2.Beyond Basic Static Analysis : X86 Disassembly

Kötü amaçlı yazılım örneklerinin derin statik analizinin merkezinde diassembly ve tersine mühendislik yatar.Bu bölümde x86 diassembly için anlamamız gereken kısımlardan bahsediyor.Ardından dissambly yi önlemek için yapılan taktiklerden nasıl geçebileceğimizi anlatacak.

disassembly demontaj: kötü amaçlı yazılımın ikili kodunu geçeri x86 derleme diline çevirdiğimiz süreçtir. malware tazarları genellikle kötü amaçlı yazılımları c veya c++ gibi yüksek seviyeli bir dilde yazar ve ardından kaynak kodunu x86 ikili koduna derlemek için derleyici kullanır. Assembly dili bu ikili kodun insan tarafından okunabilir temsilidir.

disassembly yöntemleri : linear disassembly doğrusal sökme

diğer yöntem ida pro gibi endüstriyel sınıf sökücüleri tarafından kullanılan daha karmaşık yöntemdir. bir programın bir dizi koşullu dallanma sonucunda hangi derleme yönergelerine ulaşabileceğini keşfetmek için programın yürütülmesini simüle eder veya bu konuda mantık yürütür.Bu tür bir demontaj doğrusal demontajdan daha doğru olabilse de, doğrusal sökme yöntemlerinden çok daha fazla CPU kullanır ve bu da onu binlerce hatta milyonlarca programı sökmeye odaklanan veri bilimi amaçları için daha az uygun hale getirir.

\*Basics of x86 Assembly Language:

assembly belirli bir mimari için  insan tarafından okunabilen en düşük seviyeli programlama dilidir . Bir assembly dili satırı neredeyse her zaman tek bir cpu talimatına eşdeğerdir. x86 derlemseinin assembly iki yapısı var : intel ve AT&T . Bu kitata intel sözdizimini kullanıyorç

\*CPU registers:

Registers Kayıtlar , üzerinde x86 cpuların hesaplama yaptığı küçük veri depolama birimleridir. registerlar cpunun kendisinde bulunduğundan kayıt erişimi bellek erişminden memory access çok daha hızlıdır. ayrıca cpunun çalışan programların durumu hakkında bilgi depolamak için registerları kullanmasının nedeni de budur. burada birkaç önemli registera bakalım:

yaygın hesaplama iş akışlarında (common computational workflows) programlar verileri bellekten veya harici donanım aygıtlarından yazmaçlara registerlara taşır, bu veriler üzerinde bazı işlemler gerçekleştirir ve ardından verileri depolamak için belleğe taşır. Genel amaçlı registerların bazı mimarilerii anlamak için :

eax = ax(16 bit)+ AH(8bit) +AL(8 bit)

EBX = BX(16) + BH + BL

ECX= CX (16)+CH+CL

EDX=DX(16) + DH(8) + DL

bunlar general purpose registers

ESI

EDI

ESP

EBP

\*Stack and Control Flow Registers:

stack  yönetimi kayıtları registerları fonksyolar için yerel değişkenleri ,fonksyionlara aktarılan bağımsız değişkenleri ve program kontrol akışıyla ilgili bilgilerini depolamaktan sorumlu olan program stackği hakkında bilgileri depolar.

Bu kayıtlara bakalım: ESP şu anda yürütlmekte olan işlev için stackin tepesini işaret eder. EBP kaydı şu anda yürütülen işlev için yığının altını işaret eder. EIP bufferoverflow gibi zafiyetler için önemlidir. bunun nedeni eipnin o anda yürütülmekte olan talimatın bellek adresini içermesidir. saldırganlardan eipyi ele geçirip programın akışını kontrol etmek isterler. aip malwarelerde kodların analizinde de önemlidir. herhangi bir zamanda malwarein hangi kodu çalıştırdığını anlamamıza yardımcı olur.

FLAGS, halihazırda yürütülmekte olan programın durumuyla ilgili durum bilgilerini depolayan bitler olan CPU bayraklarını içeren bir durum yazmacıdır. EFLAGS kaydı, x86 programları içinde koşullu dallar oluşturma sürecinin veya eğer / sonra tarzı program mantığının sonucundan kaynaklanan yürütme akışındaki değişiklikler için merkezidir.

