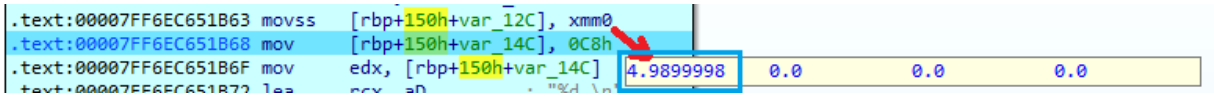
- 00h, ilk Floating değerin (0-31 bit aralığındaki) dağıtılacağını gösterir.
- 55h, ikinci Floating değerin (32-63 bit aralığındaki) dağıtılacağını gösterir.
- AAh, üçüncü Floating değerin (64-95 bit aralığındaki) dağıtılacağını gösterir.
- FFH, dördüncü Floating değerin (96-127 bit aralığındaki) dağıtılacağını gösterir.



SSE Register üzerine fikir sahibi olduktan sonra kaynak kodu incelemeye devam edelim.



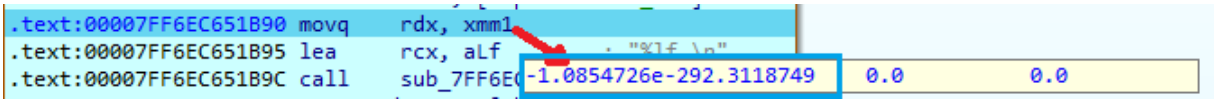
C kodunda Float tipindeki değişken tanımı yapılırken başlangıç değerinin de verildiği görülüyor. Disassemble edilen kısımda ise diğer işlemlere başlamadan önce movss komutuyla (tek bir Floating Point için) XMM0 Register'ına başlangıç değerinin taşındığı görülebiliyor. Daha sonra XMM0 Register üzerinden bu değer Stack alanındaki adresine kaydediliyor. Bunun nedeni ilerleyen adımlarda yine XMM0 Regsiter'ı kullanıldığında (örnek olarak ilerleyen süreçte Double tipindeki değişken ile işlem yapılıyor) Float tipindeki değişkenin başlangıç değerini kaybetmemektir.



Sarı çerçeve içerisine alınan kısımda 0fh (Decimal -> 15) değerinin Stack alanındaki adresine taşındığı anlaşılıyor. Farklı renklerle çerçevelenen kısımlardan da anlaşılacağı üzere C kodundaki "**Printf**" fonksiyonunun konsol ekranına basılacak değeri RDX, basılacak değişken tipini RCX Register'ından depoladığı anlaşılabiliyor.

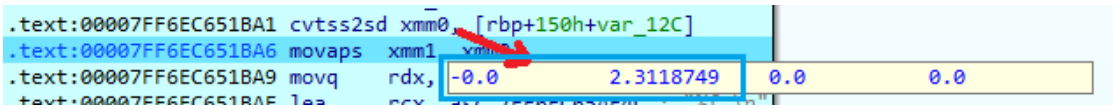
- RDX Register'ına değeri depoluyor. Ardından RCX Register'ının adresinin değişken tipinin bulunduğu değişken (aD) adresine ayarlıyor.

Yeşil çerçeveli kısımda yine ilk iki satırda Double tipinde tanımlı değişkenin atama işleminin yapıldığı görülüyor. Bu işlemde Double tipindeki değişkenlerin 64 bit yer kapladığı görülebilir. Sonraki satırlarda değer XMM0 Register'ından XMM1 Register'ına taşınıyor. Son olarak "**Printf**" fonksiyonuna dallanmadan önce RDX ve RCX Register'ları set edilerek dallanıyor.



Gri çerçeveli kısımlara bakıldığında Float tipindeki değişkenin başlangıç değerinin fonksiyon başında Stack alanına taşındığı görülmüştü (çalışma süresince bu değeri kaybetmemek için). Float tipindeki değişken üzerinde işlem yapılacağı için bu değer "**cvtss2sd**" komutuyla (Single Scalar Floating Point değerini Double Scalar Floating Point değerine dönüştürmek için kullanılıyor) Stack alanından XMM0 Register'ına aktarılıyor.

- Burada cvtss2sd komutunun kullanılma nedeni değeri değiştirmeden belekten 64 bit şekilde alabilmektir. 32 bit şeklinde alınsaydı değer XMM0 Regsiter'ı üzerinde 32-63 bir arasına konumlanmak yerine 0-31 bit arasında konumlanacaktı. Bu durumda değer XMM0 Regsiter'ı üzerinden RDX Register'ına aktarılırken olması gerekenden daha yüksek/farklı bir değerde aktarılmış olacaktı.



XMM0 Regsiter'ına depolanan değer "**movq**" (Quad Word – 64 bit değerleri taşımak için kullanılıyor) komutuyla XMM0 Regsiter'ından XMM1 Register'ına aktarıldıktan sonraki süreç ise yine Printf fonksiyonu için gerekli ayarlamaların yapılıp konsola değerin yazdırılmasından ibarettir.

Mor çerçeveli kısımda ise "**A**" harfinin ASCII karşılığı olan 41 değer Stack alanına taşındıktan sonra Printf fonksiyonu için gerekli ayarlamalar yapılarak komut ekranına yazdırılması sağlanıyor.

Kaynaklar

- https[:]//faculty.kfupm.edu.sa/COE/aimane/assembly/pagegen-95.aspx.htm#:~:text=The%20PROC%20and%20ENDP%20directives,and%20end%20of%20a%20procedure.&text=In%20a%20NEAR%20procedure%2C%20both,segments%20in%20a%20FAR%20procedure.
- https[:]//medium.com/@derya.cortuk/function-prologue-and-epilogue-b999f9637f77
- https[:]//www.youtube.com/watch?v=wbOuHloRkgk&list=PLwP4ObPL5GY_NfJs9BBjDlzXN40Ft4vMo&index=2
- https[:]//upload.wikimedia.org/wikipedia/commons/a/ac/ProgramCallStack2_en.svg
- https[:]//www.cs.fsu.edu/~engelen/courses/COP402001/notes5.html
- https[:]//medium.com/@memrekaraaslan/nedir-bu-memory-stack-heap-memory-leak-memory-management-c3c14d1c3e6e
- https[:]//www.scaler.com/topics/c/memory-layout-in-c/
- https[:]//www.cs.uaf.edu/courses/cs301/2014-fall/notes/float-asm/index.html
- https[:]//revers.engineering/applied-re-accelerated-assembly-p2/

- http[:]//www.nacad.ufrj.br/online/intel/vtune/users_guide/mergedProjects/analyzer_ec/mergedProjects/reference_olh/mergedProjects/instructions/instruct32_hh/vc61.htm
- https[:]//en.wikibooks.org/wiki/X86_Assembly/SSE
- https[:]//eng.libretexts.org/Bookshelves/Computer_Science/Programming_Languages/x86-64_Assembly_Language_Programming_with_Ubuntu_(Jorgensen)/18%3A_Floating-Point_Instructions
- https[:]//www.songho.ca/misc/sse/sse.html
- https[:]//students.mimuw.edu.pl/~zbyszek/asm/en/instrukcje-sse.html