BAŞKENT ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI
BİLGİSAYAR MÜHENDİSLİĞİ TEZLİ YÜKSEK LİSANS PROGRAMI

TERSİNE MÜHENDİSLİK YÖNTEMLERİ VE BİLGİSAYAR UYGULAMALARI ANALİZİ

HAZIRLAYAN

GÜNEY UĞURLU

YÜKSEK LİSANS TEZİ

ANKARA - 2022

BAŞKENT ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI
BİLGİSAYAR MÜHENDİSLİĞİ TEZLİ YÜKSEK LİSANS PROGRAMI

TERSİNE MÜHENDİSLİK YÖNTEMLERİ VE BİLGİSAYAR UYGULAMALARI ANALİZİ

HAZIRLAYAN

GÜNEY UĞURLU

YÜKSEK LİSANS TEZİ

TEZ DANIŞMANI

DR. ÖĞR. ÜYESİ KORAY AÇICI

ANKARA - 2022

BAŞKENT ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

Bilgisayar Mühendisliği Anabilim Dalı Bilgisayar Mühendisliği Tezli Yüksek Lisans Programı çerçevesinde Güney Uğurlu tarafından hazırlanan bu çalışma, aşağıdaki jüri tarafından Yüksek Lisans Tezi olarak kabul edilmiştir.

Tez Savunma Tarihi: 29 / 07 / 2022

Tez Adı: Tersine Mühendislik Yöntemleri ve Bilgisayar Uygulamaları Analizi

Tez Jüri Üyeleri	**İmza**
Doç. Dr. Mehmet Serdar GÜZEL, Ankara Üniversitesi	……..
Dr. Öğr. Üyesi Koray AÇICI, Başkent Üniversitesi	…….
Dr. Öğr. Üyesi Çağatay Berke ERDAŞ, Başkent Üniversitesi	……..

ONAY

…..…….……
Fen Bilimleri Enstitüsü Müdürü
Tarih : … / … / ….…….

BAŞKENT ÜNİVERSİTESİ

FEN BİLİMLER ENSTİTÜSÜ

YÜKSEK LİSANS TEZ ÇALIŞMASI ORİJİNALLİK RAPORU

Tarih: … / … / 20…

Öğrencinin Adı, Soyadı: Güney UĞURLU

Öğrencinin Numarası: 22010219

Anabilim Dalı: Bilgisayar Mühendisliği Anabilim Dalı

Programı: Bilgisayar Mühendisliği

Danışmanın Unvanı/Adı, Soyadı: Dr. Öğr. Üyesi Koray AÇICI

Tez Başlığı: Tersine Mühendislik Yöntemleri ve Bilgisayar Uygulamaları Analizi

Yukarıda başlığı belirtilen Yüksek Lisans tez çalışmamın; Giriş, Ana Bölümler ve Sonuç Bölümünden oluşan, toplam 104 sayfalık kısmına ilişkin, 26 / 07 / 2022 tarihinde tez danışmanım tarafından Turnitin adlı intihal tespit programından aşağıda belirtilen filtrelemeler uygulanarak alınmış olan orijinallik raporuna göre, tezimin benzerlik oranı %3'tür. Uygulanan filtrelemeler:

1. Kaynakça hariç
2. Alıntılar hariç
3. Beş (5) kelimeden daha az örtüşme içeren metin kısımları hariç

"Başkent Üniversitesi Enstitüleri Tez Çalışması Orijinallik Raporu Alınması ve Kullanılması Usul ve Esaslarını" inceledim ve bu uygulama esaslarında belirtilen azami benzerlik oranlarına tez çalışmamın herhangi bir intihal içermediğini; aksinin tespit edileceği muhtemel durumda doğabilecek her türlü hukuki sorumluluğu kabul ettiğimi ve yukarıda vermiş olduğum bilgilerin doğru olduğunu beyan ederim.

Öğrenci İmzası:…….

ONAY

Tarih: … / … / 2022

Öğrenci Danışmanı Dr. Öğr. Üyesi Koray AÇICI

TEŞEKKÜR

Yazar, bu çalışmanın gerçekleşmesinde katkılarından dolayı, aşağıda adı geçen kişilere içtenlikle teşekkür eder.

Sayın Dr. Öğr. Üyesi Koray AÇICI'ya (tez danışmanı), çalışmanın sonuca ulaştırılmasında ve karşılaşılan güçlüklerin aşılmasında her zaman yardımcı ve yol gösterici olduğu için…
Değerli babam Halil UĞURLU'ya, her zaman yanımda olduğu ve beni her konuda desteklediği için…
Değerli annem Gülten UĞURLU'ya, her zaman yanımda olduğu ve beni her konuda desteklediği için…
Değerli hayat arkadaşım Gülce ÇÖKENOĞLU'na, bana her zaman destek olduğu için…
Sayın Dr. Öğr. Üyesi Emre SÜMER'e, tez çalışmam esnasında destekleri ve yardımlarından dolayı…
Sayın Prof. Dr. Nizami GASİLOV'a, tez çalışmam esnasında destekleri ve yardımlarından dolayı…
Değerli yakın arkadaşlarıma, her zaman yanımda oldukları ve beni destekledikleri için…

ÖZET

Güney UĞURLU

TERSİNE MÜHENDİSLİK YÖNTEMLERİ VE BİLGİSAYAR UYGULAMALARI ANALİZİ

Başkent Üniversitesi Fen Bilimleri Enstitüsü

Bilgisayar Mühendisliği Anabilim Dalı

2022

Tersine mühendislik, ticari veya askeri avantaj için donanım ve yazılım analizine dayanmaktadır. Tersine mühendislik süreci, eserin orijinal üretimlerinde yer alan prosedürler hakkında çok az veya hiç ek bilgisi olmayan ürünlerden tasarım özelliklerini çıkarmak için yapılan bir analizin yanı sıra kendi içinde bir kopya oluşturmak ya da eseri bir şekilde değiştirmekle ilgilidir. Yazılımlarda tersine mühendislik, bir sistemi veya bir sistemin parçalarını alma ve bunların işlevselliğini ve tasarımını analiz etme sürecidir. Bir sistemin parçalarını daha yüksek bir soyutlama düzeyinde analiz etmenin ve yeniden yapılandırmanın bir yolu olarak tersine mühendisliğe başvurulur. Ancak, tersine mühendislik bundan daha fazlası için kullanılır. Yeni bir tür kötü amaçlı yazılımın ortaya çıktığını var sayarsak, kaynak kodu bilinmeyen yeni tehditlere karşı önlemlerin alınabilmesi için kötü amaçlı yazılımın içeriğini analiz etmek, nasıl çalıştığını anlamak ve aynı zamanda fikri mülkiyeti kötü niyetli kişilere karşı korumak için tersine mühendislik tekniklerine ve becerilerine ihtiyaç duyulur. Ayrıca tersine mühendislik yöntemlerini ve yaklaşımlarını bilmek yazılım korsanlığına karşı daha etkili yöntemler ortaya koymak için çok önemlidir. Tezin amacı, tersine mühendislik yöntemleri ve bu yöntemlerin nasıl aşılacağı hakkında bilgi vermeyi amaçlamıştır. X86 platformu ve assembly dili, taşınabilir yürütülebilir dosya formatı, Windows uygulama programlama arayüzü, hata ayıklamayı önleme ve bu yöntemleri aşma, paketleyiciler ve paketten çıkarma teknikleri gibi tersine mühendislik için önemli olan konulara değinilmiş ve yapılan çalışmalarda uygulamalı olarak gösterilmiştir.

ANAHTAR KELİMELER: Tersine Mühendislik, Hata Ayıklamayı Önleme, Paketten Çıkarma, Windows Uygulama Programlama Arayüzü, Assembly Dili.

ABSTRACT

Güney UĞURLU

REVERSE ENGINEERING METHODS AND COMPUTER APPLICATIONS ANALYSIS

Baskent University Institution of Science and Engineering

Department of Computer Engineering

2022

Reverse engineering relies on hardware and software analysis for commercial or military advantage. The reverse engineering process is about creating a copy in itself or modifying the work in some way, as well as an analysis to extract design features from products that have little or no additional knowledge of the procedures involved in their original production. Reverse engineering of softwares is the process of taking a system or parts of a system and analyzing their functionality and design. Reverse engineering is resorted to as a way to analyze and reconstruct parts of a system at a higher level of abstraction. However, reverse engineering is used for more than that. Assuming that you have emerged a new kind of malware, reverse engineering techniques and skills are needed to analyze the content of malware, understand how it works, and at the same time protect intellectual property against malicious individuals, so that countermeasures against new threats, the source code of which are unknown, can be taken. In addition, knowing the reverse engineering methods and approaches is very important to come up with more effective methods against software piracy. The aim of the thesis is to give information about reverse engineering methods and how to overcome these methods. The important issues for reverse engineering such as x86 platform and assembly language, portable executable file format, Windows application programming interface, anti-debugging and bypassing these methods, packers and unpacking techniques are mentioned and shown in practice in the studies.

KEYWORDS: Reverse Engineering, Anti-Debugging, Unpacking, Windows Application Programming Interface, Assembly Language.

İÇİNDEKİLER

TEŞEKKÜR i

ÖZET ii

ABSTRACT iii

İÇİNDEKİLER iv

TABLOLAR LİSTESİ viii

ŞEKİLLER LİSTESİ ix

SİMGELER VE KISALTMALAR LİSTESİ xv

1. GİRİŞ 1

1.1 Tersine Mühendislik Nedir? 1

1.2 Yazılımlarda Tersine Mühendislik 2

1.3 Tersine Mühendislik Süreçleri 3

1.3.1 Onay alınması 3

1.3.2 Statik analiz 3

1.3.3 Dinamik analiz 3

1.1.1 Raporlama 4

1.2 Tersine Mühendislik Araçları 4

1.2.1 Hex düzenleyiciler (hex editors) 4

1.2.2 İkili biçim - analiz araçları (binary format - analysis tools) 4

1.2.3 Ayrıştırıcılar (dissassemblers) 5

1.2.4 Hata ayıklayıcılar (debuggers) 5

1.2.5 Geri derleyiciler (decompilers) 6

1.2.6 Import reconstruction 6

2. X86 PLATFORMU VE ASSEMBLY DİLİ 7

2.1 Yazmaçlar 8

Bayraklar 8

2.1.1 Genel amaçlı yazmaçlar 9

2.1.2 Segment yazmaçları 10

2.1.3 Talimat işaretçisi yazmacı 11

2.1.4 Bayrak yazmaçları 11

2.1.5 Kontrol yazmaçları 14

2.2 Veri Türleri 15

2.3 Komut Setine Genel Bakış 16

2.3.1 Talimat işlenenleri (instruction operands) 16
2.3.2 En Yaygın kullanılan X86 Talimatları 17
2.4 Yürütme Akışını Kontrol Etme 20
2.4.1 Koşulsuz dallanma 20
2.4.2 Koşullu dallanma 22
2.5 Yığın (Stack) İşlemleri 24
2.6 X86 - 64 25
3. ÖNEMLİ TEMEL BİLGİLER 27
3.1 PE Dosya Formatı 27
3.1.1 MS-DOS başlığı 28
3.1.2 IMAGE_NT_HEADERS yapısı 29
3.1.3 IMAGE_FILE_HEADER yapısı 29
3.1.4 IMAGE_OPTIONAL_HEADER yapısı 30
3.1.5 IMAGE_DATA_DIRECTORY yapısı 31
3.1.6 Bölüm tablosu (section table) 31
3.1.7 Export address table 33
3.1.8 Import address table 33
3.1.9 Bellek ve dosya hizalama 34
3.2 Windows Uygulama Programlama Arayüzü (WinAPI) 35
3.2.1 Önemli windows dinamik bağlantı kitaplıkları 35
3.2.2 Windows uygulama programlama arabiriminde (API) sık kullanılan işlevler 36
3.2.3 İş parçacığı ortamı bloğu (thread environment block – TEB) 37
3.2.4 İşlem ortamı bloğu (process environment block – PEB) 38
3.3 Endianness 38
3.3.1 Big-endian 39
3.3.2 Little-endian 39
3.4 İşaretli Sayı Gösterimleri 39
3.5 Çağırma Kuralları (Calling Conventions) 40
3.6 Kesme Noktaları (Breakpoints) 41
3.7 Code Caves (Kod Mağaraları) 42
4. TERSİNE MÜHENDİSLİĞİ ÖNLEME YÖNTEMLERİ VE BUNLARI AŞMAK 43
4.1 Hata Ayıklama Bayrakları 43
4.1.1 BeingDebugged / IsDebuggerPresent 43

4.1.2 CheckRemoteDebuggerPresent / NtQuerryInformationProcess 44
4.1.3 NTGlobalFlag 44
4.1.4 NtQuerySystemInformation 45
4.2 Nesne Tanıtıcıları (Object Handles) 45
4.2.1 OpenProcess / SeDebugPrivilege 45
4.2.2 CloseHandle 46
4.2.3 LoadLibrary / CreateFile 46
4.2.4 NtQuerryObject 47
4.3 Zaman Temelli İşlevler 47
4.3.1 RDTSC (read time-stamp counter) 47
4.3.2 QueryPerormanceCounter 48
4.3.3 GetTickCount 48
4.3.4 timeGetTime 49
4.4 İşlem Belleği 49
4.4.1 ReadFile 49
4.4.2 Kod checksum hesaplaması ile yama tespit etme 49
4.5 Hata Ayıklayıcı ile Etkileşim 49
4.5.1 NtSetInformationThread / ThreadHideFromDebugger 50
4.5.2 BlockInput 50
4.6 Anti-Dumping 51
4.6.1 Nanomitler (nanomites) 51
4.6.2 Stolen bytes (stolen codes) 51
4.6.3 Kendi kendine eşlemeyi kaldırma (self-unmapping) 51
4.6.4 Sanallaştırma 52
4.6.5 PE başlığını kaldırma 52
4.6.6 Guard Pages 52
4.7 Obfuscation 53
4.7.1 Kontrol akışı gizleme (control flow obfuscation) 53
4.7.2 Opak yüklemler (opaque predicates) 53
4.7.3 Talimat permutasyonu (instruction permutation) 54
4.7.4 Şifreleme 54
4.8 TLS Callback 55
4.9 Mutasyon 55
4.10 İşlem Enjeksiyonu 55