\*Arithmetic Instructions:

Instructions          Description

add ebx, 100    EBX'teki değere 100 ekler ve ardından sonucu EBX'e kaydeder

sub ebx, 100     EBX'teki değerden 100'ü çıkarır ve sonucu EBX'e kaydeder

inc ah    AH'deki değeri 1 artırır

dec al  AL'deki değeri 1 azaltır

Add komutu, iki tamsayı ekler ve sonucu, bu ister bir bellek konumu ister aşağıdaki sözdizimine göre bir kayıt olsun, belirtilen ilk işlenende depolar. Yalnızca bir argümanın bir bellek konumu olabileceğini unutmayın. Alt komut, tamsayıları çıkarması dışında add komutuna benzer. İnc komutu, bir yazmaç veya bellek konumunun tam sayı değerini artırırken, dec, bir yazmaç veya bellek konumunun tamsayı değerini azaltır.

\*Data Movement Instructions:Veri Taşıma Talimatları

X86 işlemci, verileri yazmaçlar ve bellek arasında taşımak için sağlam bir talimatlar kümesi sağlar. Bu talimatlar, verileri işlememize izin veren temel mekanizmaları sağlar.  mov komutu ile yaparsın.

Instructions   Description

mov ebx,eax   Moves the value in register EAX into register EBX

mov eax, [0x12345678]  Moves the data at memory address 0x12345678 into the EAX register

mov edx, 1        Moves the value 1 into the register EDX

mov [0x12345678], eax        EAX'teki değeri 0x12345678 hafıza konumuna taşır

Mov komutu ile ilgili olarak, lea komutu, bir hafıza konumuna bir gösterici almak için kullanılan kayıt listesine belirtilen mutlak hafıza adresini yükler. Örneğin, lea edx, [esp-4] ESP'deki değerden 4'ü çıkarır ve elde edilen değeri EDX'e yükler.

\*Yığın stack talimatları:

x86 assembly stack yığın , değerleri üzerine itip çıkarmanıza olanak tanıyan bir veri yapısıdır.  Kontrol akışı genellikle x86 derlemesinde C stili fonksiyon çağrıları ile ifade edildiğinden ve bu fonksiyon çağrıları argümanları iletmek, yerel değişkenleri tahsis etmek ve bir fonksiyonun yürütülmesi bittikten sonra programın hangi kısmına dönüleceğini hatırlamak için yığını kullandığından, yığın ve kontrol akışın birlikte anlaşılması gerekir.

push değeri stacke yazar. pop stackten alır.

push 1 -> program yığın işaretçisini (ESP yazmacı) yeni bir bellek adresine işaret eder, böylece artık yığının en üst konumunda depolanan değer (1) için yer açar. Daha sonra argümandaki değeri CPU'nun yığının tepesinde yer açtığı bellek konumuna kopyalar.

pop eax->en üst değeri yığından çıkarmak ve belirli bir kayda registera taşımak için pop kullanır.op eax yığından en yüksek değeri çıkarır ve onu eax'a taşır.. X86 program yığını hakkında anlaşılması gereken sezgisel olmayan ancak önemli bir ayrıntı, bellekte memoryde aşağı doğru büyümesidir, böylece yığındaki en yüksek değer aslında yığın belleğindeki en düşük adreste depolanır. . X86 yığını bellekte aşağı doğru büyüdüğünden, itme talimatı program yığınında yeni bir değer için yer ayırdığında, ESP'nin değerini bellekte daha düşük bir konuma işaret edecek şekilde azaltır ve ardından hedef yazmacıdan bir değeri kopyalar. bu bellek konumu, yığının en üst adresinden başlayıp büyüyor. Tersine, pop komutu aslında yığından en yüksek değeri kopyalar ve ardından ESP'nin değerini artırarak daha yüksek bir bellek konumuna işaret eder.

\*control Flow Instructions: kontrol akışı talimatları

Bir x86 programının kontrol akışı, verilere, cihaz etkileşimlerine ve programın alabileceği diğer girdilere bağlı olarak bir programın yürütebileceği olası komut yürütme dizilerinin ağını tanımlar.

