Chapter 4 ch 4.2

Simple Instructions

* x 86 mimarisinde kullanılan en temel komutlardan bir tanesi mov komutudur. Kullanımı;

mov destination, source şeklindedir.

(2) komuttan sonra destination (hedef) adres daha sonra source (kaynak) adres gelir.

mov komutu veriyi belleğe veya register'lara taşır.

Dolaylı adreslemede köşeli parantezler içerisine yazılan Ram adresindeki değer ifade edilir.

* Bu tabloda mov komutunun farklı kullanım şekilleri gösterilmektedir.

mov eax, ebx → EBX register'ının içeriğini EAX'e kopyalar.

mov eax, 0x42 → 42 değerini EAX'e kopyalar.

mov eax, [0x4037C4] → 0x4037C4 adresinden 4 byte'lık bir veriyi EAX register'ına kopyalar.

(3) mov eax, [ebx] → EBX register'ının içerisinde depolanan Ram adresine gider ve o adresten 4 byte'lik veriyi EAX register'ına kopyalar.

mov eax, [ebx + esi*4] → İşlem sonucunda elde edilen Ram adresine gider ve o adresten 4 byte'lik veriyi EAX register'ına kopyalar.

Load Effective Address (lea)

* Bir başka genel ve çok kullanılan komut lea (Load Effective Address) komutudur.

lea 'nın kullanımı ve yazım formatı mov komutuna benzer.

lea destination, source

Kullanımları benzer olsa da yaptıkları işler farklıdır.

lea eax, [ebx + 8] → source 'taki adresi destination 'a kopyalar. Yani burada EBX 'in içerisinde bulunan adres değerine 8 ekler ve EAX 'a kopyalar.

mov eax, [ebx +8] → Ram 'deki ebx+8 adresindeki veriyi EAX 'e kopyalar.

Mantık olarak lea 'nın yaptığı işlem mov ile şu şekilde yapılabilir :

```
mov  eax, ebx+8
```

ancak bu şekilde bir kullanıma izin verilmemektedir.

* Şekil mov ve lea komutlarının kullanım farkını göstermektedir.

```
mov  eax, [ebx+8]
```

çalıştırıldığı zaman source 'ta bulunan adresin içeriği kastedilir.

lea eax, [ebx + 8]

çalıştırıldığı zaman source'taki adres değeri EAX'in içine kopyalanır. Yani burada EAX'in içeriği 0x00B30040 olur.

Arithmetic

* Aritmetik operatörler ise şu şekildedir:

sub → Subtracts (çıkarma)

add → Adds (toplama)

(6) inc → Increments (artırma)

dec → Decrements (eksiltme)

mul → Multiplies (çarpma)

div → Divides (bölme)

* Aritmetik operatörlerin kullanımı aşağıdaki gibidir:

sub eax, 0x10 → EAX'taki değerden 0x10 çıkarılır.

add eax, ebx → EAX ve EBX'i toplar ve sonucu EAX'e atar

inc edx → EDX'i 1 artırır.

(7) dec ecx → ECX'i 1 eksiltir.

mul 0x50 → EAX'i 0x50 ile çarpar ve sonucu EDX ve EAX'ta tutar.

div 0x75 → EDX:EAX'teki değeri 0x75'e böler ve bölümü EAX'te kalanı EDX'e atar.

* Genel aritmetik ve shift işlemleri :

`xor eax, eax` ⟶ EAX register'ını temizler. Bir register'ı kendisi ile XOR'lamak o register'ın değerini 0 yapar.

`or eax, 0x775` ⟶ EAX'in içeriği ile 0x7575 değerini "VEYA" işlemine alır.

```
mov eax, 0xA
shl eax, 2
```

⟶ İlk komut hexadecimal A yani decimal 10 değerini EAX'e atar. Binary olarak ifade edersek 1010'dır.

İkinci komut EAX'in içeriğini 2 defa sola shift eder yani binary olarak değeri 1 0 1 0 0 0 olur.