5. YÜRÜTÜLEBİLİR PAKETLEYİCİLER VE KORUYUCULAR (EXECUTABLE PACKERS AND PROTECTORS) 57
5.1 Paketleyici Algılama Yöntemleri 57
5.1.1 İmza tabanlı paketleyici algılama 57
5.1.2 Bilgi yoğunluğu (entropi) eşikleri 58
5.1.3 Şablon tanıma 58
5.2 Paketten Çıkarma (Unpacking) ve Orijinal Giriş Noktasına (OEP) Ulaşma Yöntemleri 58
5.2.1 ESP trick 59
5.2.2 Özel durum (exception) sayımı ve belleğe erişimde kesme noktası 59
5.2.3 Yığında geri izleme 60
5.2.4 Yapılandırılmış özel durum işleme (structured exception handling / SEH) izleme 60
5.3 Bilinen Giriş Noktaları 60
6. YAPILAN ÇALIŞMALAR 63
6.1 Örnek 1 63
6.1.1 IsDebuggerPresent ve PEB->BeingDebugged yöntemlerini aşma 63
6.1.2 NtGlobalFlag yöntemini aşma 66
6.1.3 CheckRemoteDebuggerPresent yöntemini aşma 68
6.1.4 ScyllaHide Eklentisi 69
6.2 Örnek 2 71
6.2.1 Manuel olarak yeni bölüm ekleme 72
6.2.2 Araştırma ve hata ayıklamayı önlemeden kurtulma aşaması 76
6.2.3 Kod mağarası kullanarak seri anahtarı oluşturma 84
6.3 Örnek 3 89
6.3.1 İşlemin yaratılma süreci, bölümler ve eşleme işlemleri 89
6.3.2 Paketten çıkarma ve içe aktarımların düzeltilmesi 94
6.3.3 IDA Pro ile statik analiz ve içe aktarımların tekrar düzeltilmesi 98
7. SONUÇ 104
KAYNAKLAR 105

TABLOLAR LİSTESİ

Sayfa

Tablo 1.1. Statik analiz ve dinamik analizin farkları…….4

Tablo 2.1. Yazmaçlar ve tanımları…….8

Tablo 2.2. Genel amaçlı yazmaçlar ve geleneksel kullanımları……....10

Tablo 2.3. Segment yazmaçları ve tanımları…….11

Tablo 2.4. Durum Bayrakları…….……12

Tablo 2.5. Sistem bayrakları…….....13

Tablo 2.6. Kontrol Yazmaçları……..14

Tablo 2.7. Veri Tipleri……...15

Tablo 2.8. Operand türleri, örnekler ve karşılıkları……...……16

Tablo 2.9. En yaygın kullanılan X86 talimatları……...17

Tablo 2.10. Koşullu atlamalar ve bayrak durumları…….....23

Tablo 3.1 Bölüm adları ve içerikleri…….....32

Tablo 3.2. Big-endian temsili…….....39

Tablo 3.3. Little-endian temsili……..39

Tablo 3.4. İşaretli ve işaretsiz sayı gösterimleri…….40

Tablo 3.5 Çağırma kuralları ve özellikleri…….41

Tablo 4.1. NTGlobalFlag bayrakları ve değerleri……..44

ŞEKİLLER LİSTESİ

Sayfa

Şekil 1.1. IDA Pro ile ayrıştırma işlemi…….….5

Şekil 2.1. Genel amaçlı yazmaçlar…….….9

Şekil 2.2. Assembly dili sözdizimi……16

Şekil 2.3. Yığın işlemleri örnek kod……..24

Şekil 2.4. Yığın kullanımı……..25

Şekil 2.5. x64 kayıt seti……..26

Şekil 3.1. PE dosya yapısı……..27

Şekil 3.2. MS-DOS başlığı yapısı……..…28

Şekil 3.3. IMAGE_NT_HEADERS yapısı……29

Şekil 3.4. IMAGE_FILE_HEADER yapısı……...29

Şekil 3.5. IMAGE_OPTIONAL_HEADER yapısı……...…30

Şekil 3.6. IMAGE_DATA_DIRECTORY yapısı……..…31

Şekil 3.7. IMAGE_SECTION_HEADER yapısı……...……32

Şekil 3.8. IMAGE_IMPORT_DESCRIPTOR yapısı…….....33

Şekil 3.9. IMAGE_THUNK_DATA yapısı……...34

Şekil 3.10. IMAGE_IMPORT_BY_NAME yapısı……34

Şekil 3.11. İş parçacığı ortamı bloğu yapısı……...…37

Şekil 3.12. İşlem ortamı bloğu yapısı…….38

Şekil 3.13. Orijinal yazılım ve kod mağarası eklenmiş yazılım…….42

Şekil 4.1. BeingDebugged bayrağının kontrolü…….43

Şekil 4.2. OpenProcess kullanımı…….……..46

Şekil 4.3. RDTSC kullanımı……...48

Şekil 4.4. QueryPerformanceCounter kullanımı……...…48

Şekil 4.5. NtSetInformationThread() parametreleri……...…50

Şekil 4.6. opak isimli alt program……..53

Şekil 4.7. add eax, ebx talimatının karşılığı……...54

Şekil 4.8. Şifre çözme rutini……...54

Şekil 5.1. Microsoft Visual C++ 6.0 ile derlenmiş bir programın giriş noktası…….60

Şekil 5.2. Microsoft Visual C++ 7.x ile derlenmiş bir programın giriş noktası…….61

Şekil 5.3. Microsoft Visual C++ 8.0 – 9.0 ile derlenmiş bir programın giriş noktası……61

Şekil 5.4. Borland C++ Builder ile derlenmiş bir programın giriş noktası……61

Şekil 5.5. Dev C++ ile derlenmiş bir programın giriş noktası……...…62

Şekil 5.6. Borland Delphi 6.0 – 7.0 ile derlenmiş bir programın giriş noktası…….…62

Şekil 5.7. Microsoft Visual Basic 5.0 – 6.0 ile derlenmiş bir programın giriş noktası…..62

Şekil 5.8. MASM32/TASM32 ile derlenmiş bir programın giriş noktası……...…….....62

Şekil 6.1. Anti Debug Arayüzü……..63

Şekil 6.2. IsDebuggerPresent() ve PEB->BeingDebugged butonları etkin …….....63

Şekil 6.3. IsDebuggerPresent() kesme noktası komutu…….64

Şekil 6.4. 778A20D0 kesmesi…….…….....64

Şekil 6.5. IsDebuggerPresent() kod bloğu……...…….64

Şekil 6.6. Follow in Dump > Address: EAX+2 seçenekleri…….…….64

Şekil 6.7. 00C76002 adresinin dump sekmesinde gösterimi…….65

Şekil 6.8. Dump sekmesinde 0 bayt ile değişim……65

Şekil 6.9. İlk değişikliğe uğramış anti debug arayüzü…….……..66

Şekil 6.10. NtGlobalFlag butonu etkin…….…….66

Şekil 6.11. NtGlobalFlag kesme noktası……..….66

Şekil 6.12. Follow in Dump > Address :EAX+68 seçenekleri…….……67

Şekil 6.13. NtGlobalFlag alanın değeri……..…..67

Şekil 6.14. İkinci değişikliğe uğramış anti debug arayüzü……67

Şekil 6.15. CheckRemoteDebuggerPresent butonu etkin……..…68

Şekil 6.16. CheckRemoteDebuggerPresent kesme noktası……...…68

Şekil 6.17. CheckRemoteDebuggerPresent kesme noktası……...68

Şekil 6.18. Nop talimatı ile doldurma……...69

Şekil 6.19. Üçüncü değişikliğe uğramış anti debug arayüzü……69

Şekil 6.20. Plugins > ScyllaHide > Options seçenekleri……..69

Şekil 6.21. ScyllaHide arayüzü ve seçenekler……..70

Şekil 6.22. ScyllaHide ve anti debug arayüzü……..…70

Şekil 6.23. Code cave arayüzü ve hata mesajı……..…71

Şekil 6.24. CFF Explorer ile bölüm bilgileri……71

Şekil 6.25. AdressOfEntryPoint ve ImageBase değeri…….……72

Şekil 6.26. Code cave giriş noktası……..…..72

Şekil 6.27. Section headers sekmesi…….73

Şekil 6.28. Add section (empty space) seçeneği……...73

Şekil 6.29. Eklenecek bölümün boyutu……...….74

Şekil 6.30. .ornekcc yeni bölümü……74

Şekil 6.31. SectionAlignment ve FileAlignment alanları…….……74

Şekil 6.32. Change section flags seçeneği……74

Şekil 6.33. Section flags penceresi……...……75

Şekil 6.34. Rebuild image size seçeneği……...75

Şekil 6.35. Yeni bölümün memory map sekmesindeki görünüşü……76

Şekil 6.36. Hata ayıklayıcıda karşılaşılan hata…….…76

Şekil 6.37. AdressOfEntryPoint ve ImageBase değeri…….……76

Şekil 6.38. Seçili kod bloğu……..77

Şekil 6.39. Nop talimatları ile değişim……77

Şekil 6.40. Patch Penceresi……...77

Şekil 6.41. Search for > Current Module > Intermodular Calls seçeneği……..….78

Şekil 6.42. MessageBoxA fonksiyonu……78

Şekil 6.43. MessageBoxA kod bloğu……...…78

Şekil 6.44. Hata mesajı…….....79

Şekil 6.45. Ana kod bloğu……79

Şekil 6.46. Kullanıcı adına göre seri numarası üreten kod bloğu…….80

Şekil 6.47. Kullanıcı adı ve seri numarası girdisi…….80

Şekil 6.48. 00401187 alt rutini……..81

Şekil 6.49. Follow in dump seçeneği……81

Şekil 6.50. Dump sekmesindeki oluşan değişiklikler……...82

Şekil 6.51. Şifre çözme alt rutini……...82

Şekil 6.52. Şifre çözme işleminden sonra oluşan değişiklikler…….83

Şekil 6.53.Seri anahtarı ile deneme ve başarılı sonuç……...83

Şekil 6.54. Kod akışı değiştirme……...84

Şekil 6.55. Yeni bölümün görüntüsü……84

Şekil 6.56. MessageBoxA fonksiyonunun C++ dilindeki sözdizimi……....85

Şekil 6.58. Multiline ultimate assembler eklentisi……85

Şekil 6.59. Parametreler……86

Şekil 6.60. Tamamlanmış talimatlar…….87

Şekil 6.61. Talimatların eklenmiş hali……..88

Şekil 6.62. Geçerli seri anahtarı mesaj kutusu……..88

Şekil 6.63. Doğru değerlerin girildiğine dair mesaj kutusu……..89

Şekil 6.64. CreateProcessInternalW ve WriteProcessMemory kesme noktaları……..89

Şekil 6.65. CreateProcessInternalW çağrısında kesme noktası……90

Şekil 6.66. Yığın argümanları……..….90

Şekil 6.67. WriteProcessMemory çağrısında kesme noktası……90

Şekil 6.68. WriteProcessMemory sözdizimi…….90

Şekil 6.69. Yığın Penceresi……..….91

Şekil 6.70. Follow DWORD in current dump seçeneği……91

Şekil 6.71. Dump sekmesi…….91

Şekil 6.72. Follow in Memory Map seçeneği……...92

Şekil 6.73. Dump memory to file seçeneği……...92

Şekil 6.74. icedid_00250000.bin dosyasının PE-bear programı ile açılmış hali……..92

Şekil 6.75. HxD editörü ile açılmış dosya……93

Şekil 6.76. Save the executable as… seçeneği…….94

Şekil 6.77. Giriş noktasının kod bloğu…….94

Şekil 6.78. Donanım kesme noktası eklenmesi……95

Şekil 6.79. 0041E076 adresinde kesmeye uğrama……...95

Şekil 6.80. Orijinal giriş noktası……...96

Şekil 6.81. Scylla eklentisinin arayüzü…….96

Şekil 6.82. İçe aktarımlar ve içe aktarım sayısı……97

Şekil 6.83. IDA pro ile açılmış zararlı yazılım…….98

Şekil 6.84. Zararlı yazılımın geri derlenmiş hali……..98

Şekil 6.85. sub_4013B4 kod bloğu……...99

Şekil 6.86. sub_4018B6 kod bloğu……...99

Şekil 6.87. sub_401178 kod bloğu……...100

Şekil 6.89. Değişen içe aktarım sayısı……..101

Şekil 6.90. Zararlı yazılımın geri derlenmiş kod bloğu……..…..102

Şekil 6.91. Anlamlandırılmış değişken atamaları…….103

SİMGELER VE KISALTMALAR LİSTESİ

ABI	Application Binary Interface
APC	Asynchronous Procedure Calls
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BCD	Binary Coded Decimal
CISC	Complex Instruction Set Computer
COFF	Common Object File Format
CPU	Central Process Unit
DIE	Detect It Easy
DLL	Dynamic Link Library
EIP	Extended Instruction Pointer
EP	Entry Point
ETW	Event Tracing for Windows
EXE	Executable
GPR	General-Purpose Registers
I/O	Input/Output
IA-32	Intel Architecture, 32-bit
IAT	Import Address Table
INT	Import Name Table
IOPL	Imput/Output Privilege Level
LIFO	Last In First Out
MS-DOS	Microsoft Disk Operating System
MSVC	Microsoft Visual C++
MZ	Mark Zbikowski
OEM	Original Equipment Manufacturer
OEP	Original Entry Point
PE	Portable Executable
PEB	Process Environment Block
PID	Process Identifier
R	Read
RDTSC	Read Time-Stamp Counter
RVA	Relative Virtual Address
RW	Read-Write
RWE	Read-Write-Execute
SIMD	Single Instruction / Multiple Data
TEB	Thread Environment Block
TLS	Thread Local Storage
VA	Virtual Address
VM	Virtual Machine

1. GİRİŞ

1.1 Tersine Mühendislik Nedir?

Tersine mühendislik, bir nesnenin ayna görüntüsünü yeniden oluşturmak veya geçmiş bir olayı almak için ölçme, analiz etme, test etme, yeniden icat etme sürecinin yanı sıra yeniden yapılanma ve yeniden üretime götüren bir yol haritasıdır. Tersine mühendislik aynı zamanda orijinal parçanın tasarım amacının korunması için uygulamalı bir bilim sanatıdır.

Tersine mühendislik, ticari kazançlar için yüksek değerli ticari parçaları veya tarihi restorasyon için değersiz eski parçaları yeniden oluşturmak için uygulanabilir. Bu görevi başarmak için mühendisin, orijinal parçanın işlevselliğini anlaması ve karakteristik detaylarını tekrarlama becerisine ihtiyacı vardır. Tarihte çok eski zamanlara dayanmasına rağmen, tersine mühendislikteki son gelişmeler bu teknolojiyi havacılık, otomotiv, elektronik, tıbbi cihaz, spor ekipmanı, oyuncak ve mücevher gibi birçok endüstride kullanılan birincil metodolojilerden biri haline getirmiştir. Dünyanın her yerindeki üreticiler, ürün geliştirmelerinde tersine mühendislik tekniklerini uygulamışlardır. Üç boyutlu lazer tarama ve yüksek çözünürlüklü mikroskop gibi yeni analitik teknolojiler, tersine mühendisliği kolaylaştırmıştır[1].