Ayrıca bu işlev çağrıları, argümanları iletmek, yerel değişkenleri ayırmak ve bir işlevin yürütülmesi bittikten sonra programın hangi kısmına dönüleceğini hatırlamak için yığını kullandığı için de ilişkilidir.

call talimatı instructions bir işlevi çağırır.Bunu, çağrı talimatı çağrıldıktan ve işlevin yürütülmesi tamamlandıktan sonra programın talimata dönmesine izin vermek için C gibi daha yüksek seviyeli bir dilde yazabileceğiniz bir işlev olarak düşünün. Aşağıdaki sözdizimini kullanarak çağrı talimatını çağırabilirsiniz; burada adres, işlevin kodunun başladığı bellek konumunu belirtir:

call address

çağrı talimatı iki şey yapar . İlk olaak , işlevin fonksiyonun çağrısı yığının tepesine döndükten sonra yürütülecek komutun adresini iter , böylece program çağrılan fonksyonun yürütülmesi bittikten sonra hangi adrese döneceğini bilir. İkinci olarak, çağrı EIP nin geçerli değerini adres işleneniyle belirtilen değerle değiştirir. Ardından CPU, EIP ile gösterilen yeni bellek konumunda çalışmaya başlar. Çağrının call bir işlev fonksiyon çağrısı başlatması gibi , ret komutu da onu tamalar. ret komutunu , burada gösterildiği gibi kendi başına ve herhangi bir parametre olmadan kullanabilirsiniz.

ret

Çağrıldığında, ret, çağrı talimatı çağrıldığında çağrı talimatının yığına ittiği kaydedilmiş program sayacı değeri (EIP) olmasını beklediğimiz yığından en yüksek değeri çıkarır. Ardından, açılan program sayacı değerini tekrar EIP'ye yerleştirir ve yürütmeyi sürdürür. Jmp komutu, çağrıdan daha basit çalışan bir başka önemli kontrol akışı yapısıdır. EIP'yi kaydetme konusunda endişelenmek yerine, jmp basitçe CPU'ya kendi parametresi olarak belirtilen bellek adresine gitmesini ve orada çalışmaya başlamasını söyler.

gibi, jmp ve çağrı talimatlarının koşullu bir şekilde yürütülmesini nasıl sağlayabileceğinizi merak ediyor olabilirsiniz. Cevap, x86 derlemesinin if, then, else, else if, vb. Gibi üst düzey yapılara sahip olmamasıdır. Bunun yerine, bir programın kodu içindeki bir adrese dallanma tipik olarak iki talimat gerektirir: bazı kayıtlardaki değeri bazı test değerleriyle karşılaştıran ve bu testin sonucunu EFLAGS kaydında saklayan bir cmp talimatı ve bir koşullu dallanma talimatı. conditional branch instructions

bu koşullu dal komutlarının çoğu progrmaın bir bellek adresine atlamasına izin veren bir j ile başlar ve tets edilen koşulu temsil eden harflerle sabitlenmiştir.  Örneğin jge programa şundan büyükveya eşitse atlamsını söyler. Bu test edilen kayıt defterindeki register değerin test değerinden büyük veya ona eşit olması gerektiği anlamına gelir. Cmp taliamtı aşağıdaki sözdizimini kullanır:

cmp register, memory location, or literal,    register, memory location, or literal(gerçek tam basın)

Daha önce belirtildiği gibi, cmp, belirtilen genel amaçlı kayıt defterindeki değeri değerle karşılaştırır ve ardından bu karşılaştırmanın sonucunu EFLAGS kaydında saklar. Çeşitli koşullu jmp talimatları daha sonra aşağıdaki gibi çağrılır:

Gördüğünüz gibi, herhangi bir sayıdaki koşullu test talimatının önüne j ekleyebiliriz. Örneğin, yalnızca test edilen değer kayıt defterindeki değerden büyük veya ona eşitse atlamak için aşağıdaki talimatı kullanın:

jge address

Çağrı ve ret komutlarının aksine, jmp komut ailesinin program yığınına asla dokunmadığını unutmayın. Aslında, jmp talimat ailesi durumunda, x86 programı kendi yürütme akışını takip etmekten ve hangi adresleri ziyaret ettiği ve belirli bir dizi talimat yürütüldükten sonra nereye dönmesi gerektiği hakkındaki bilgileri potansiyel olarak kaydetmekten veya silmekten sorumludur. .