(1x32 0x16 1x8 0x4 0x2 0x1 = 40)

Yani decimal olarak 40, hexadecimal olarak 0x28 olur. Zaten binary bir sayıyı sola shift etmek 2 ile çarpmaktır. Örnekteki sayı decimal 10 değeriydi, 2 defa sola shift ettik yani 4 ile çarpmış olduk ve değeri 40 oldu.

```
mov  bl, 0xA
ror  bl, 2
```

→ Birinci komut yine hexadecimal A değerini BL'ye atıyor. ror işlemi "right rotate" işlemidir. Yani BL'nin içeriğini sağa 2 bit döndüreceğiz.

(8)

Right rotate işleminde bitler sayı kadar sağa kaydırılır ve sağdan kaybolan bitler soldan gelir. BL'nin şu anki içeriği 00001010 sayısıdır. 2 bit sağa rotate'ten sonra 10000010 olur.

No Operation (NOP)

* Sıkça kullanılan bir başka komut No Operation komutu NOP'tur. Bu komut işlemciye bir şey yapmamasını yani beklemesini söyler.

Bu komut aslında xhcg eax, eax komutunu çalıştırır ve böylece gerçek anlamda hiçbir şey yapmamış olur.

(9)

NOP'un Opcode'u 0x90'dır.

(10) (Kızak)

NOP Sled Saldırısı: Saldırgan programın bellek alanına kendi kodunu enjekte eder ve buffer overflow (tampon taşması) gerçekleştirir.

Daha sonra bir dizi NOP komutu ile programın kendi enjekte ettiği komutlara gelmesini sağlar. Yani NOP sled saldırganın bir programın belleğindeki istenen bir kod konumunu hedeflemesine yardımcı olmak için kullanılabilir.

The Stack

* Akış kontrolünün, metotların, parametrelerin ve lokal değişkenlerin fiziksel bellekte depolandığı yer, programın Ram'deki stack bölgesidir. Stack kısa zamanlı depolama için kullanılır.

Adından da anlaşılacağı üzere stack (yığın) veri yapısı mantığı ile çalışır. İşlemler Last In First Out (LIFO) kuralı ile yapılır.

PUSH işlemi stack'e eleman ekler.
POP işlemi stack'ten eleman çeker.

ESP (Extended Stack Pointer) stack bölgesinin üstünü gösterir yani buranın adresini tutar. Stack'e eleman eklendiğinde veya stack'ten eleman çıkarıldığında değeri değişir.

EBP (Extended Base Pointer) stack bölgesinin altını gösterir yani buranın adresini tutar.

Other Stack Instructions

* Call ve Enter komutları assembly dilinde alt program çağrısı ve yerel değişkenlerin yönetimi için önemli araçlardır.

Ret komutu fonksiyonun çalışması tamamlandığında yani alt programın çalışması bittiğinde program akışının kaldığı yere geri dönmesini sağlar.

Leave komutu fonksiyonun çalışması tamamlandığında yerel değişkenlerin temizlenmesini sağlar.

Function Calls

* Fonksiyon bir program içindeki belirli bir görevi gerçekleştiren ve geri kalan koddan nispeten bağımsız olan kod bölümüdür.

Ana kod fonksiyon çağırıldığında yürütmeyi fonksiyona bırakır taki fonksiyonun çalışması sonlana kadar.

Küçük fonksiyonlar tek bir iş yaparlar, Printf() gibi. Printf() fonksiyonu parametre olarak gelen değeri ekrana basar.

Birçok fonksiyon prologue içerir. Prologue, fonksiyon başında
12 bulunan birkaç satır komuttur. Prologue, fonksiyonun kullanımı için stack ve register'ları hazırlar.

Birçok fonksiyon epilogue da içerir. Epilogue, fonksiyon sonunda bulunan birkaç satır komuttur. Epilogue komutlar stack ve register'ların durumunu fonksiyon çağırılmadan önceki durumuna getirir.

Stack Frames

* Herhangi bir programın Ram'deki stack alanı şekildeki gibi gösterilebilir.