Birçok meslek kuruluşu, tersine mühendislik tanımlarını kendi bakış açılarından sunmuştur. Üretim Mühendisleri Derneği, tersine mühendislik uygulamasının "bitmiş bir ürün veya süreçle başlayıp, altta yatan yeni teknolojiyi keşfetmek için logosal bir şekilde geriye doğru çalıştığını" belirtir. Bu ifade, icatta daha belirgin roller oynayan yaratma ve yeniliğin aksine, tersine mühendisliğin yeniden icat süreci ve analizine odaklandığını vurgular.

Amerika Birleşik Devletleri askeri el kitabı MILHDBK-115A, tersine mühendisliği, ürünün ekonomik değerini içerecek şekilde daha geniş bir perspektifte tanımlamaktadır: "Bir öğenin, gerekli teknik verileri (fiziksel ve malzeme özellikleri) geliştirmek için fiziksel olarak incelenerek ve ölçülerek bir öğenin işlevsel ve boyutsal olarak çoğaltılması sürecidir." [2]. Bu tanım, tersine mühendisliğin birincil itici gücü olan "rekabet edebilirlik" fikrini ön plana çıkarmaktadır.

Tersine mühendislik öncelikle üç işlev için kullanılır:

- Tasarım verileri mevcut olmayan orijinal ekipman üreticisi (OEM) parçalarını çoğaltmak veya üretmek,

- Orijinal tasarım verilerini bilmeden aşınmış parçaları onarmak veya değiştirmek,
- Analiz için mevcut parçaya dayalı bir model veya prototip oluşturmak.

Tersine mühendislik, contalar, cıvatalar ve somunlar, motor parçaları gibi birçok mekanik parçayı üretmek için kullanılmıştır ve birçok endüstride yaygın olarak kullanılmaktadır.

Tersine mühendisliğin günümüz endüstrisi üzerindeki etkisi, daha az pahalı ürünler sunmanın ve daha fazla rekabeti teşvik etmenin ötesindedir. Aynı zamanda endüstriyel evrimi teşvik etmede önemli bir rol oynar. Yeni bir buluşun yaşam döngüsü, eski zamanlarda genellikle yüzyıllarca sürmüştür. Fenerin değiştirilmesi için elektrikli ampulü icat etmek binlerce yıl almış ve hem endüstri hem de toplum bu yavaş tempoyu kabul etmiştir. Ancak modern icatların ortalama yaşam döngüsü çok daha kısadır. Dijital kameranın icadının film kamerası ve analog kameranın yerini alması yalnızca birkaç on yıl sürmüştür ve bu, fotoğraf endüstrisinde hızlı bir evrime yol açmıştır. Tersine mühendislik, endüstriyel devrim ve yeniden icat etme sürecini hızlandırma açısından çok önemli bir hale gelmiştir.

1.2 Yazılımlarda Tersine Mühendislik

Yazılımlarda tersine mühendislik, bir sistemi veya bir sistemin parçalarını alma ve bunların işlevselliğini ve tasarımını analiz etme sürecidir. Bir sistemin parçalarını daha yüksek bir soyutlama düzeyinde analiz etmenin ve yeniden yapılandırmanın bir yolu olarak tersine mühendisliğe başvurulur.

Ancak, tersine mühendislik bundan daha fazlası için kullanılır. Yeni bir tür kötü amaçlı yazılımın ortaya çıktığını var sayarsak, kaynak kodu bilinmeyen yeni tehditlere karşı önlemlerin alınabilmesi için kötü amaçlı yazılımın içeriğini analiz etmek, nasıl çalıştığını anlamak ve aynı zamanda fikri mülkiyeti kötü niyetli kişilere karşı korumak için tersine mühendislik tekniklerine ve becerilerine ihtiyaç duyulur. Bu işlem, kötü amaçlı yazılım dosyasını bir hata ayıklayıcıyla açıp içyapısını kontrol etmekten daha karmaşık işlemlere (VM anlık görüntü karşılaştırmaları, ağ izleme vb.) kadar değişebilir. Bir tersine mühendis, hata ayıklayıcı (Debugger) ve ayrıştırıcı (Disassembler) yardımıyla (x64dbg, OllyDbg, IDA Pro, Ghidra, vb.), yazılımın kaynak koduna erişmeden işlevselliğini ve alt programlarını denetleyebilir. Tersine mühendis, Assembly kodunu anlayabilmeli, işletim sisteminin dahili çağrıları ve API'leri konusunda bilgili olmalı ve ayrıca farklı programlama dillerinin ayrıştırılmasıyla (Disassemble) ortaya çıkan yapılar hakkında bilgi sahibi olmalıdır.

1.3 Tersine Mühendislik Süreçleri

Diğer tüm faaliyetler gibi tersine mühendislik de bir süreçtir. Hem analistlere hem de paydaşlara yardımcı olabilecek bilgiler üretilmesine yardımcı olması için izlenmesi gereken bir süreç vardır.

1.3.1 Onay alınması

Etik, yazılımın tersine mühendisliğini yapan herkesin yazılımın sahibinden onay almasını gerektirir. Bazı firmalar, yazılımlarının onay alınmadan tersine mühendislik ile incelenmesi konusunda daha hoşgörülüdür ve yazılımda bulunan herhangi bir güvenlik açığının doğrudan sahibine bildirilmesi ve kamuya açıklanmaması gerekir. Güvenlik açığının topluluğa ne zaman bildirileceğine karar vermek sahibine bağlıdır. Bu durum, bir yazılım yaması yayınlanmadan önce saldırganların bir güvenlik açığı kullanmasını önler.

Kötü amaçlı yazılım veya bilgisayar korsanlığı söz konusu olduğunda, kötü amaçlı yazılımı tersine çevirmek için kötü amaçlı yazılımın yaratıcısından onay alınması gerekmez. Aksine, kötü amaçlı yazılım analizinin hedeflerinden biri bilgisayar korsanını yakalamaktır[3].

1.3.2 Statik analiz

Statik analiz, analiz edilecek yazılımı çalıştırmadan, kod, varlık ve bağımlılıkları incelemek için gerçekleştirilir. Genellikle daha karmaşık bir yaklaşım olarak düşünülür çünkü yazılımın çalıştırıldığı platform, yazılım tarafından kullanılan çerçeveler ve kullanılan programlama diline ve ortama bağlı olarak yazılımın derlendiği dil hakkında derinlemesine bilgi gerektirir.

1.3.3 Dinamik analiz

Dinamik analiz, kodun canlı olarak yürütülmesini gerektiren bir analiz türüdür. Örneğin, büyük miktarda verinin şifresini çözen veya sıkıştırmasını açan bir kodla karşılaşılırsa ve kodu çözülen verilerin içeriğinin görülmesi isteniyorsa, en hızlı seçenek dinamik analiz yapmak olacaktır.

Statik ve dinamik analiz arasında temel farklar Tablo 1.1.'de gösterilmektedir. Statik analiz, kodda daha derin bir anlayışa ve sistemle bazı gerçek etkileşimlere ihtiyaç duyduğumuz noktaları belirlememize yardımcı olur. Statik analizi dinamik analiz ile takip

ederek dosya tanıtıcıları, rastgele oluşturulmuş sayılar, ağ soketi ve paket verileri ve API işlev sonuçları gibi gerçek verileri görebiliriz.

Tablo 1.1. Statik analiz ve dinamik analizin farkları

Statik Analiz	**Dinamik Analiz**
Yürütmeden önce yapılır	Yürütme sırasında yapılır
Yapısal yönlere odaklanır	İşlevsel yönlere odaklanır
Belirli parametrelere sahip sabit bir yaklaşımdır	Dinamiktir ve büyük ölçüde yürütme sırasında karşılaşılan zorluklara bağlıdır
Paketleme ve kod karıştırma yöntemlerine karşı savunmasızdır	Paketleme ve kod karıştırma yöntemlerine karşı güçlüdür

1.1.1 Raporlama

Analiz yapılırken her türlü bilgi toplanmalı ve belgelenmelidir. Gelecekteki analizlere yardımcı olması için tersine mühendislik yapılmış bir nesneyi belgelemek yaygın bir uygulamadır. Yapılan her analiz, gelecekte tasarlayacakları programlarını kusurlardan korumak isteyen geliştiriciler için bir bilgi bankası niteliğindedir.

1.2 Tersine Mühendislik Araçları

1.2.1 Hex düzenleyiciler (hex editors)

Hex düzenleyici, her tür dosyayı açabilen ve içeriğini bayt olarak görüntüleyebilen özel bir düzenleyici türüdür. Çoğu zaman bir dosyayı açtığınızda, programın o dosyanın içeriğini yorumlamasını görürsünüz. Düz metin dosyaları bile dosyanın başlangıcını, satırların nerede kesilmesi gerektiğini, dosyanın sonunu ve daha fazlasını belirten görünmez karakterler içerir. Hex düzenleyiciler ile görünmez karakterleri ve normal karakterleri onaltılık değerler olarak görüntüleyebilir ve üzerlerinde işlem yapılabilir.

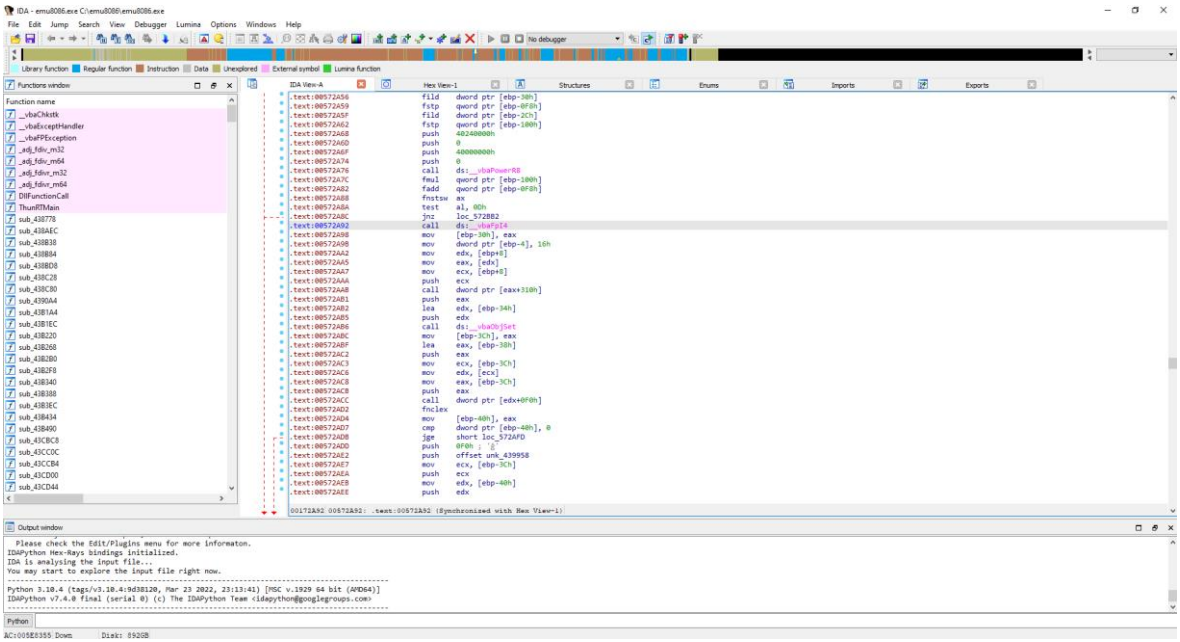
1.2.2 İkili biçim - analiz araçları (binary format - analysis tools)

İkili analiz araçları, ikili dosyaları ayrıştırmak ve dosya hakkında bilgi çıkarmak için kullanılır. Bir analist, hangi uygulamaların ikili dosyayı okuyabildiğini veya yürütebildiğini belirleyebilir. Dosya türleri genellikle sihirli başlık baytlarından tanımlanır. Bu Sihirli Başlık

baytları genellikle bir dosyanın başında bulunur. Örneğin, bir Microsoft yürütülebilir dosyası, MZ başlığıyla başlar.

1.2.3 Ayrıştırıcılar (dissassemblers)

Ayrıştırıcılar, bir programın düşük seviyeli kodunu görüntülemek için kullanılır. Düşük seviyeli kod okumak, assembly dili bilgisi gerektirir. Bir ayrıştırıcı ile yapılan analiz, bir programın yürütüldüğünde gerçekleştireceği yürütme koşulları ve sistem etkileşimleri hakkında bilgi verir. Ayrıştırma, işlemciye özel bir işlemdir ve bazı ayrıştırıcılar birden fazla CPU mimarisini destekler. Şekil 1.1.'de Hex-Rays firmasının yayınladığı IDA Pro isimli ayrıştırıcının bir programı ayrıştırma örneği gösterilmiştir.



Şekil 1.1. IDA Pro ile ayrıştırma işlemi

1.2.4 Hata ayıklayıcılar (debuggers)

Hata ayıklayıcı, genellikle kullanıcının başka bir uygulamanın yürütme durumunu ve verilerini çalışırken görüntülemesine izin vermek için kullanılan bir araçtır. Hata ayıklayıcılar, kullanıcıların programın yürütülmesini durdurmasına, değişkenlerin değerlerini incelemesine, programın satır satır yürütülmesine ve isabet edildiğinde programın o noktada yürütülmesini durduracak satırlara veya belirli işlevlere kesme noktaları (breakpoints) ayarlamasına olanak tanır. Hata ayıklayıcılar tersine mühendisler için önemli bir araçtır çünkü ayrıştırıcılardan farklı olarak programın durumunun çalışma zamanı denetimine izin verirler.

1.2.5 Geri derleyiciler (decompilers)

Geri derleyiciler, ayrıştırıcılara benzer. Bir programın düşük seviyeli (assembly dili) kaynak kodunu görüntülemeye çalışan ayrıştırıcıların aksine, programın üst seviye kaynak kodunu geri getirmeye çalışan araçlardır. Çoğu üst düzey dilde, derleme işlemi sırasında atlanan ve kurtarılması imkânsız olan önemli öğeler vardır. Geri derleyiciler, bazı durumlarda, bir program ikili dosyasından yüksek oranda okunabilir bir kaynak kodunu yeniden oluşturabilen güçlü araçlardır.