\*Basic Blocks and Control Flow Graphs : Temel Bloklar ve Kontrol Akış Grafikleri:

Bir metin düzenleyicide kodlarını kaydırdığımızda x86 programları sıralı görünse de, aslında döngüleri, koşullu dalları ve koşulsuz dalları vardır (kontrol akışı). Bunların tümü her x86 programına bir ağ yapısı network structure verir. Bunun nasıl çalıştığını görmek için Liste 2-1'deki basit simple toy assembly programını  kullanalım.

setup: # sonraki satırda talimat adresi için duran sembol

(1) mov eax, 10 //eax da 10 değeri saklanır

loopstart: # sonraki talimatın adresi için duran sembol

(2) sub eax, 1 herhangi bir döngü için eax 1 azalır

(3) cmp 0, eax // 0 a ulaşınca karışlaştııryor program döngünün dışına çıkar

jne $loopstart

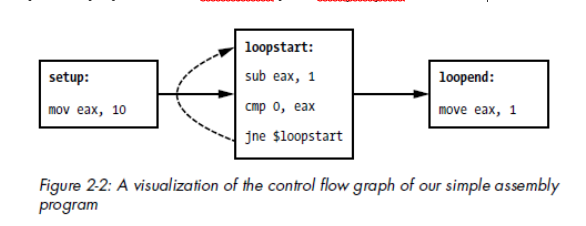
loopend: # sonraki talimatın adresi için duran sembol

mov eax, 1

# daha fazla kod buraya gider

Son olarak, EAX 0  değerine ulaştığında, program döngüden çıkar. Kontrol akış grafiği analizi dilinde in the language of control flow graph analysis , bu talimatların üç temel bloktan oluştuğunu düşünebiliriz. Temel bir blok  basic block , her zaman bitişik olarak yürütüleceğini bildiğimiz bir talimatlar dizisidir. Başka bir deyişle, temel bir blok her zaman bir dallanma talimatı veya bir dalın hedefi olan bir talimatla biter ve her zaman programın giriş noktası olarak adlandırılan programın ilk talimatıyla veya bir dal hedefiyle başlar. Liste 2-1'de, basit programımızın temel bloklarının nerede başladığını ve bittiğini görebilirsiniz. İlk temel blok, kurulum  setup altındaki hareket 10  mov eax 10 komutundan oluşur :. İkinci temel blok loopstart , döngü başlangıcı altında 1'den jne $ loopstart'a kadar olan alt eax ile başlayan satırlardan oluşur ve üçüncüsü, döngü sonunda loopend  1 mov eax  harekette başlar :. Şekil 2-2'deki grafiği kullanarak temel bloklar arasındaki ilişkileri görselleştirebiliriz. (Grafik graph  terimini ağ  network terimiyle eşanlamlı olarak kullanıyoruz; bilgisayar biliminde bu terimler birbirinin yerine kullanılabilir.)

cmp komutu sub ile aynı işi yapar ama sonucu herhangi bir kaydediciye yüklemez. compare yani karşılaştor demektir.



Bir temel blok başka bir temel bloğa akabilirse, onu Şekil 2-2'de gösterildiği gibi bağlarız. Şekil, kurulum temel bloğunun, döngü sonu temel bloğuna geçmeden önce 10 kez tekrar eden döngü başlatma temel bloğuna yol açtığını göstermektedir. Gerçek dünya programları, bunun gibi kontrol akış grafiklerine sahiptir, ancak binlerce temel blok ve binlerce ara bağlantı ile çok daha karmaşıktır.

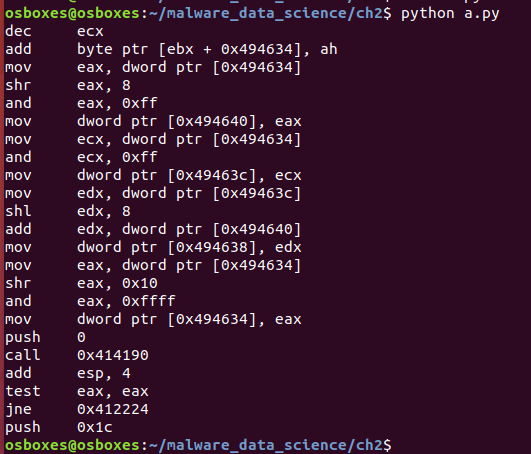
\*Disassembling ircbot.exe Using pefile and capstone:Pefile ve capstone kullanarak ircbot.exe'yi sökme

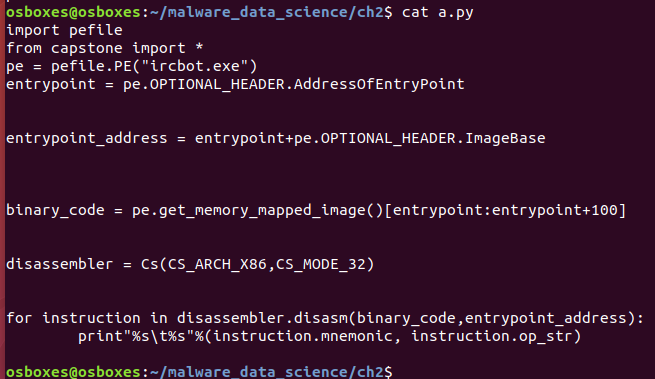
Artık montaj dilinin temellerini iyi anladığınıza göre, ircbot.exe’nin montaj kodunun ilk 100 baytını doğrusal sökme linear disassembly kullanarak parçalarına ayıralım.

