Chapter 4 ch 4.1

A Crash Course In X86 Disassembly

* Temel Static Analiz
Temel Dinamik Analiz yapıldı.

Bu iki teknik de bazı açılardan doğal sınırlara sahip. Dolayısı ile bu iki tekniğin yetersiz kaldığı yerlerde süpheli yazılımın kodlarına bakılabilir.

İleri statik analizde, hangi dil ile yazılmış olursa olsun, hangi derleyici ile derlenmiş olursa olsun executable dosyaları
(1) kaynak kodlarına veya assembly kodlarına dönüştüreceğiz.

Bu bölüm assembly kodlarını ve bilgisayar mimarisini hatırlatmak amacı ile yazılmış bir bölüm. Kitap X86 mimarisi için yazılmış zararlı yazılımları konu alıyor. Bu sebep ile bu bölümde X86 mikro işlemci mimarisi ele alınmış.

* Neden İleri Statik Analiz Yapalım?

Temel statik analizde şüpheli yazılımı dışarıdan hızlıca kontrol ediyorduk,

(2) Ancak bazı sebeplerden ötürü temel statik analiz yeterli gelmediği için temel dinamik analize yöneldik,

Bu sebepler nelerdi?

→ Yazılım gizlenmiş ise temel statik analiz için kullanılan yöntemler etkisiz kalabilir.

→ Zararlı yazılımın fonksiyonelliği yani zararlı yönü runtime 'da ortaya çıkabilir.

- Runtime Linking
- Network (Zararlı yazılımı internetten indirme)

→ Sadece temel statik analiz yaparak bir programın zararlı olup olmadığına karar veremediysek.

Temel statik analizden sonuç alamadığımızda, temel dinamik analize yöneldik. Temel dinamik analizin de yeterli kalmadığı durumlar var;

(2)

→ Zararlı yazılım çok iyi yazılmış ise sanal makine içerisinde çalıştırıldığını anlar, dolayısı ile zararlı yönünü göstermeyebilir.

→ Yazılım zararlı yönünü çok uzun bir süre sonunda gösterebilir. Analiz için o kadar uzun süre beklenilmeyecektir.

→ Temel dinamik analiz yaparken aslında yazılımı tek bir case (durum) için çalıştırırız. Belki de yazılımın zararlı yönü bir olay gerçekleştiğinde tetiklenecektir. Örneğin network'ten bir mesaj gelmesini bekliyor olabilir.

→ Programı incelemel şansımız olmuyordur. Lab 3-1'te olduğu gibi program çalışıp kapanıyor ve kendini siliyor olabilir.

Dolayısı ile bu sebeplerden ötürü temel dinamik analiz de yetersiz kaldığında şüpheli yazılımın kodlarına bakmak isteriz. İleri statik analizde executable dosyaların kaynak kodlarını veya assembly kodlarını inceliyor olacağız.

Assembly makine dilinin üstünde ve insanın okuyabileceği, anlayabileceği en alt seviye dildir. Yani human readable bir dildir.

Levels of Abstraction

* Geleneksel bilgisayar mimarisi implementasyonların detaylarını gizlemek için soyutlamayı bir çok yerde kullanır. (abstraction)

(3) Örneğin Windows işletim sistemini farklı donanımlar üzerinde çalıştırabiliriz çünkü söz konusu donanımlar işletim sisteminden bağımsızdırlar yani soyutlanmıştırlar.

* Şekilde görüldüğü gibi zararlı yazılım yazarı yüksek seviye bir yazılım dili kullanarak bir zararlı yazılım yazıp derliyor. Bu yazılım artık CPU üzerinde çalıştırılabilecek makine kodlarına dönüştürülüyor.

(4) Zararlı yazılım analisti de tersine mühendislik ile makine kodlarını Disassembler kullanarak düşük seviye bir dile yani assembly diline çeviriyor.

veya

Decompiler kullanarak yüksek seviye bir dile çeviriyor. Bu kodları inceleyerek yazılımın zararlı olup olmadığını anlamaya çalışacağız.

Decompile (kaynak koda dönüştürme): Yürütülebilir bir programı yani bir executable'ı kaynak kodlarına (source code) dönüştürme işlemidir. Bu işlemi yapan programlara da Decompiler denir.

Disassemble (assembly koda dönüştürme): Yürütülebilir bir programı assemble koduna dönüştürme yani makine kodunu sembolik koda dönüştürme işlemidir. Disassembler ise bu işlemi yapan programlara denir.