1.2.6 Import reconstruction

Paketlenmemiş bir Windows PE yürütülebilir dosyasında üstbilgi, yürütülebilir dosyanın bağlı olduğu diğer kitaplıklardan hangi simgelerin işletim sistemine bağlı olduğunu açıklayan meta veriler içerir. İşletim sisteminin yükleyicisi, bu kitaplıkları belleğe yüklemekten ve içe aktarılan simgelerin adreslerini yürütülebilir dosyanın bellek görüntüsündeki yapılara yerleştirmekten sorumludur. Öte yandan, paketleyiciler genellikle bu meta verileri yok eder ve normalde yükleyici tarafından gerçekleştirilmesi gereken çözüm aşamasını kendisi gerçekleştirir. Paketten çıkarmanın amacı, eksik içe aktarma bilgileri de dahil olmak üzere korumaları ikili dosyadan kaldırmaktır. Import Reconstruction, paketleyicinin yürütülebilir dosya için yüklediği içe aktarma koleksiyonunu belirleyen ve işletim sisteminin içe aktarmaları her zamanki gibi düzgün bir şekilde yüklemesini sağlamak için paketlenmemiş yürütülebilir dosyanın görüntüsünde meta verileri yeniden oluşturan araçlardır.

2. X86 PLATFORMU VE ASSEMBLY DİLİ

X86, Intel 8086 mikroişlemci ve onun 8088 varyantına dayalı olarak geliştirilen bir karmaşık komut seti bilgisayarı (CISC) ve komut seti mimarileri ailesidir. 1978'de Intel'in 8-bit 8080 mikroişlemcisinin 16-bitlik uzantısı olarak 8086 mikroişlemcisi tanıtıldı ve bellek segmentasyonu, 16-bit adresin kapsayabileceğinden daha fazla belleği adreslemek için bir çözüm oldu. "X86" terimi, Intel'in 8086 işlemcisinin ardıllarının adlarının "86" ile bitmesinden dolayı ortaya çıktı.

X86-32 platformunun orijinal düzenlemesi 1985 yılında tanıtılan Intel 80386 mikroişlemcisiydi. 80386, 32-bit geniş kayıtları ve veri türleri, düz bellek modeli seçenekleri, 4 GB mantıksal adres alanı ve sanal bellek disk belleği içerecek şekilde 16-bit öncüllerinin mimarisini genişletti.

X86-32 mikro mimarisinin genişletilmesi, 1993 yılında ilk Pentium marka işlemcinin piyasaya sürülmesiyle devam etti. P5 mikro mimarisi olarak bilinen performans geliştirmeleri arasında dual-instruction execution pipeline, 64 bit harici veri yolu ve ayrı yonga üzerinde kod ve veri önbellekleri bulunuyordu. (Bir mikro mimari, kayıt dosyaları, yürütme birimleri, yönerge hatları, veri yolları ve bellek önbellekleri dahil olmak üzere bir işlemcinin dahili bileşenlerinin organizasyonunu tanımlar. Mikro mimariler genellikle birden çok işlemci ürün grubu tarafından kullanılır.) P5 mikro mimarisinin sonraki sürümlerinde, 64 bit genişliğinde kayıtlar (1997) kullanılarak paketlenmiş tamsayılar üzerinde tek komutlu çoklu veri (SIMD) işlemlerini destekleyen MMX teknolojisi adı verilen yeni bir hesaplama kaynağı eklendi.

X86 platformu öncelikle gömülü sistemler ve küçük çok kullanıcılı veya tek kullanıcılı bilgisayarlar için geliştirilmiş olsa da, günümüzde x86 hem sabit hem de taşınabilir kişisel bilgisayarlar dahil hemen her yerde kullanılmakta ve geliştirilmesine devam edilmektedir.

X86, halka düzeyi adı verilen bir soyutlama aracılığıyla ayrıcalık ayrımı kavramını destekler. İşlemci, 0'dan 3'e kadar numaralandırılmış dört halka (ring) seviyesini destekler. 0 halkası en yüksek ayrıcalık seviyesidir ve tüm sistem ayarlarını değiştirebilir. Halka 3, en düşük ayrıcalıklı düzeydir ve yalnızca sistem ayarlarının bir alt kümesini okuyabilir, değiştirebilir.

2.1 Yazmaçlar

İşlemcinin gerçekleştirdiği işlemlerin çoğu veri işleme gerektirir. Ne yazık ki, bir işlemcinin üstlenebileceği en yavaş işlemler, verileri bellekten okumaya veya depolamaya çalışmaktır. İşlemci, bir veri öğesine eriştiğinde istek, kontrol veri yolu üzerinden bellek depolama birimine gitmelidir. Bu işlem karmaşıktır ve aynı zamanda işlemciyi, belleğe erişim yapılırken beklemeye zorlar. Bu aksama süresi, diğer talimatların işlenmesi için harcanabilir.

Bu sorunu çözmeye yardımcı olmak için işlemci, yazmaç adı verilen dahili bellek konumlarını içerir. Yazmaçlar, bellek depolama birimine erişmek zorunda kalmadan veri öğelerini işlemek için depolayabilir. Yazmaçların dezavantajı, işlemci çipinde yerleşik olarak sınırlı sayıda bulunmasıdır[4].

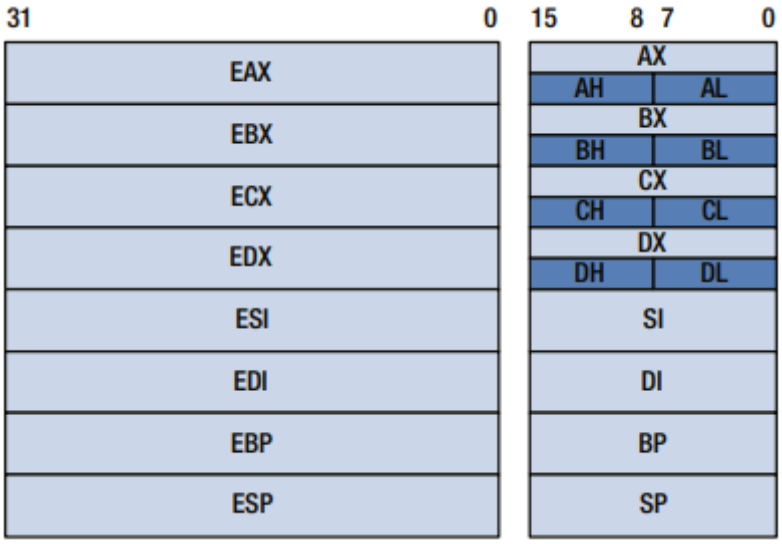
IA-32 platform işlemcileri, farklı boyutlarda çok sayıda yazmaç grubuna sahiptir. IA-32 platformundaki farklı işlemciler, özel kayıtlar içerir. IA-32 ailesindeki tüm işlemciler tarafından kullanılabilen yazmaçlar ve tanımları Tablo 2.1.'de gösterilmektedir.

Tablo 2.1. Yazmaçlar ve tanımları

Yazmaç	**Tanım**
Genel Amaçlı	Çalışma verilerini depolamak için kullanılan her biri 32 bitten oluşan sekiz adet yazmaç vardır.
Segment	Bellek erişimini işlemek için kullanılan her biri 16 bitten oluşan altı adet yazmaç vardır.
Talimat İşaretçisi	Yürütülecek olan bir sonraki talimat koduna işaret eden ve 32 bitten oluşan tek bir yazmaç vardır.
Bayraklar	CPU'nun mevcut durumunu içeren durum kaydıdır ve 32 bit genişliğindedir.(EFLAGS için)
Kontrol	İşlemcinin çalışma modunu belirlemek için kullanılan her biri 32 bitten oluşan beş adet yazmaç vardır.
Hata Ayıklama	İşlemcide hata ayıklama işlemi ile ilgili bilgiyi içeren ve her biri 32 bitten oluşan yazmaç vardır.

2.1.1 Genel amaçlı yazmaçlar

x86-32 çekirdek yürütme birimi, sekiz adet 32 bit genel amaçlı yazmaç içerir. Bu yazmaçlar öncelikle mantıksal, aritmetik ve adres hesaplamaları yapmak için kullanılır. Ayrıca geçici depolama için ve bellekte depolanan veri öğelerine işaretçiler olarak kullanılabilirler. Şekil 2.1.'de bir komut işleneni (instruction operand) olarak bir yazmacı belirtmek için kullanılan adlarla birlikte genel amaçlı yazmaçların tamamı gösterilmektedir. 32-bit işlenenleri desteklemenin yanı sıra, genel amaçlı yazmaçlar ayrıca 8-bit veya 16-bit işlenenleri kullanarak hesaplamalar yapabilir. AX, BX, CX ve DX kayıtları için, en küçük ve en önemli baytlara daha küçük kayıtlar tarafından erişilebilir. AX için, alttaki 8 bit AL kaydı kullanılarak okunabilirken, üstteki 8 bit, burada gösterildiği gibi AH kaydı kullanılarak okunabilir:



Şekil 2.1. Genel amaçlı yazmaçlar

Bazı talimatlar tarafından belirli kayıtların zorunlu veya örtük kullanımı, görünüşte kod yoğunluğunu iyileştirmek için 8086'ya kadar uzanan eski bir tasarım modelidir. Modern bir programlama perspektifinden bunun anlamı, belirli kayıt kullanım kurallarının olmasıdır.

x86-32 derleme kodu yazarken gözlemlenme eğilimindedir. Tablo 2.2.'de genel amaçlı yazmaçlar ve bunların geleneksel kullanımları gösterilmektedir[5].

Tablo 2.2. Genel amaçlı yazmaçlar ve geleneksel kullanımları

Yazmaç	Geleneksel Kullanım
EAX	Akümülatör
EBX	Bellek işaretçisi, genel yazmaç
ECX	Döngü kontrolü, sayaç
EDX	Tamsayı çarpma, tamsayı bölme
ESI	Dizi talimatı kaynak işaretçisi, dizin kaydı
EDI	Dizi talimatı hedef işaretçisi, dizin kaydı
ESP	Yığın işaretçisi
EBP	Yığın çerçevesi taban işaretçisi

Tablo 2.2.'de gösterilen kullanım kuralları yaygın uygulamalardır ancak zorunlu değildir. x86-32 komut seti, örneğin, ECX yazmacının bir sayaç olarak geleneksel kullanımına rağmen, yürütülen bir görevin bellek işaretçisi olarak kullanmasını engellemez. Ayrıca, x86 derleyicileri bu kullanım kurallarını zorlamaz. x86-32 modunda mevcut olan sınırlı sayıda genel amaçlı yazmaçlar göz önüne alındığında, genellikle geleneksel olmayan bir şekilde genel amaçlı bir yazmaç kullanmak gerekir. Son olarak, Tablo 2.2.'de özetlenen kullanım kurallarının, C++ gibi yüksek seviyeli bir dil tarafından tanımlanan bir çağrı kuralı ile aynı olmadığına dikkat edilmelidir.

2.1.2 Segment yazmaçları

x86-32 çekirdek yürütme birimi, program yürütme ve veri depolama için mantıksal bir bellek modeli tanımlamak üzere segment yazmaçlarını kullanır. Bir x86 işlemci, kod, veri ve yığın alanı için bellek bloklarını belirleyen Tablo 2.3.'te görünen altı segment yazmacını içerir. Her segment yazmacı 16 bittir ve belleğe özgü segmentin başlangıcına yönelik işaretçiyi içerir.

CS yazmacı, bellekteki kod segmentinin işaretçisini içerir. Kod bölümü, komut kodlarının bellekte saklandığı yerdir. İşlemci, CS yazmaç değerine ve EIP talimat işaretçi kaydında bulunan bir ofset değerine dayalı olarak talimat kodlarını bellekten alır. Bir program, CS kaydını açıkça yükleyemez veya değiştiremez.

SS segment yazmacı, yığın segmentine işaret etmek için kullanılır. Yığın, program içindeki işlevlere ve prosedürlere iletilen veri değerlerini içerir.

DS, ES, FS ve GS segment yazmaçları, veri segmentlerine işaret etmek için kullanılır. Program, dört ayrı veri segmentine sahip olarak, veri öğelerinin çakışmamasını sağlayarak ayırmaya yardımcı olabilir. Program, segmentler için uygun işaretçi değeriyle veri segmenti kayıtlarını yüklemeli ve bir ofset değeri kullanarak bireysel bellek konumlarına başvurmalıdır.

Tablo 2.3. Segment yazmaçları ve tanımları

Segment Yazmacı	Tanım
CS	Code Segment
DS	Data Segment
SS	Stack Segment
ES	Extra Segment Pointer
FS	Extra Segment Pointer
GS	Extra Segment Pointer

2.1.3 Talimat işaretçisi yazmacı

Program sayacı olarak da adlandırılan talimat işaretçi yazmacı (EIP), yürütülecek bir sonraki talimatın adres kaydını tutar. Bir uygulama programı, talimat işaretçisini kendi başına doğrudan değiştiremez. Ayrıca bir bellek adresi belirleyip EIP kaydına yerleştirme yapılamaz. Bunun yerine, okunacak bir sonraki talimatı değiştirmek için atlamalar gibi normal program kontrol komutları kullanmak gerekir.

2.1.4 Bayrak yazmaçları

İşlemcide gerçekleştirilen her işlem için, işlemin başarılı olup olmadığını belirleyen bir mekanizma olmalıdır. Bu işlevi gerçekleştirmek için işlemci bayrakları kullanılır.

Bayraklar, bir programın işlevinin başarılı olup olmadığını belirlemenin tek yolu olduklarından, assembly dili programları için önemlidir. Örneğin, bir uygulama negatif bir değerle sonuçlanan bir çıkarma işlemi yaparsa, işlemci içinde özel bir bayrak ayarlanır. Bayrağı kontrol etmeden, assembly dili programının bir şeylerin yanlış gittiğini bilmesinin hiçbir yolu olmaz.

IA-32 platformu (x86-32 mimarisi), EFLAGS isimli, belirli bilgi bayraklarını temsil etmek üzere eşlenen 32 bit bilgileri içeren yazmacı kullanır. Bazı bitler, gelecekteki işlemcilerde yeni bayrakların tanımlanmasına izin vermek ve kullanılmak üzere ayrılmıştır.

Bayraklar, işlevine göre durum, kontrol ve sistem bayrakları olarak üç gruba ayrılır.

Durum bayrakları, işlemci tarafından matematiksel bir işlemin sonuçlarını belirtmek için kullanılır. Mevcut durum bayrakları Tablo 2.4.'te gösterilmektedir.

Tablo 2.4. Durum Bayrakları

Bayrak Kısaltması	**Bit**	**İsim**
CF	0	Carry Flag
PF	2	Parity Flag
AF	4	Auxiliary Carry Flag
ZF	6	Zero Flag
SF	7	Sign Flag
OF	11	Overflow Flag

Carry Flag, işaretsiz bir tamsayı değerindeki matematiksel bir işlemde en önemli bit için bir taşıma veya ödünç alma oluşturursa ayarlanır. Bu durum, matematiksel işlemde yer alan kayıt için bir taşma koşulunu temsil eder.