Bunu yapmak için, açık kaynak Python kitaplıkları pefile (Bölüm 1'de tanıtıldı) ve 32-bit x86 ikili kodunu parçalarına ayırabilen açık kaynaklı bir sökme kitaplığı olan capstone kullanacağız. Bu kitaplıkların her ikisini de aşağıdaki komutları kullanarak pip ile kurabilirsiniz:

pip install pefile pip install capstone

Bu iki kitaplık yüklendikten sonra, Listing 2-2'deki kodu kullanarak ircbot.exe'yi parçalarına ayırmak için disassemble  bunlardan yararlanabiliriz.





#!/usr/bin/python

import pefile

from capstone import \*

# load the target PE file

pe = pefile.PE("ircbot.exe")

# program başlığından program giriş noktasının adresini alın

entrypoint = pe.OPTIONAL\_HEADER.AddressOfEntryPoint

# giriş kodunun belleğe yükleneceği hesap bellek adresi

entrypoint\_address = entrypoint+pe.OPTIONAL\_HEADER.ImageBase

# PE dosya nesnesinden ikili kodu alın

binary\_code = pe.get\_memory\_mapped\_image()[entrypoint:entrypoint+100]

# 32 bit x86 ikili kodunu sökmek için sökücüyü disassemble başlatın

disassembler = Cs(CS\_ARCH\_X86, CS\_MODE\_32)

# kodu parçalarına ayırın disassemble

for instruction in disassembler.disasm(binary\_code, entrypoint\_address):

print "%s\t%s" %(instruction.mnemonic, instruction.op\_str)

\*Factors That Limit Static Analysis:Statik analizi Sınırlayan Faktörler

Bazı taktikler yapabiliriz zararlı yazılımı yazan kişi bunlara bakalım:

-Packing:

Kötü amaçlı yazılım paketleme, kötü amaçlı yazılım yazarlarının kötü amaçlı programlarının büyük kısmını kötü amaçlı yazılım analistlerine anlaşılmaz görünecek şekilde sıkıştırdığı, şifrelediği veya başka şekilde karıştırdığı süreçtir. Kötü amaçlı yazılım çalıştırıldığında, kendisini paketinden çıkarır ve ardından yürütmeye başlar. Kötü amaçlı yazılımları paketlemenin en bariz yolu, kötü amaçlı yazılımı güvenli bir ortamda çalıştırmaktır; bu, Bölüm 3'te ele alacağım dinamik bir analiz tekniğidir.

Not:

Yazılım paketleme, meşru nedenlerle zararsız yazılım yükleyicileri tarafından da kullanılır. İyi huylu yazılım yazarları, yazılım yükleyici indirme boyutlarını azaltmak için program kaynaklarını sıkıştırmalarına izin verdiği için kodlarını teslim etmek için paket kullanırlar. Aynı zamanda, iş rakiplerinin tersine mühendislik girişimlerini engellemelerine yardımcı olur ve birçok program kaynağını tek bir yükleyici dosyası içinde bir araya getirmek için uygun bir yol sağlar.

\*Resource Obfuscation: Kaynak  Gizleme

Kötü amaçlı yazılım yazarlarının kullandığı bir başka anti-tespit, anti-analiz tekniği de kaynak gizlemedir. Dizeler ve grafik görüntüler gibi program kaynaklarının diskte depolanma şeklini gizler ve ardından kötü amaçlı program tarafından kullanılabilmeleri için çalışma zamanında bunları gizler. Örneğin, basit bir karmaşıklaştırma, PE kaynakları bölümünde depolanan görüntüler ve dizelerdeki tüm baytlara 1 değeri eklemek ve ardından çalışma zamanında tüm bu verilerden 1 çıkarmak olabilir. Elbette, burada herhangi bir sayıda şaşırtma mümkündür ve bunların tümü, statik analiz kullanarak kötü amaçlı yazılım ikili dosyasını anlamaya çalışan kötü amaçlı yazılım analistleri için hayatı zorlaştırır. Paketlemede olduğu gibi, kaynak gizlemesinin bir yolu, kötü amaçlı yazılımı güvenli bir ortamda çalıştırmaktır. Bu bir seçenek olmadığında, kaynak gizlemesini azaltmanın tek yolu, kötü amaçlı yazılımın kaynaklarını nasıl gizlediğini anlamak ve bunları manuel olarak açığa çıkarmaktır; bu, profesyonel kötü amaçlı yazılım analistlerinin sıklıkla yaptığı şeydir.