IDA Pro, Ghidra, Scylla vb disassembler piyasada mevcut. Bu programların bazıları ücretli, ücretli olanların sınırlı versiyonları var.

Örneğin IDA Pro ücretli bir Disassembler ve Decompiler. Ücretsiz versiyonunda sadece Disassembler özelliği çalışıyor.

Ghidra tamamen ücretsiz bir yazılım. NSA tarafında 2000'li yılların başında zararlı yazılımları analiz etmek için geliştirilmiş kuruma özel bir yazılımken daha sonradan 2017 yılında Wiki-Leaks'in CIA belgelerini ifşa etmesi ile birlikte ortaya çıkıyor. Bu sebepten ötürü NSA bu programı herkesin kullanımına açıyor. Programı NSA ücretsiz sunuyor dolayısı ile programda bir açık kapı olma ihtimali var. Mümkünse bir sanal makine ile kullanmak daha doğru olur.

* Geleneksel bilgisayar mimarisinde altı seviye soyutlama var.

- Hardware
- Microcode
- Machine code
- Low-level languages
- High-level languages
- Interpreted languages

Anlamak zor — Ayrıntı çok

Bilgisayar Mimarisi Modeli

Anlamak kolay — Ayrıntı az

* Hardware (Donanım):

→ Fiziksel düzeyi tarif eder.

→ Dijital elektronik ile alakalıdır. XOR, AND, OR, NOT kapıları ile işlemler yapılır.

→ Fiziksel doğası sebebiyle yazılımlar tarafından kolayca manipule edilemez, değiştirilemez.

* Microcode (Mikrokod):

→ Mikrokoddan kasıt aslında firmware'lardır. Firmware'lar fiziksel donanım üzerine gömülü yazılımlardır.

→ Hangi spesifik donanım için yazılmışlar ise o donanımı yönetirler. Yani alt seviye bir driver'dır denilebilir.

→ Daha yüksek düzey olan makine kodunu donanımların anlayabilmesi için microinstruction'lar içeren bir arayüzdür.

→ Zararlı yazılımlar genellikle bu seviyede yazılmazlar, dolayısıyla bizim için şu an önemli olmayan bir katman, seviye.

* Machine Code (Makine Kodu – Opcode):

→ Makine kodu Opcode'lardan oluşur. Opcode'lar da hexadecimal digit'lerden oluşur ve mikroişlenciye yapılması gerekeni bildirir.

→ Bir programı yüksek-seviye bir dil ile yazıp derledikten sonra oluşur.

(8)

→ Makine kodu genel olarak bir çok mikrokod instruction seti kullanılarak implemente edilebilir. İlgili donanım da bu instruction setini kullanarak kodu çalıştırır.

* Low-Level Languages (Düşük Seviye Diller):

→ Düşük seviye diller aslında bilgisayar mimarisinde instruction set'in insan tarafından okunabilen yani human-readable bir versiyonudur.

(9)

→ En genel bilinen düşük seviye dil assembly dilidir.

→ İnsan tarafından okunabildiği için zararlı yazılım analizcileri genellikle düşük seviye diller üzerinde çalışırlar. Çünkü makine kodunu okumak ve anlamak zordur.

Not: Kaynak kodun erişilebilir olmadığı durumlarda Assembly dili insanın anlayabileceği en yüksek seviye dildir.

* High-Level Languages (Yüksek Seviye Diller):

→ Bir çok bilgisayar programı yüksek-seviye bir dil ile yazılıp çalıştırılırlar. Çünkü yüksek seviye diller makine düzeyindeki diller için bir soyutlama sağlar ve programlamanın mantık ve akış kontrol mekanizmasının kolay bir şekilde kullanılmasını sağlar.

→ C ve C++ yüksek seviye dillere örnektir. Bu diller ile yazılan kodlar compiler (derleyiciler) tarafından makine koduna çevrilir. Yapılan işleme de derleme işlemi denir.

* Interpreted Languages (Yorumlayıcı Diller):

→ Bu programlama dilleri en üst seviye dillerdir.

→ Java, C#, Perl, .NET, Python vb. diller bu gruba girer.

→ Bu seviyedeki diller ile yazılmış bir program direk makine koduna çevrilmez bytecode'a çevrilirler ve diske o şekilde kaydedilirler.

Bytecode: Makine dili ile programlama dili arasında ara bir gösterimdir ve dile özgüdür. Program çalıştırıldığında bytecode gerçek zamanlı olarak yorumlayıcı tarafından yorumlanarak donanımın anlayacağı makine koduna çevrilir.