Parity Flag, matematiksel bir işlemdeki sonuç kaydının bozuk veri içerip içermediğini belirtmek için kullanılır. İşlem sonucunda değeri 1 olan bitlerin sayısı tek ise 0, çift ise 1 değerine ayarlanır.

Auxiliary Carry Flag, sonucun 3 ve 4 numaralı bit konumları arasında toplamadan sonra taşımayı (yarım taşıma) veya çıkarmadan sonra ödünç almayı tutar. Bu son derece özel bayrak biti, bir BCD toplama veya çıkarma işleminden sonra AL'nin değerini ayarlamak için DAA ve DAS talimatları tarafından test edilir.

Zero Flag, aritmetik veya mantıksal bir işlemin sonucunun sıfır olduğunu gösterir. Sonuç sıfır ise 1 değerine, sonuç sıfırdan farklı ise 0 değerine ayarlanır.

Sign Flag, aritmetik veya mantık talimatı yürütüldükten sonra sonucun aritmetik işaretini tutar. S=1 ise, sonuç negatiftir; S=0 ise, sonuç pozitiftir.

Overflow Flag işaretli bir tamsayı değerindeki matematiksel bir işlemde taşma meydana gelirse ayarlanır. İşaretsiz işlemler için OF yok sayılır.

Kontrol bayrakları, işlemcideki belirli bir davranışı kontrol etmek için kullanılır. Şu anda yalnızca bir kontrol bayrağı tanımlanmıştır, DF bayrağı (Direction Flag) dizelerin işlemci tarafından işlenme şeklini kontrol etmek için kullanılır.

DF bayrağı ayarlandığında (bire ayarlandığında), dize talimatları, dizedeki bir sonraki baytı almak için bellek adreslerini otomatik olarak azaltır. DF bayrağı temizlendiğinde (sıfıra ayarlandığında), dize komutları dizedeki bir sonraki baytı almak için bellek adreslerini otomatik olarak artırır.

Sistem bayrakları, işletim sistemi düzeyindeki işlemleri denetlemek için kullanılır. Uygulama programları asla sistem bayraklarını değiştirmeye çalışmamalıdır. Sistem bayrakları Tablo 2.5.'te listelenmiştir.

Tablo 2.5. Sistem bayrakları

Bayrak Kısaltması	**Bit**	**İsim**
TF	8	Trap Flag
IF	9	Interrupt Enable Flag
IOPL	12 ve 13	I/O Privilege Level Flag
NT	14	Nested Task Flag
RF	16	Resume Flag
VM	17	Virtual 8086 Mode Flag
AC	18	Alignment Check Flag
VIF	19	Virtual Interrupt Flag
VIP	20	Virtual Interrupt Pending Flag
ID	21	Identification Flag

Trap Flag, tek adımlı modu etkinleştirmek için ayarlanmıştır. Tek adım modunda, işlemci bir seferde yalnızca bir talimat kodu gerçekleştirir ve bir sonraki talimatı gerçekleştirmek için bir sinyal bekler. Bu özellik, assembly dili uygulamalarında hata ayıklarken son derece kullanışlıdır.

Interrupt Enable Flag, işlemcinin harici kaynaklardan alınan sinyallere nasıl yanıt vereceğini kontrol eder.

I/O Privilege Level Flag, çalışmakta olan görevin Giriş-Çıkış (Input/Output - I/O) ayrıcalık düzeyini gösterir. Bu, I/O adres alanı için erişim düzeylerini tanımlar. Görev veya programın ayrıcalık alanı değeri, I/O adres alanına erişmek için IOPL'den küçük veya ona eşit olmalıdır; aksi takdirde adres alanına erişim talebi reddedilecektir.

Nested Task Flag, çalışmakta olan görevin önceden yürütülen göreve bağlı olup olmadığını denetler. Bu, kesintiye uğramış ve çağrılan görevleri zincirlemek için kullanılır.

Resume Flag, işlemcinin hata ayıklama modundayken özel durumlara nasıl yanıt vereceğini denetler.

Virtual 8086 Mode Flag, işlemcinin korumalı veya gerçek mod yerine sanal 8086 modunda çalıştığını gösterir.

Alignment Check Flag, bellek başvurularının hizalama denetimini etkinleştirmek için kullanılır.

Virtual Interrupt Flag, işlemci sanal modda çalışırken Interrupt Enable bayrağı gibi davranır.

Virtual Interrupt Pending Flag, İşlemci sanal modda çalışırken, bir kesmenin beklemede olduğunu belirtmek için kullanılır.

Identification Flag işlemcinin CPUID yönergesini destekleyip desteklemediğini göstermesi bakımından önemlidir. İşlemci bu bayrağı ayarlayabiliyorsa veya temizleyebiliyorsa, CPUID yönergesini destekler. Değilse, CPUID yönergesi kullanılamaz.

2.1.5 Kontrol yazmaçları

İşlemcinin çalışma modunu ve o anda yürütülen görevin özelliklerini belirlemek için kullanılır. Kontrol yazmaçlarındaki değerlere doğrudan erişilemez, ancak kontrol yazmacında bulunan veriler genel amaçlı yazmaçlara taşınabilir. Kontrol yazmaçları ve tanımları Tablo 2.6.'da gösterilmektedir.

Tablo 2.6. Kontrol Yazmaçları

Kontrol Yazmacı	**Tanım**
CR0	İşlemcinin çalışma modunu ve durumlarını kontrol eden sistem bayrakları
CR1	Ayrılmıştır, şu anda kullanılmamaktadır.
CR2	Bellek sayfası arıza bilgisi için kullanılır
CR3	Bellek sayfası dizin bilgileri için kullanılır
CR4	Giriş-çıkış kesme noktalarının etkinleştirilmesi, sayfa boyutu uzantısı ve makine denetimi özel durumları gibi işlemleri kontrol etmek için korumalı modda kullanılır.
CR5-CR7	Ayrılmıştır, şu anda kullanılmamaktadır.

2.2 Veri Türleri

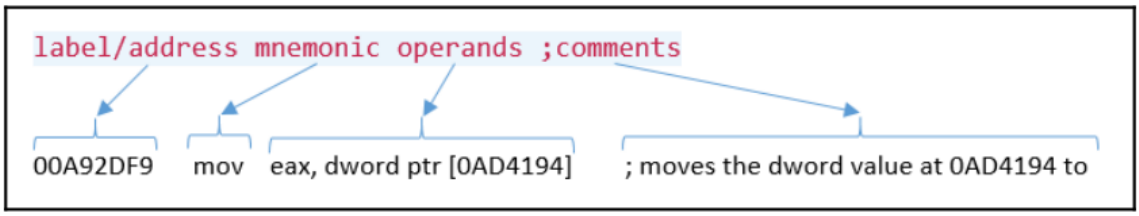
Temel veri türü, program yürütme sırasında işlemci tarafından manipüle edilen temel bir veri birimidir. x86 platformu, uzunlukları 8 bit (1 bayt) ile 256 bit (32 bayt) arasında değişen kapsamlı bir temel veri türleri grubunu destekler. Tablo 2.7.'de veri türleri ve tipik kullanımları gösterilmektedir.

Tablo 2.7. Veri Tipleri

Veri Türü	**Boyut (Bit)**	**Kullanım Alanları**
Byte	8	Karakter, tamsayılar, İkili Kodlu Ondalık (BCD) değerleri
Word	16	Karakter, tamsayılar
Double (Long) Word	32	Tamsayılar, tek duyarlıklı kayan nokta
Quad Word	64	Tamsayılar, çift duyarlıklı kayan nokta, paketlenmiş tamsayılar
Quint Word	80	Çift genişletilmiş hassas kayan nokta, paketlenmiş BCD
Double Quad Word	128	Paketlenmiş tamsayılar, paketlenmiş kayan nokta
Quad Quad Word	256	Paketlenmiş tamsayılar, paketlenmiş kayan nokta

2.3 Komut Setine Genel Bakış

Assembly dili, Şekil 2.2.'deki sözdizimini izleyen doğrudan kod satırlarından oluşur.



Şekil 2.2. Assembly dili sözdizimi

Label/adress, komut satırının adresini tanımlamak için kullanılmaktadır. Mnemonic, MOV, ADD ve SUB gibi insan tarafından okunabilen talimatlardır. Her mnemonic, bir bayt numarası veya işlem kodu (opcode) adı verilen birkaç bayt ile temsil edilir. Operandlar komutun argümanlarıdır. Comment ise programa açıklama satırı eklenebildiğini göstermek için kullanılmaktadır.

2.3.1 Talimat işlenenleri (instruction operands)

Çoğu x86-32 talimatı, bir talimatın etki edeceği değerleri belirleyen işlenenleri yani operandları kullanır. Çoğu talimat, programcının kaynak (source) ve hedef (destination) operandları açıkça belirtmesini gerektirir. Immediate, Register ve Memory olmak üzere üç temel operand türü vardır. . Tablo 2.8'de çeşitli operand türlerini kullanan birkaç talimat örneği görülmektedir.

Tablo 2.8. Operand türleri, örnekler ve karşılıkları

Operand	Örnek	C/C++ Karşılığı
Immediate	Talimat hedef, kaynak mov eax,30 imul ebx,10h xor dl,60h	 eax=30 ebx*=0x10 dl^=0x60

Register	mov eax,ebx
	inc ecx
	add eax,edx
		eax=ebx
		ecx+=1
		eax+=edx

Memory	mov eax,[ebx] sub word ptr [edi],10	eax=*ebx *(short*)edi-=10

Immediate Operandlar tipik olarak aritmetik, mantıksal veya ofset değerlerini belirtmek için kullanılan sabit değerlerdir. Immediate Operand olarak yalnızca kaynak operandları kullanılabilir.

Register Operandları genel amaçlı yazmaçlarda bulunur.

Memory Operandı bellekteki bir konumu belirtir. Bir komut, kaynak veya hedef operandı Memory Operand olarak belirtebilir, ancak ikisinde birden Memory Operand belirtemez.

2.3.2 En Yaygın kullanılan X86 Talimatları

En yaygın kullanılan X86 talimatları Tablo 2.9.'da gösterilmektedir.

Tablo 2.9. En yaygın kullanılan X86 talimatları

Talimat	Açıklama	Örnek
mov	İkinci operanddaki veriyi ilk operanda aktarır.	mov eax,02 mov eax, ebx
push	Yığına bir Genel amaçlı yazmaç(GPR), bellek konumu veya anlık değer gönderir. Bu komut, ESP'den dört çıkarır ve belirtilen operandı ESP'nin gösterdiği bellek konumuna kopyalar.	push ebp
pop	Yığından en üstteki öğeyi çağırır. Bu talimat, ESP tarafından işaret edilen hafıza konumunun içeriğini belirtilen GPR veya hafıza konumuna kopyalar; daha sonra ESP'ye dört ekler.	pop ebp

xchg	İki GPR veya bir GPR ve bir bellek konumu arasındaki verileri değiş tokuş eder. Komutun kayıt-bellek formu kullanılıyorsa, işlemci kilitli bir veri yolu döngüsü kullanır.	xchg eax,ebx
add	Kaynak operandı ve hedef operandını toplar ve hedef operandına kaydeder. Bu komut hem işaretli hem de işaretsiz tamsayılar için kullanılabilir.	add ecx,1
sub	Kaynak operandı hedef operandından çıkarır. Bu komut hem işaretli hem de işaretsiz tamsayılar için kullanılabilir.	sub esp,4
imul	İki operand arasında işaretli bir çarpma işlemi gerçekleştirir. Bu komut, tek bir kaynak operandı (örtük operand olarak AL, AX veya EAX ile), açık bir kaynak ve hedef operandı ve üç operand türevi (anlık kaynak, bellek/kayıt kaynağı ve GPR hedefi) dahil olmak üzere birden çok formu destekler.	imul ecx,esi
idiv	Bölünen olarak AX, DX:AX veya EDX:EAX ve bölen olarak kaynak operandı kullanarak işaretli bir bölme işlemi gerçekleştirir. Elde edilen bölüm ve kalan, AL:AH, AX:DX veya EAX:EDX kayıt çiftine kaydedilir.	idiv ebx
inc	Belirtilen operandı bir arttırır.	inc bh
dec	Belirtilen operanddan bir eksiltir.	dec eax
neg	Belirtilen operanda ikinin tümleyeni değerini kaydeder.	neg al

cmp	Kaynak operandı hedeften çıkararak iki operandı karşılaştırır ve ardından durum bayraklarını ayarlar. Çıkarmanın sonuçları atılır.	cmp eax,eax
and	Kaynak ve hedef operandların bit düzeyinde AND değerini hesaplar.	and eax,0ffffh
or	Kaynak ve hedef operandların bit düzeyinde OR değerini hesaplar.	or ebx,ecx
xor	Kaynak ve hedef operandların bit düzeyinde XOR değerini hesaplar.	xor al,al
not	Belirtilen operand bire tümleyenini hesaplar. Bu komut durum bayraklarını etkilemez.	not eax
test	Kaynak ve hedef operandının bit düzeyinde AND değerini hesaplar ve sonuçları atar. Bu komut, durum bayraklarını ayarlamak için kullanılır.	test eax,eax
jmp	Operand tarafından belirtilen bellek konumuna koşulsuz bir atlama gerçekleştirir.	jmp label1
lea	Kaynak operandın etkin adresini hesaplar ve genel amaçlı bir yazmaç olması gereken hedef operandına kaydeder.	lea eax, [ecx+10]
nop	Talimat işaretçisini (EIP) bir sonraki talimata ilerletir. Başka hiçbir kayıt veya bayrak değiştirilmez.	nop
rol	Belirtilen operandı sola döndürür.	rol ebx,3
ror	Belirtilen operandı sağa döndürür.	ror ax, 2

fst*	ST(0)'ı ST(i)'ye veya bir hafıza konumuna kopyalar.	
fld*	Kayıt yığınına bir kayan nokta değeri gönderir. Kaynak operandı, ST(i) içeriği veya bir bellek konumu olabilir.	fld qword ptr ss:[ebp-20]
shl	Belirtilen operandın bitlerinin sola kaydırılmasını gerçekleştirir.	shl dl,2
sar	Belirtilen operandın bitlerinin sağa kaydırılmasını gerçekleştirir.	sar edx,3
jz	jz komutu, koşullu bir atlamadır. Zero Flag (ZF) 1 ise belirtilen konuma atlar.	jz function.004013E3
jnz	jnz komutu, koşullu bir atlamadır. Zero Flag (ZF) 0 ise belirtilen konuma atlar.	jnz label1

2.4 Yürütme Akışını Kontrol Etme

İşlemci, bilgisayar programını çalıştırdığında, ilk komut ile başlaması ve programdaki tüm komutlar boyunca son komuta kadar sırayla ilerlemesi pek olası değildir. Bilgisayar programları büyük olasılıkla ihtiyaç duyduğu işlevleri uygulamak ve gerekli mantığı gerçekleştirmek için dalları ve döngüleri kullanacaktır.