Bu şekle göre altta bulunan fonksiyon ortadakini çağırmıştır.
Artık onu çağırdıktan sonra program kontrolünü ona bırakır ve
13 onun çalışmasının bitmesini bekler. Ortada bulunan fonksiyon ise üstteki fonksiyonu çağırmıştır ve programın kontrolünü ona bırakır. Çağırdığı fonksiyonun çalışması bitene kadar beklemeye

Üstteki fonksiyonun çalışması bittiğinde bir değer döndürerek veya bir değer döndürmeden çağırıldığı yere geri döner. Bu şekilde programın stack alanı LIFO bir şekilde çalışmasını sürdürür.

En altta bulunan metod yeni bir fonksiyon çağırmadan çalışması sonlanırsa program da sonlanmış olur.

* Herhangi bir metodun stack çerçevesine de baktığımızda şekil-
14 deki gibi bir görüntü ile karşılaşırız. İçerisinde parametreleri, lokal değişkenleri ve geri dönüş adresi gibi veriler mevcuttur.

Conditionals

* Her programlama dili karşılaştırma yeteneğine sahiptir ve bu karşılaştırmalara dayanarak da kararlar alırlar.

Burada da conditionals 'dan kasıt karşılaştırma yapan Assembly komutlarıdır.

Assembly dilinde sıkça kullanılan iki adet conditional vardır: "test" ve "cmp" komutları.

15

test komutunun yaptığı iş AND komutuğu ile aynıdır ancak test komutu operand'ları değiştirmez, yanlızca bayrakları ayarlar.

Bir register'in içeriğinin NULL olup olmadığını anlamak için kendisine karşı test'i yapılır.

Örneğin test eax, eax komutu ile EAX'in içeriği sıfır ise ZF (Zero Flag) set edilir.

cmp conditional'ı da sub komutu ile aynı işi yapar ancak bu işlemden operand'lar etkilenmez.

cmp dst, src gibi bir komutta;

			ZF	CF	
dst	=	src	ise 1	0	
dst	<	src	ise 0	1	olur.
dst	>	src	ise. 0	0	

Branching

* Dallanma, program akışının belirli bir koşula göre farklı bir yere yönlendirilmesidir. Programın koşullara bağlı olarak farklı yollar izlemesini sağlar.

Assembly dilinde iki çeşit dallanma vardır:

Unconditional Jump (Koşulsuz sıçrama): Programın belirli bir adrese doğrudan dallanmasını sağlar. Komutu JMP'dir.

Conditional Jump (Koşullu Sıçrama): Belirli bir koşula bağlı olarak program akışını farklı bir yere yönlendirir. 30'dan fazla koşullu sıçrama komutu vardır.

jz loc → Eğer Zero Flag set edilmiş ise loc adresine sıçrama yapar.

jnz loc → Eğer Zero Flag sıfır ise loc adresine sıçrama yapar.

C Main Method

* Birçok malware C dili ile yazılmıştır. Bu sebep ile C'deki main fonksiyonunun assembly karşılığını bilmek malware'ları analiz ederken kolaylık sağlar.

Her C programı hatta her program main() fonksiyonu ile başlar. Derleyici kodu derlemek için main() fonksiyonunu arar ve bu satırdan kodu derlemeye başlar.

C programı nasıl başlar?

int main(int argc, char** argv) satırı ile başlar. Burada argc komut satırından girilen parametrelerin sayısını tutar. argv ise komut satırından girilen parametrelerin isimlerinin tutulduğu dizinin referansını tutar.

* Program komut satırından şu şekilde çağırılsın:

filetestprogram.exe -r filename.txt

argc = 3

argv[0] = filetestprogram.exe

argv[1] = -r

argv[2] = filename.txt

değerlerini alır.

* Burada basit bir C kodu var. Bir önceki sayfada komut satırından çalıştırılan filetestprogram.exe isimli programın kaynak kodları.

Bu kaynak kodlar derlenmiş ve filetestprogram.exe olmuş. Yani çalıştırılabilir bir exe dosyası olmuş. Daha sonra komut satırından program bazı parametreler ile çağırılmış.

Kod Açıklaması

* ## Assembly Kod Açıklaması