→ Yorumlayıcı diğer geleneksel olarak derlenen koda kıyasla

bağımsız olarak hataları yakalama, hafıza yönetimi olanakları sağlar.

Java yorumlayıcısı ⟶ JVM (Java Virtual Machine)
.Net yorumlayıcısı ⟶ Framework X.X

Reverse Engineering (Tersine Mühendislik)

* Tersine mühendislikten kasıt bir program herhangi bir programlama dili kullanılarak makine diline dönüştürülmüş. Makine
(12) dilini anlamamız zor dolayısı ile makine dilini human-readable bir dile yani assembly diline çevireceğiz.

* Diğer programlar gibi zararlı yazılımlar da disk üzerinde ya bytecode ya da makine kodu düzeyinde binary formatta bulunur.

(13) Disk'te duran bu binary formattaki kodu disassembler yardımı ile assembly diline çevireceğiz.

Bu işlemi yapan popüler programlardan bir tanesi de IDA Pro.

* Assembly işlemci özelinde bir dildir. Yani farklı tip işlemciler için farklıdır. Çünkü instruction set farklıdır. Üzerinde çalışılan işlemci mimarisi bu yüzden önemlidir.

(14) PC'ler için kullanılan işlemci mimarileri X86, X64, SPARC, PowerPC, MIPS, ARM vb'leridir.

(9)

Dünya üzerinde en çok kullanılan işlemci Intel firmasının 32 bitlik X86 'sıdır. Son zamanlarda Intel'in 64 bitlik X64 işlemcisi de yaygınlaşmaya devam ediyor.

(14) Windows işletim sistemi X64 ve X86 işlemcileri üzerinde çalıştırılabilmektedir. Ancak X86 mimarisindeki bir işlemci 64 bitlik işletim sistemini ve 64 bitlik programları çalıştıramaz.

Günümüzde bir çok malware X86 mimarisi üzerinde çalışmaktadır.

* Günümüzde bir çok bilgisayar Von Nouman mimarisini benimser.

- Control Processing Unit (CPU) kodu çalıştırır.
- Main Memory (RAM) bütün veriyi ve kodu depolar.

(16)

- Input / Output Devices (I/O) veri depolayan (hard disk, klavye, monitor vb.) ve istenildiğinde veriyi geri getiren aygıtlardır.

*

CPU Components
(CPU Bileşenleri)

* Control Unit: Instruction pointer isimli register'ı kullanarak RAM içerisinde işletilmesi gereken komutları alır, yönetir.

(17) Registers: Register'lar CPU'nun temel depolama birimleridir. CPU işlemi esnasında kullandığı ve elde ettiği verileri register'larda depolar. RAM'den hızlıdır.

ALU (Arithmetic Logic Unit): Kontrol biriminin RAM'den getirdiği komutları çalıştırır. Elde edilen sonuçları register

(17) lara veya RAM'e kayd eder.

Programın çalışması bitene kadar fetching - executing işlemleri devam eder.

Main Memory
(Ana Bellek)

* Bir programda bellek şekildeki gibi 4 ana bölüme ayrılır. Stack, Heap, Code ve Data olmak üzere.

(18)

Bu şekil temsili bir gösterim başka bir programda stack alanı kod alanından büyük olabilir.

* Data: Programın başlangıcında yüklenen verileri tutmak için RAM'de ayrılan bölümdür. Bu veriler zaman zaman static değerler olarak adlandırılabilirler. Çünkü programın

(19) çalışması sırasında değişmezler. Veya bazen de buradaki veriler global değerler olarak bilinirler. Çünkü programın her yerinden erişilebilirler.

* Code: RAM'de ki code bölümü çalıştırmak için CPU tarafından fetch edilen yani alınan kodları barındırır. Buradaki