Üst düzey dillere benzer şekilde, assembly dili, programcının kod mantığının uygulamalara girmesine yardımcı olacak talimatlar sağlar. Programın farklı bölümlerine atlayarak veya bölümler arasında birden çok kez dolaşarak, programın verileri işleme şeklini değiştirebilir.

2.4.1 Koşulsuz dallanma

Programda koşulsuz bir dallanma ile karşılaşıldığında, talimat işaretçisi (EIP) otomatik olarak farklı bir konuma yönlendirilir. Üç tür koşulsuz dallanma vardır:

- Jumps (Atlamalar)
- Calls (Çağrılar)

- Interruptions (Kesmeler)

Bu koşulsuz dallanmaların her biri program içinde farklı davranır.

Atlamalar, assembly dili programlamasında kullanılan en temel dallanma türüdür. Jump deyimleri, GOTO deyiminin derleme dilindeki karşılığıdır.

Yapılandırılmış programlamada, GOTO ifadeleri hatalı kodlamanın bir işareti olarak kabul edilir. Programların bölümlere ayrılması ve sıralı bir şekilde akması, atlama yerine işlevleri çağırması gerekir; ancak assembly dili ile yazılmış programlarda, atlama komutları kötü programlama olarak kabul edilmez ve aslında birçok işlevi uygulamak için gereklidir.
3 farklı koşulsuz atlama türü vardır:

- Short jump
- Far jump
- Near jump

Üç atlama türü, mevcut talimatın bellek konumu ile hedef noktasının bellek konumu (atlama konumu) arasındaki mesafeye göre belirlenir. Atlanan bayt sayısına bağlı olarak, farklı atlama türleri kullanılır. Short Jump, atlama ofseti 128 bayttan az olduğunda kullanılır. Far Jump, bölümlenmiş bellek modellerinde, atlama başka bir bölümdeki bir komuta gittiğinde kullanılır. Near Jump, diğer tüm atlamalar için kullanılır.

Bir sonraki koşulsuz dal türü çağrıdır. Bir çağrı, atlama talimatına benzer, ancak nereden atladığını hatırlar ve gerekirse oraya geri dönme yeteneğine sahiptir. Bu, assembly dili programlarında işlevleri uygularken kullanılır.

İşlevler, bölümlere ayrılmış kod yazmanıza olanak tanır; yani, farklı işlevler metnin farklı bölümlerine ayrılabilir. Programın birden fazla alanı aynı işlevleri kullanıyorsa, aynı kodun birden çok kez yazılmasına gerek yoktur. Tek işleve, çağrı ifadeleri kullanılarak başvurulabilir.

CALL komutu yürütüldüğünde, EIP kaydını yığına yerleştirir ve ardından EIP kaydını, çağrılan işlev adresini gösterecek şekilde değiştirir. Çağrılan işlev tamamlandığında, yığından eski EIP kayıt değerini alır ve denetimi orijinal programa geri döndürür.

Kesme, işlemcinin mevcut talimat kodu yolunu "kesmesi" ve farklı bir yola geçmesi için bir yoldur. Kesmeler iki çeşittir:

- Yazılım kesmeleri (Software Interrupts)
- Donanım kesmeleri (Hardware Interrupts)

Donanım aygıtları, donanım kesmeleri oluşturur. Donanım düzeyinde meydana gelen olayları bildirmek için kullanılırlar (örneğin, bir Giriş/Çıkış bağlantı noktası gelen bir sinyal

aldığında). Programlar, yazılım kesmeleri oluşturur. Kontrolü başka bir programa devretmek için bir sinyaldir.

Bir program bir kesme ile çağrıldığında, çağıran program beklemeye alınır. Talimat işaretçisi, çağrılan programa aktarılır ve yürütme, çağrılan programın içinden devam eder. Çağrılan program tamamlandığında, kontrolü çağıran programa geri verebilir (bir kesme dönüş talimatı kullanarak).

Yazılım kesmeleri, uygulamaların işletim sistemi içindeki ve hatta bazı durumlarda temel BIOS sistemi içindeki işlevlerden yararlanmasını sağlamak için işletim sistemi tarafından sağlanır. Microsoft DOS işletim sisteminde birçok işlev 0x21 yazılım kesmesi ile sağlanmaktadır. Linux dünyasında, 0x80 kesmesi, düşük seviyeli çekirdek işlevleri sağlamak için kullanılır.

2.4.2 Koşullu dallanma

Koşulsuz dallanmaların aksine, koşullu dallanmalar her zaman işleme alınmaz. Koşullu dallanmanın sonucu, dalın yürütüldüğü andaki EFLAGS kaydının durumuna bağlıdır.

EFLAGS kaydında birçok bit vardır, ancak koşullu dallanmalar bunlardan yalnızca beşi ile ilgilidir:

- Carry Flag (CF) - bit 0
- Parity Flag (PF) - bit 2
- Zero Flag (ZF) - bit 6
- Sign Flag (SF) - bit 7
- Overflow Flag (OF) - bit 11

Her koşullu atlama komutu, koşulun, atlamanın gerçekleşmesi için uygun olup olmadığını belirlemek için belirli bayrak bitlerini inceler. Beş farklı bayrak biti ile birkaç atlama kombinasyonu gerçekleştirilebilir.

Koşullu atlamalar, EFLAGS kaydının mevcut değerine göre atlama yapılıp yapılmayacağını belirler. Tablo 2.10.'da, mevcut tüm koşullu atlama talimatları ve bu talimatların bağlı olduğu bayrak bitleri gösterilmektedir.

Tablo 2.10. Koşullu atlamalar ve bayrak durumları

Talimat	**Bayrakların Durumları**	**Açıklama**
JZ/JE	ZF = 1	Sıfırsa / Eşitse
JNZ/JNE	ZF = 0	Sıfır Değilse / Eşit Değilse
JS	SF = 1	Sign Flag Set Edildiyse
JNS	SF = 0	Sign Flag Set Edilmediyse
JC/JB/JNAE	CF = 1	Carry Flag Set Edildiyse / Küçükse / Büyük ve Eşit Değilse
JNC/JNB/JAE	CF = 0	Carry Flag Set Edilmediyse / Küçük Değilse / Büyük veya Eşitse
JO	OF = 1	Overflow Flag Set Edildiyse
JNO	OF = 0	Overflow Flag Set Edilmediyse
JA/JNBE	CF = 0 ve ZF = 0	Büyükse / Küçük ve Eşit Değilse
JNA/JBE	CF = 1 veya ZF = 1	Büyük Değilse / Küçük veya Eşitse
JG/JNLE	ZF = 0 ve SF = OF	Büyükse / Küçük ve Eşit Değilse
JNG/JLE	ZF = 1 veya SF != OF	Büyük Değilse / Küçük veya Eşitse
JL/JNGE	SF != OF	Küçükse / Büyük ve Eşit Değilse
JNL/JGE	SF = OF	Küçük Değilse / Büyük veya Eşitse
JP/JPE	PF = 1	Parity Flag Set Edildiyse
JNP/JPO	PF = 0	Parity Flag Set Edilmediyse
JCXZ	CX = 0	CX = 0 ise
JECXZ	ECX = 0	ECX = 0 ise

Karşılaştırma talimatı, koşullu bir atlama için iki değeri değerlendirmenin en yaygın yoludur. Karşılaştırma talimatı, adından da anlaşılacağı gibi, iki değeri karşılaştırır ve EFLAGS kayıtlarını buna göre ayarlar. CMP talimatının formatı aşağıdaki gibidir:

cmp operand1, operand2

CMP talimatı, ikinci operandı birinci operand ile karşılaştırır. Arka planda iki operand üzerinde bir çıkarma işlemi gerçekleştirilir (operand2 – operand1). Operandların hiçbiri değiştirilmez, ancak EFLAGS kaydı, çıkarma gerçekleşmiş gibi ayarlanır.

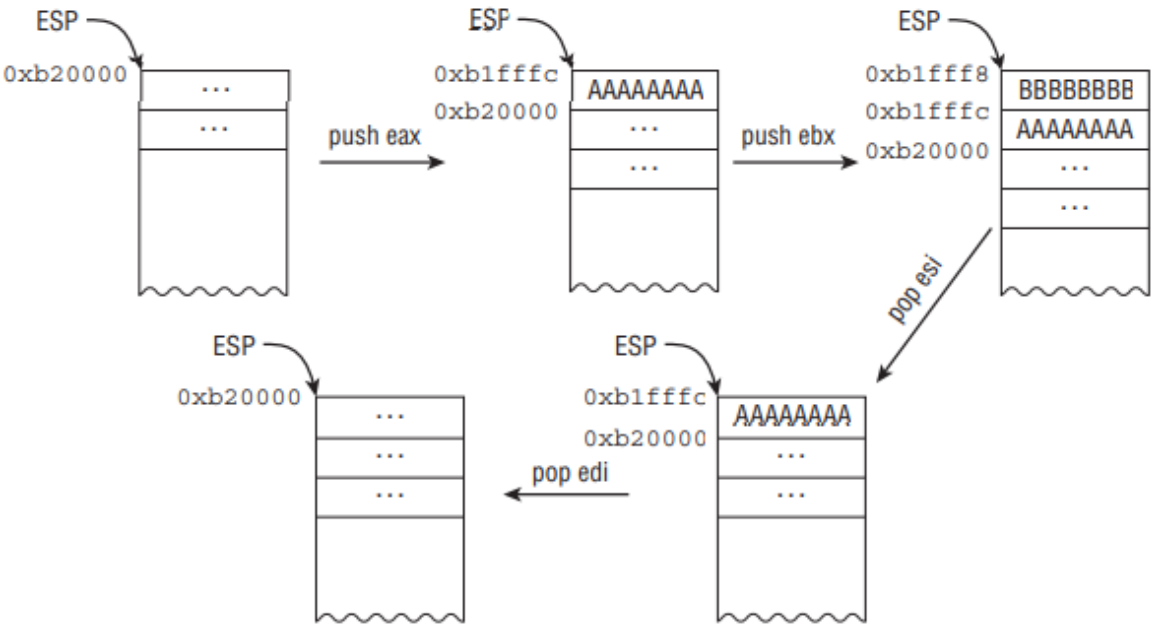
2.5 Yığın (Stack) İşlemleri

Programlama dillerinde ve işletim sistemlerinde temel bir veri yapısı olan yığın, verilerin geçici olarak depolandığı bir bellek alanıdır. Örneğin, C'deki yerel değişkenler, işlevlerin yığın alanında depolanır. İşletim sistemi ring 3'ten ring 0'a geçtiğinde, durum bilgilerini yığına kaydeder. Yığına veri ekleme ve çıkarma, son giren ilk çıkar (Last in First Out / LIFO) yöntemi şeklindedir ve PUSH/POP komutları aracılığıyladır. Yığının tepesinin adresi ESP kaydında saklanır. PUSH komutu ESP'yi azaltır ve ardından verileri ESP'nin gösterdiği konuma yazar; POP, verileri okur ve ESP'yi artırır. Varsayılan otomatik artırma/azaltma değeri 4'tür, ancak önek geçersiz kılma (prefix override) ile 1 veya 2 olarak değiştirilebilir[6].

Şekil 2.3.'te örnek bir assembly kodu ve Şekil 2.4.'te yığın kullanımı gösterilmiştir.

```
; ESP başlangıç = 0xb20000
mov eax,0AAAAAAAAh
mov ebx,0BBBBBBBBh
mov ecx,0CCCCCCCCh
mov edx,0DDDDDDDDh
push eax
; 0xb1fffc adresi 0xAAAAAAAA değerini içerecektir
; ESP 0xb1fffc (=0xb20000-4) olacaktır.
push ebx
; 0xb1fff8 adresi 0xBBBBBBBB değerini içerecektir
; ESP 0xb1fff8 (=0xb1fffc-4) olacaktır
pop esi
; ESI 0xBBBBBBBB değeri olacaktır
; ESP 0xb1fffc (=0xb1fff8+4) olacaktır
pop edi
; EDI 0xAAAAAAAA değeri olacaktır
; ESP 0xb20000(=0xb1fffc+4) olacaktır
```