\*Anti-diassembly Techniques: diassembly demontajı önleme teknikleri

Kötü amaçlı yazılım yazarları tarafından kullanılan üçüncü bir anti-algılama, anti-analiz teknikleri grubu sökme önleme teknikleridir. Bu teknikler, kodu kötü amaçlı yazılım analistlerinden gizlemek veya kötü amaçlı yazılım analistlerinin diskte depolanan bir kod bloğunun gerçekte olduğundan farklı talimatlar içerdiğini düşünmesini sağlamak için son teknoloji sökme tekniklerinin doğasında var olan sınırlamalardan yararlanmak için tasarlanmıştır. Parçalara ayırmayı önleme tekniğine bir örnek, kötü amaçlı yazılım yazarının çözücülerinin farklı bir talimat olarak yorumlayacağı bir bellek konumuna dallanmayı içerir, bu da kötü amaçlı yazılımın gerçek talimatlarını tersine mühendislerden gizler. Demontajı önleme tekniklerinin büyük bir potansiyeli vardır ve bunlara karşı savunma yapmanın mükemmel bir yolu yoktur. Uygulamada, bu tekniklere karşı iki ana savunma, kötü amaçlı yazılım örneklerini dinamik bir ortamda çalıştırmak ve bir kötü amaçlı yazılım örneğinde sökme önleme stratejilerinin nerede ortaya çıktığını ve bunları nasıl atlayacağını manuel olarak bulmaktır.

Dynamically Downloaded Data: dinamik olarak indirilen veriler

Dinamik Olarak İndirilen Veriler

Kötü amaçlı yazılım yazarlarının kullandığı son bir anti-analiz teknikleri sınıfı, harici olarak veri ve kod sağlamayı içerir. Örneğin, bir kötü amaçlı yazılım örneği, kötü amaçlı yazılım başlatma sırasında harici bir sunucudan dinamik olarak kod yükleyebilir. Durum böyleyse, statik analiz bu tür koda karşı işe yaramaz. Benzer şekilde, kötü amaçlı yazılım, başlangıç sırasında harici sunuculardan şifre çözme anahtarları elde edebilir ve ardından bu anahtarları kötü amaçlı yazılımın yürütülmesinde kullanılacak verilerin veya kodun şifresini çözmek için kullanabilir.

Açıktır ki, kötü amaçlı yazılım endüstriyel güçte bir şifreleme algoritması kullanıyorsa, şifrelenmiş verileri ve kodu kurtarmak için statik analiz yeterli olmayacaktır. Bu tür anti-analiz ve anti-algılama teknikleri oldukça güçlüdür ve bunların etrafından dolaşmanın tek yolu, harici sunuculardaki kodu, verileri veya özel anahtarları bir şekilde elde etmek ve ardından bunları kişinin söz konusu kötü amaçlı yazılımın analizinde kullanmaktır.

ÖZet: SUMmary:

Bu bölümde x86 montaj kodu analizi tanıtıldı ve açık kaynaklı Python araçlarını kullanarak ircbot.exe üzerinde sökme tabanlı statik analizi nasıl gerçekleştirebileceğimiz gösterildi. Bunun x86 derlemesinde tam bir başlangıç olması amaçlanmasa da, belirli bir kötü amaçlı yazılım derleme dökümünde neler olup bittiğini anlamak için bir başlangıç yeriniz olduğu için artık kendinizi yeterince rahat hissetmelisiniz. Son olarak, kötü amaçlı yazılım yazarlarının parçalarına ayırma ve diğer statik analiz tekniklerine karşı nasıl savunma yapabileceklerini ve bu analiz önleme ve algılama önleme stratejilerini nasıl azaltabileceğinizi öğrendiniz. Bölüm 3'te, statik kötü amaçlı yazılım analizinin birçok zayıf yönünü oluşturan dinamik kötü amaçlı yazılım analizi yapmayı öğreneceksiniz.