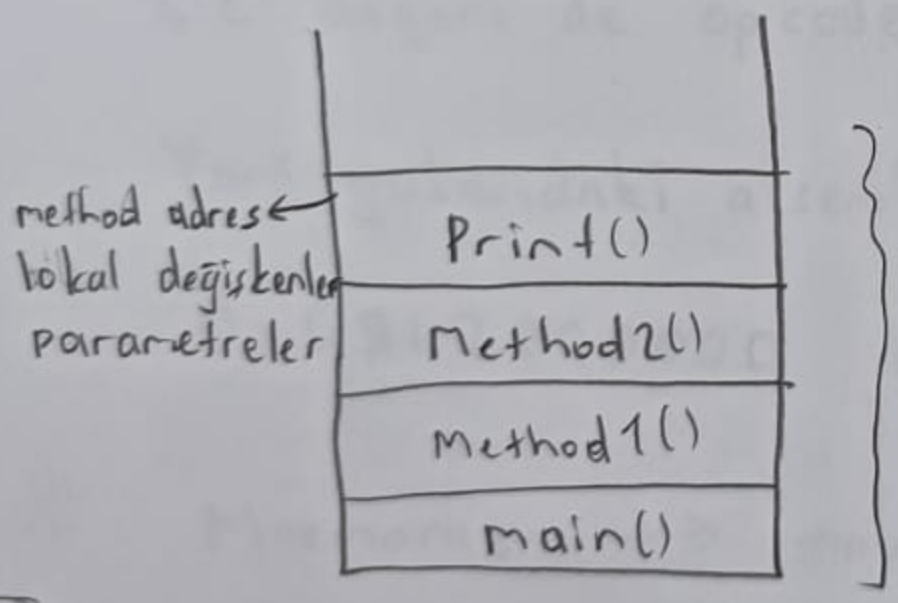
(20) kodlar programın ne yapacağını, nasıl yapacağını belirtir, kontrol eder.

* **Heap:** Programın icrası sırasında dinamik bellek olarak kullanılır. RAM'e bir veri yazmak için bu alandan yer tahsisi yapılır veya herhangi bir veriye daha fazla ihtiyaç yoksa buradaki alan serbest bırakılır yani veri silinir.

(21)

Heap programın dinamik bellek alanı olarak bilinir. Çünkü programın çalışması sırasında içeriği sürekli değişir.

* **Stack:** Çağrılan fonksiyonların adresleri, lokal değişkenler ve fonksiyon parametreleri bu bellek alanında tutulur. Program akışını kontrol etmeye yardımcı olur.



(22)

Stack veri yapısı mantığında çalışır. Yani son giren ilk çıkar (LIFO).

Yeni bir metot çağırıldığında bu stack'in üstüne eklenir. Bu işleme pushing denir.

Bir metodun icrası bittiğinde bu stack'in üstündeki metot silinir. Bu işleme pop etmek denir.

Instructions
(Talimatlar - Komutlar)

* Instruction 'lar assembly programlama dilinin yapıtaşlarıdır. X86 mimarisinde bir instruction bir mnemonic ve sıfır veya daha fazla operand 'dan oluşur.

```
mov  ecx  0x42
 ↓    ↓    ↓
mnemonic operand operand
```

(23) Bu komut 42 değerini Extended C -register'ına atar.

"mov ecx" 'in hexadecimal karşılığı opcode olarak 0xB9 'dur.

42 değeri de opcode olarak 42000000 olarak ifade edilir.

Yani yukarıdaki assembly komutunun hexadecimal gösterimi:

0xB942000000 olur.

* Mnemonic → mov

Destination Operand → ecx

(24) source Operand → 0x42

Intel işlemcilerde <u>Source Operand</u> sağ tarafta <u>Destination Operand</u> sol tarafta bulunur. Farklı işlemcilerde bu durum ters olabilir.

Endianness
(Endianlık)

* Farklı işlemcilerin çalışma şekilleri de farklı olabilmektedir. Bazıları Big-Endian, bazıları Little-Endian çalışırlar.

Big-Endian : Most significant (En önemli) byte'lar ilk önce yazılır.

Örneğin 0x42 64 bitlik değeri 0x0000042 olur.

Little-Endian: Least significant (En önemsiz) byte'lar ilk önce yazılır.

25

Örneğin 0x42 64 bitlik değeri 0x42000000 olur.

Network big-endian çalışır, yani network verileri big-endian ifade edilir. X86 işlemcisi little-endian çalışır yani bu işlemci üzerindeki veriler little-endian ifade edilir.

Zararlı yazılım analiz ederken Endian'lığın farkında olmak gerekir. Aksi takdirde yanlış çıkarımlar yapılabilir.

* Örneğin IP adreslerini ele alırsak;

26

127.0.0.1

↓

0 1111111 1

↓ ↓

128 1

big-endian

7F.00.00.01 (hex)

↓ ↓

16 1

16x7 + 15.1 = 127

Bu IP verisi network üzerinden RAM'e geldiği zaman RAM'de 0x0100007F olarak ifade edilir.

Opcodes (Operation Code)
(İşlem Kodu)

* Instruction mov ecx 0x42 olsun.

mov ecx'in Opcode'u B9

(27) X86 mikroişlemci ailesi little-endian çalıştığı için 0x42'nin değeri de 42 00 00 00 olur.