Şekil 2.3. Yığın işlemleri örnek kod



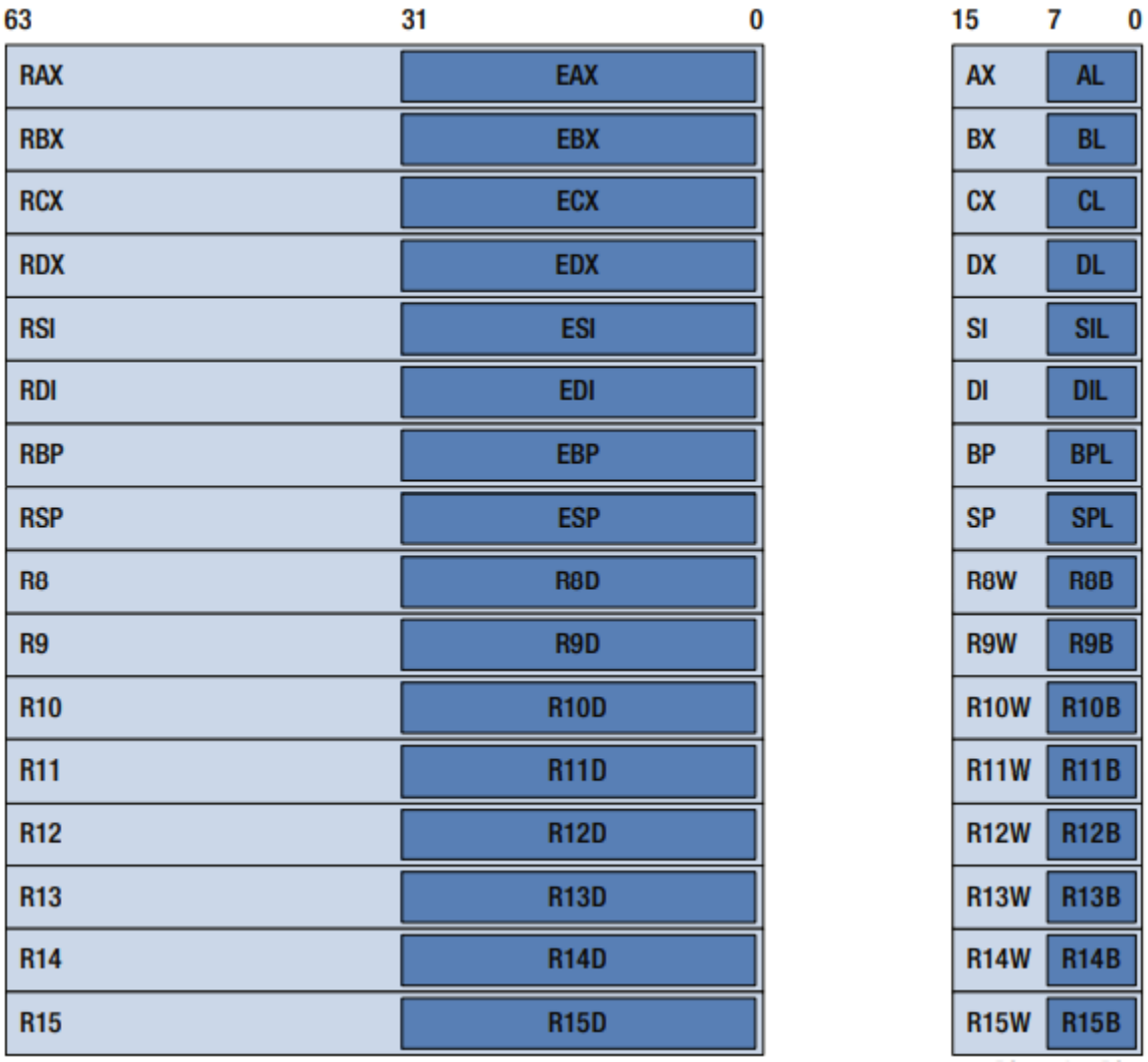
Şekil 2.4. Yığın kullanımı

2.6 X86 - 64

Intel ve AMD tarafından son on yıldır üretilen çoğu x86 işlemci, makinenin kayıt kümesini ve talimat kümesini değiştiren 64 bit modunu destekler. "x86-64" modelini kullanarak programlamayı seçmek, hem bu modu kullanmak hem de işlev çağırma kuralları gibi şeyleri dikte eden belirli bir Uygulama İkili Arayüzünü (ABI) benimsemek anlamına gelir.

x64, x86'nın bir uzantısıdır, bu nedenle mimari özelliklerin çoğu aynıdır, yazmaç boyutu gibi küçük farklılıklar vardır ve bazı talimatlar kullanılamaz. Kayıt seti Şekil 2.5.'de gösterilmektedir.

- Adresler 64 bittir.
- 64 bit tam sayılarda aritmetik ve mantıksal işlemler için doğrudan donanım desteği vardır.
- 16 adet 64-bit genel amaçlı kayıt vardır.
- Argümanları iletmek için (bunları bellekteki yığına geçirmek yerine) kayıtları yoğun bir şekilde kullanan farklı bir çağrı kuralı kullanılır.
- Kayan nokta (floating point) için, x86 komut setinin alt kümesi olan x87 kayan nokta komutları yerine SSE uzantıları tarafından sağlanan *%xmm* kayıt kümesi kullanılır.



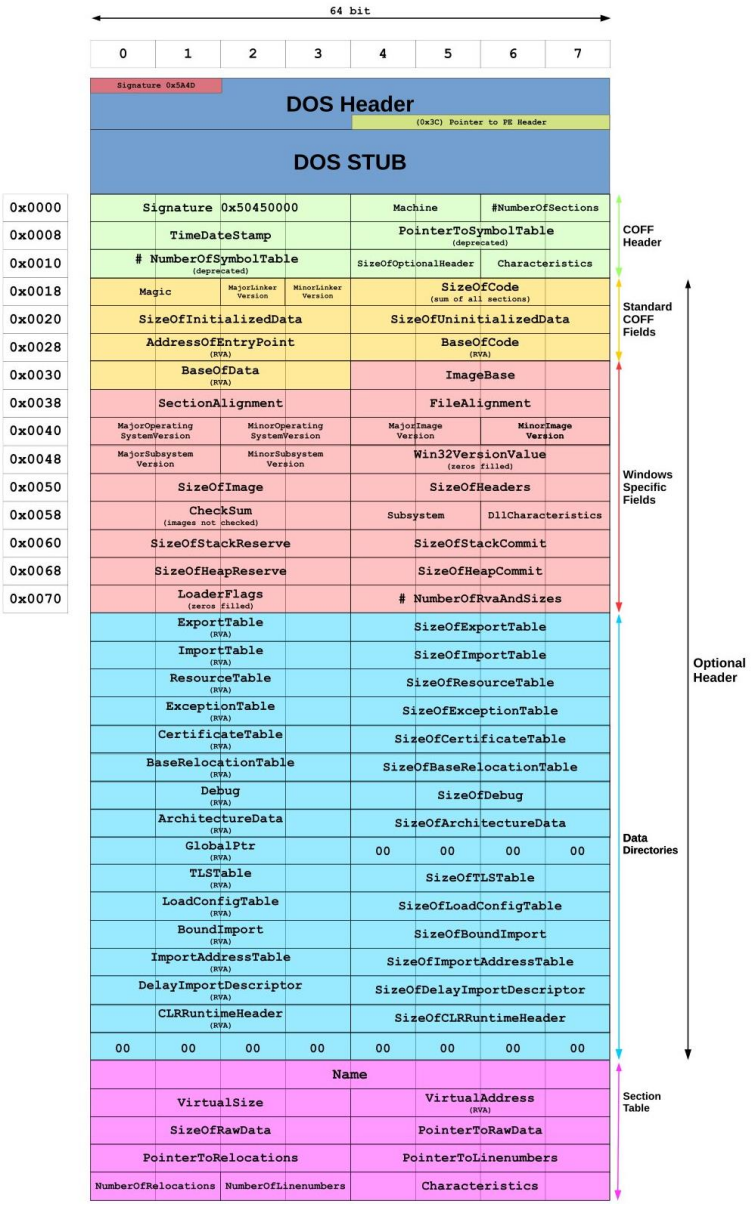
Şekil 2.5. x64 kayıt seti

3. ÖNEMLİ TEMEL BİLGİLER

3.1 PE Dosya Formatı

Portable Executable (PE), taşınabilir, yürütülebilir dosya anlamına gelir, Windows işletim sistemlerinde kullanılan yürütülebilir dosyalar için bir dosya biçimidir ve COFF dosya biçimine (Ortak Nesne Dosyası Biçimi) dayanır.

PE dosyası, işletim sistemi yükleyicisinin bu yürütülebilir dosyayı belleğe yükleyebilmesi ve yürütebilmesi için gerekli bilgileri tutan bir veri yapısıdır. Tipik bir PE dosyasının yapısı Şekil 3.1.'de gösterilmektedir.



Şekil 3.1. PE dosya yapısı

Uygulamalar doğrudan fiziksel belleğe değil, yalnızca sanal belleğe erişir. Başka bir deyişle, bir uygulama tarafından başvurulan bellek adresleri Virtual Address (VA) yani sanal adreslerdir.

Belleğe erişimi sanallaştırmak, uygulamaların kullanılabilir fiziksel belleği kullanma biçiminde esneklik sağlar. Uygulamaların bitişik veya ardışık bir fiziksel bellek parçasını işgal etmesi gerekmez. İşletim sistemi, uygulamalara, sanal adresleri fiziksel bellek adreslerine çevirerek bitişik bir bellek alanı kapladığı yanılsamasını sağlar.

Relative Virtual Address (RVA) yani göreli sanal adres, iki sanal adres arasındaki farktır ve en yüksek olanı ifade eder. Sanal Adres bellekteki özgün adres iken, göreli Sanal Adres (RVA) ImageBase'e göre göreli adrestir. Buradaki ImageBase, yürütülebilir dosyanın belleğe ilk yüklendiği temel adres anlamına gelir.

RVA, (VA – ImageBase) formülü yardımıyla hesaplanır.

3.1.1 MS-DOS başlığı

Her PE dosyası küçük bir MS-DOS yürütülebilir dosyasıyla başlar. Bu, Windows'un popüler bir işletim sistemi haline gelmeden önce gerekli bir bileşendi. Bu küçük stub yürütülebilir dosyası, uygulamayı çalıştırmak için Windows işletim sisteminin gerekli olduğunu belirten bir ileti görüntüler.

PE dosyasının ilk baytları, aslında IMAGE_DOS_HEADER olarak da adlandırılan geleneksel MS-DOS başlığıdır. DOS başlığının yapısı winnt.h dosyasında Şekil 3.2.'deki gibi tanımlanmıştır.

```
typedef struct _IMAGE_DOS_HEADER {      // DOS .EXE header
  WORD   e_magic;                     // Magic number (4D5A / "MZ")
  WORD   e_cblp;                      // Bytes on last page of file
  WORD   e_cp;                        // Pages in file
  WORD   e_crlc;                      // Relocations
  WORD   e_cparhdr;                   // Size of header in paragraphs
  WORD   e_minalloc;                  // Minimum extra paragraphs needed
  WORD   e_maxalloc;                  // Maximum extra paragraphs needed
  WORD   e_ss;                        // Initial (relative) SS value
  WORD   e_sp;                        // Initial SP value
  WORD   e_csum;                      // Checksum
  WORD   e_ip;                        // Initial IP value
  WORD   e_cs;                        // Initial (relative) CS value
  WORD   e_lfarlc;                    // File address of relocation table
  WORD   e_ovno;                      // Overlay number
  WORD   e_res[4];                    // Reserved words
  WORD   e_oemid;                     // OEM identifier (for e_oeminfo)
  WORD   e_oeminfo;                   // OEM information; e_oemid specific
  WORD   e_res2[10];                  // Reserved words
  LONG   e_lfanew;                    // File address of new exe header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Şekil 3.2. MS-DOS başlığı yapısı

Bu başlıktaki iki önemli değer şunlardır:

- e_magic: 0x5A4D (ASCII'DE “MZ” - MS–DOS'un özgün mimarlarından Mark Zbikowski'nin baş harfleri) IMAGE_DOS_SIGNATURE olarak da adlandırılır.
- e_lfanew: IMAGE_NT_HEADERS yapısı olarak da adlandırılır ve ofset 3Ch'de PE başlığının başlangıcındaki dosya ofsetini içerir.

PE Başlığı, diğer birkaç yapıyı bir araya getirerek oluşturulmuştur. Başka bir deyişle, yapıların bir yapısıdır.

3.1.2 IMAGE_NT_HEADERS yapısı

Şekil 3.3.’te IMAGE_NT_HEADERS yapısı gösterilmektedir.

```
typedef struct _IMAGE_NT_HEADERS {
  DWORD                   Signature;
  IMAGE_FILE_HEADER       FileHeader;
  IMAGE_OPTIONAL_HEADER OptionalHeader;
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
```

Şekil 3.3. IMAGE_NT_HEADERS yapısı

3.1.3 IMAGE_FILE_HEADER yapısı

Bu yapı, hedef CPU mimarisi (x86, x64), bölüm sayısı (.text, .data vb..) gibi yürütülebilir dosyanın bazı özellikleri hakkında bilgiler içerir. Ayrıca IMAGE_OPTIONAL_HEADER yapısının boyutunun değerini tutan SizeOfOptionalHeader (değeri E0h olarak ayarlanmalıdır) isimli bir üyeyi içerir. Bu yapı, “isteğe bağlı” olarak adlandırılsa da, her PE yürütülebilir dosyası için gereklidir.

Characteristics üyesi, yürütülebilir modülün türü (.exe veya .dII) gibi yürütülebilir dosyanın bazı özelliklerini tanımlar. Şekil 3.4.’te IMAGE_FILE_HEADER yapısı gösterilmektedir.

```
typedef struct _IMAGE_FILE_HEADER {
  WORD  Machine;
  WORD  NumberOfSections;
  DWORD TimeDateStamp;
  DWORD PointerToSymbolTable;
  DWORD NumberOfSymbols;
  WORD  SizeOfOptionalHeader;
  WORD  Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Şekil 3.4. IMAGE_FILE_HEADER yapısı

3.1.4 IMAGE_OPTIONAL_HEADER yapısı

Yukarıda da belirtildiği gibi, bu yapı isteğe bağlı değildir, IMAGE_OPTIONAL_HEADER yapısı Şekil 3.5.'te gösterilmektedir.

```
typedef struct _IMAGE_OPTIONAL_HEADER {
  WORD                 Magic;
  BYTE                 MajorLinkerVersion;
  BYTE                 MinorLinkerVersion;
  DWORD                SizeOfCode;
  DWORD                SizeOfInitializedData;
  DWORD                SizeOfUninitializedData;
  DWORD                AddressOfEntryPoint;
  DWORD                BaseOfCode;
  DWORD                BaseOfData;
  DWORD                ImageBase;
  DWORD                SectionAlignment;
  DWORD                FileAlignment;
  WORD                 MajorOperatingSystemVersion;
  WORD                 MinorOperatingSystemVersion;
  WORD                 MajorImageVersion;
  WORD                 MinorImageVersion;
  WORD                 MajorSubsystemVersion;
  WORD                 MinorSubsystemVersion;
  DWORD                Win32VersionValue;
  DWORD                SizeOfImage;
  DWORD                SizeOfHeaders;
  DWORD                CheckSum;
  WORD                 Subsystem;
  WORD                 DllCharacteristics;
  DWORD                SizeOfStackReserve;
  DWORD                SizeOfStackCommit;
  DWORD                SizeOfHeapReserve;
  DWORD                SizeOfHeapCommit;
  DWORD                LoaderFlags;
  DWORD                NumberOfRvaAndSizes;
  IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

Şekil 3.5. IMAGE_OPTIONAL_HEADER yapısı

Bu yapı ayrıca PE dosyası hakkında çok önemli bilgiler içerir.

- Magic üyesi, modülün 32 veya 64 bit olup olmadığını tanımlar.
- AddressOfEntryPoint, modülün giriş noktasının (EP) RVA'SINI tutar. Bu, yürütülecek ilk talimatın bulunduğu modülün içindeki adresin RVA'sıdır.
- BaseOfCode ve BaseOfData üyeleri sırasıyla kod ve veri bölümlerinin başlangıcındaki RVA'ları tutar.

- ImageBase üyesi, modülün ImageBase'ini içerir. Bu aslında PE dosyasının belleğe yükleneceği tercih edilen VA'dır. Bu, uygulamalar için varsayılan olarak 0x00400000 dll'ler için varsayılan olarak 0x10000000'dir.
- SectionAlignment ve FileAlignment üyeleri, PE'nin bellekteki ve dosyadaki bölümlerinin sırasıyla hizalandığını gösterir.
- SizeOfImage üyesi, çalışma zamanında PE dosyası tarafından işgal edilen bellek boyutunu gösterir. Değeri, SectionAlignment değerinin bir katı olmalıdır.
- IMAGE_OPTIONAL_HEADER yapısının sonunda, IMAGE_DATA_DIRECTORY yapılarının DataDirectory üyesi bulunur. DataDirectory üyesi temel olarak ilk IMAGE_DATA_DIRECTORY yapısına bir işaretçidir.

3.1.5 IMAGE_DATA_DIRECTORY yapısı

Şekil 3.6.'da IMAGE_DATA_DIRECTORY yapısı gösterilmektedir.