Zararlı yazılım Network ile ilgili işlemler yaparken bu endian'lik durumu değişebilir. Bu duruma dikkat etmek gerekir.

Operands
(İşlenen)

* Operand'lar instruction tarafından kullanılacak veriyi tanımlar. 3 farklı operand türü vardır:

Immediate: Operand'lar sabit değerlerdir.

(28) Register: Operand'lar register'ları temsil eder. Bir önceki örnekteki ecx register'ı gibi.

Memory Address: Operand'lar direk bellekteki adresi temsil ederler ve köşeli parantezler içine yazılırlar. [eax] gibi.

Registers
(Kayıtçılar)

* Register CPU'nun küçük boyutlu verileri depolayabildiği ve bu depoladığı verilere herhangi bir depolama aygıtından çok daha hızlı ulaşabildiği bir depolama birimidir.

29 X86 işlemcileri geçici depolama ve çalışma alanı için bir dizi register kullanır.

X86 işlemci ailesindeki register'ları 4 ana gruba ayıra-biliriz: General registers, Segment registers, Status register ve Instruction pointer.

* General registers: CPU'nun çalışması boyunca kullanılan genel amaçlı register'lardır.

Segment registers: Program ile ilgili verilerin bellekte bölüm-

30 lerini belirtmek için kullanılır.

Status flags: Durum register'larıdır. Durum belirttikleri için ve karar verme işlemlerinde kullanıldıkları için flag ismini alırlar.

Instruction pointer: Bir sonraki işletilecek instruction'ın adresini tutar.

Size of Registers
(Register'ların Boyutu)

* Genel amaçlı bütün register'lar 32 bittir. Ancak 16 bitlik veya 32 bitlik bir assembly kodu ile belirtilebilirler. Yani 16 bitlik veya 32 bitlik kullanılabilirler.

edx (Extended dx) → 32 bit
dx → 16 bit

(31) olarak kullanılır.

4 adet register eax, ebx, ecx, edx 32 bit, 16 bit veya 8 bit olarak kullanılabilir.

EAX → 32 bit
AX → 16 bit
AL → 8 bit
AH → 8 bit

* Şekilde EAX register'ının değişik uzunluklardaki kullanımı

(32) görülüyor.

General Registers
(Genel Kayıtçılar)

* Genel register'lar tipik olarak veri veya bellek adresi tutarlar.

(33)

Bazı instruction'lar da yaptıkları iş için genellikle aynı spesifik register'ı kullanırlar. Örneğin çarpma ve bölme komutları EAX ve EDX register'larını kullanırlar. Bu bir gele-
(33) nek haline gelmiştir.

Derleyiciler de aynı geleneği sürdürürler. Örneğin bir fonksiyon çağırımından sonra EAX register'ını görürseniz, bu register fonksiyonun dönüş değerini tutar.

Bu geleneksel kullanımları bilmek Zararlı Yazılım Analizcisi için kodu anlamada kolaylık sağlar.

Flags (Bayraklar)

* EFLAGS register'ı statu, yani durum register'ıdır. Karar verme mekanizması için kullanılır.

(34) 32 bittir ve her biti bir flag'dır. Bitleri 1'e set edilir veya temizlenir 0 olur.

* ZF Zero Flag: Bir işlemin sonucu sıfır olursa set edilir.

CF Carry Flag: Bir işlemin sonucu hedef için çok büyük veya çok küçük olursa set edilir.

(35) SF Sign Flag: Bir işlemin sonucu negatif olursa veya most significant bit 1'e eşit olursa set edilir.

TF Trap Flag: Debug yaparken kullanılır. Eğer bu register set edilmiş ise işlemci her seferinde bir adet instruc-

EIP (Extended Instruction Pointer)

* Bir sonraki çalıştırılacak instruction'ın bellekteki adresini içerir. Yani işlemciye, işi bittiğinde ne yapacağını söyler.

Instruction pointer veya program counter olarak bilinir.

⚠ EIP register'ının içeriği bozulursa program hata verir ve çöker.
EIP register'ının içeriği değiştirilirse program başka instruction lar çalıştırmaya başlar.

(36) Saldırganlar önce bellek adresine çalışmasını istedikleri kodu yerleştirirler yani enjekte ederler. Sonra EIP register'larını manüpüle ederek belleğe yerleştirdikleri kodun çalışmasını sağlarlar.

Buffer Overflow saldırısının hedefi de EIP register'ıdır. Program buffer'ına extra veri yazılır sonra EIP register'ı değiştirilip buffer'a yazılan kodların çalışması sağlanır.