```
typedef struct _IMAGE_DATA_DIRECTORY {
  DWORD VirtualAddress;
  DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

Şekil 3.6. IMAGE_DATA_DIRECTORY yapısı

Bu yapıların her biri, RVA'yı ve çalışma zamanında PE görüntüsünün içindeki belirli verilerin boyutunu içerir. Bunlara örnek olarak ExportTableAddress (dışa aktarılan işlevler tablosu), ImportTableAddress (içe aktarılan işlevler tablosu), ResourcesTable (PE'ye gömülü resimler gibi kaynaklar tablosu) ve içe aktarılan işlevlerin adreslerini çalışma zamanında depolayan ImportAddressTable (IAT) verilebilir.

3.1.6 Bölüm tablosu (section table)

Bölüm tablosu, IMAGE_SECTION_HEADER yapıları dizisidir. Her yapı ilişkili bölümü hakkında konum, boyut ve bu bölümdeki erişim izinlerini açıklayan özellikleri içerir. IMAGE_SECTION_HEADER yapısı Şekil 3.7.'deki gibidir.

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
    DWORD PhysicalAddress;
    DWORD VirtualSize;
  } Misc;
  DWORD VirtualAddress;
  DWORD SizeOfRawData;
  DWORD PointerToRawData;
  DWORD PointerToRelocations;
  DWORD PointerToLinenumbers;
  WORD  NumberOfRelocations;
  WORD  NumberOfLinenumbers;
  DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

Şekil 3.7. IMAGE_SECTION_HEADER yapısı

- Bölüm adı en fazla 8 ASCII karakter uzunluğunda olabilir, bu nedenle NAME üyesi her zaman bellekte 8 bayt yer kaplar.
- VirtualSize, bellekteki bölümün boyutunu,
- VirtualAddress, bellekteki bölümün RVA'sını,
- SizeOfRawData, dosyadaki bölümün gerçek boyutunu,
- PointerToRawData, ham veri bölümünün dosyada başladığı uzaklığı,
- Characteristics, bellekteki o bölümün bellek erişim haklarını (R, RW, RWE vb.) içerir.

En çok kullanılan bölüm adları ve içerikleri Tablo 3.1'de gösterilmektedir.

Tablo 3.1 Bölüm adları ve içerikleri

Bölüm	**İçerik**
.text	Yürütülebilir Kodlar
.data	Global ve Statik Veriler (INIT.)
.bss	Global ve Statik Veriler (UNINIT.)
.rsrc	Kaynaklar
.idata	İçe Aktarım Tabloları
.rdata	.data Bölümünün Salt Okunur Hali ve Debug Bilgileri
.edata	Dışa Aktarım Tabloları
.reloc	Relocation Bilgileri
.tls	Thread Local Storage

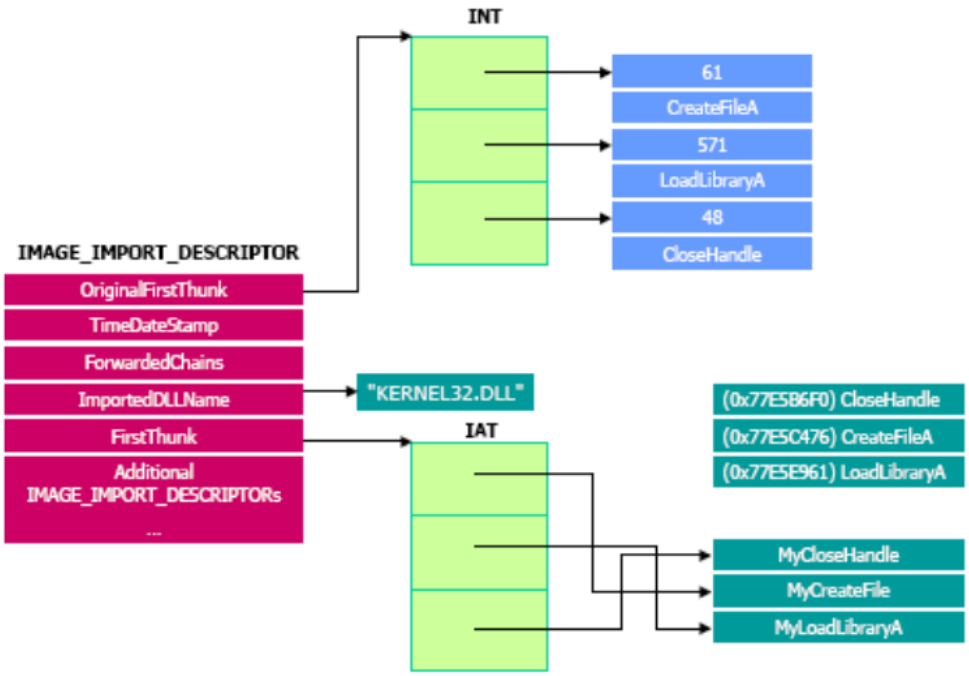
3.1.7 Export address table

Her DLL, işlevlerini herkese açık hale getirirken Export Adress Table kullanır. Windows bir programı ve onun DLL'lerini yüklediğinde, hangi DLL'lerin yükleneceğini bulmak için yüklediği programın Import Address Table'ını kontrol eder. Ardından DLL'leri belleğe yükler ve işlevlerinin nerede olduğunu öğrenmek için DLL'nin Export Address Table'ını arar.

3.1.8 Import address table

Import Address Table işlev işaretçilerinden oluşur ve DLL'ler yüklendiğinde işlevlerin adreslerini almak için kullanılır. Derlenmiş bir uygulama, tüm API çağrılarının doğrudan kodlanmış adreslerini kullanmayacağı, bunun yerine bir işlev işaretçisi üzerinden çalışacağı şekilde tasarlanmıştır. İşaretçi tablosuna doğrudan [işaretçi adresi] çağrısıyla veya "Import Lookup Table" ve "Hint / Name Table" ile erişilebilir.

Import Address Table, IMAGE_IMPORT_DESCRIPTOR adı verilen ve IAT ile aynı olan Import Name Table (INT) adlı başka bir arama tablosunu da içeren daha büyük bir veri yapısının parçasıdır. Şekil 3.8.'de örnek bir IMAGE_IMPORT_DESCRIPTOR yapısı görülmektedir.



Şekil 3.8. IMAGE_IMPORT_DESCRIPTOR yapısı

OriginalFirstThunk, Import Name Table'ı ve FirstThunk, Import Address Table'ı işaret eder. DLL'den içeri aktarılan her işlev için 2 adet _IMAGE_THUNK_DATA yapısı bulunur. Şekil 3.9.'da IMAGE_THUNK_DATA yapısı gösterilmektedir.

```
typedef struct _IMAGE_THUNK_DATA {
    union {
        uint32_t* Function;             // içe aktarılan işlevin bellek adresi
        uint32_t  Ordinal;              // içe aktarılan API'nin sıra değeri
        PIMAGE_IMPORT_BY_NAME AddressOfData;        // içe aktarılan ismin RVA'sı
        DWORD ForwarderStringl              // RVA to forwarder string
    } ul;
} IMAGE_THUNK_DATA, *PIMAGE_THUNK_DATA;
```

Şekil 3.9. IMAGE_THUNK_DATA yapısı

FirstThunk kullanıldığında bu yapıdaki Function girdisi import edilen işlevin RVA değerine, OriginalFirstThunk kullanıldığında bu yapıdaki AddressOfData girdisi IMAGE_IMPORT_BY_NAME yapısına işaret eder ve Şekil 3.10'daki Name elemanı kullanılan işlevin ismini verir.

```
typedef struct _IMAGE_IMPORT_BY_NAME {
    uint16_t  Hint;     // Kullanılacak olası sıra numarası
    uint8_t   Name[1];  // İşlevin adı (Null sonlandırıcı içerir)
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

Şekil 3.10. IMAGE_IMPORT_BY_NAME yapısı

3.1.9 Bellek ve dosya hizalama

Bir yürütülebilir dosyanın fiziksel dosyadaki bölümlerinin hizalanması, genellikle çalışma zamanında bellekteki görüntüsünden farklıdır.

Tipik olarak, dosyadaki bölümlerin varsayılan hizalaması 0x200 ve bellekte varsayılan hizalaması 0x1000'dir, bu elbette aynı hizalama değerinin gerektiğinde her iki durumda da kullanılamayacağı anlamına gelmez. Bu değerler, dosyadaki veya bellekteki bir bölümün boyutunun bu değerlerin katı olması gerektiğini gösterir ve IMAGE_OPTIONAL_HEADER yapısı içinde, FileAlignment ve SectionAlignment adlı iki 32-bit üyeye depolanır. Bu varsayılan değerlerin verilmesinin nedeni, dosyadaki her bölümün başlangıcının, genellikle 0x200 (512) bayt boyutunda olan bir disk sektörünün başlangıcına karşılık gelmesini sağlamaktır. Bağlayıcı, yürütülebilir dosyayı oluştururken, bölümün hizalama değerini kontrol eder ve eğer 0x200 ise, .text bölümünün boyutunu, fazladan baytlarla (genellikle 0 baytlarla) doldurarak 0x200'ün en yakın katına (0x400) değiştirir. Örneğin, .text bölümündeki kodun boyutu 0x1050 bayt ise, bağlayıcı 0x200'ün

katı olan bir değere ulaşmak için fazladan baytlarla dolduracak ve en yakın değer olan 0x1200'e tamamlayacaktır. Ancak bellekte, 0x1000'in katı olan bir sayıya hizalanmalıdır ve bu örnekte 0x1050'ye en yakın sayı 0x2000'dir.

3.2 Windows Uygulama Programlama Arayüzü (WinAPI)

Windows uygulama programlama arayüzü (API), Windows işletim sistemi ailesinin kullanıcı modu sistem programlama arayüzüdür. Windows'un 64-bit sürümlerinin piyasaya sürülmesinden önce, Windows işletim sistemlerinin 32-bit sürümlerine yönelik programlama arayüzü, Windows'un programlama arayüzü olan orijinal 16-bit Windows API'sinden ayırt edilmesi için Win32 API olarak adlandırılıyordu.

Windows API, aşağıdaki ana kategorilere ayrılan binlerce çağrılabilir işlevden oluşur:

- Sistem Hizmetleri
- Kullanıcı Arayüzü
- Grafik ve Multimedya
- Veri Erişimi ve Depolama
- Ağ ve İnternet
- Güvenlik ve Kimlik
- Cihazlar
- Uygulama Kurulumu ve Servisi
- Teşhis
- Windows Ortamı (Kabuk)
- Sistem Yöneticisi ve Yönetimi
- Kullanıcı Girişi ve Mesajlaşma

3.2.1 Önemli windows dinamik bağlantı kitaplıkları

KERNEL32.DLL: İşlemler ve iş parçacıkları yönetimi dahil olmak üzere dosya Giriş/Çıkış işlemlerinden ve bellek yönetiminden sorumlu Windows'un temel işlevlerini içerir. Bazı işlevler, NTDLL kitaplığında daha fazla yerel API çağırmak için yardımcıdır.

USER32.DLL: Program pencereleri, menü ve simgeler gibi ekran ve grafik arayüzle ilgilenen işlevleri içerir. Ayrıca pencere mesajlarını kontrol eden işlevleri içerir.

NTDLL.DLL: Öncelikle alt sistem DLL'lerinin kullanımı için özel bir sistem destek kitaplığıdır. İki tür işlev içerir:

- Windows yürütme sistemi hizmetlerine sistem hizmeti gönderme taslakları,
- Alt sistemler, alt sistem DLL'leri ve diğer yerel görüntüler tarafından kullanılan dahili destek işlevleri.

İlk işlev grubu, kullanıcı modundan çağrılabilen Windows yürütme sistemi hizmetlerine arabirim sağlar. NtCreateFile, NtSetEvent vb. gibi 400'den fazla işlev vardır. Bu işlevlerin çoğuna Windows API aracılığıyla erişilebilir.

Bu işlevlerin her biri için Ntdll, aynı ada sahip bir giriş noktası içerir. İşlevin içindeki kod, sistem hizmeti dağıtıcısını çağırmak için çekirdek moduna geçişe neden olan mimariye özgü talimatı içerir; bu talimat, bazı parametreleri doğruladıktan sonra, gerçek çekirdek modu sistem hizmetini çağırır. Ntoskrnl.exe içindeki gerçek kodu içerir.

Ntdll ayrıca görüntü yükleyici (Ldr ile başlayan işlevler), yığın yöneticisi ve Windows alt sistem işlem iletişim işlevleri (Csr ile başlayan işlevler) gibi birçok destek işlevleri içerir. Ntdll ayrıca genel çalışma zamanı kitaplığı yordamlarını (Rtl ile başlayan işlevler), kullanıcı modu hata ayıklaması (DbgUi ile başlayan işlevler), Windows için olay izleme (ETW'de başlayan işlevler) kullanıcı modu eş zamansız prosedür çağrısı (APC) göndericisi ve özel durum göndericisi için destek işlevleri içerir[7].

ADVAPI32.DLL: Bu kitaplık, kayıt defterine erişecek işlevleri içerir.

MSVCRXX.DLL: Bu kitaplık Microsoft Visual C çalışma zamanı işlevlerini içerir ve programın Visual C kullanılarak derlendiğini söyler. (Burada XX bir sürüm numarasıdır)

WININET.DLL: Bu kitaplık, internete erişen işlevleri içerir.

MSVBVMXX.DLL : Bu kitaplık Microsoft Visual Basic çalışma zamanı işlevlerini içerir ve Visual Basic kullanılarak derlendiğini söyler. (Burada XX sürüm numarasıdır)

3.2.2 Windows uygulama programlama arabiriminde (API) sık kullanılan işlevler

Bazı işlevler, ASCII sürümü için -A ve Unicode sürümü için -W son ekine sahip olabilir.

- Kayıt defteri erişimi (advapi32.dll): RegEnumKeyEx, RegEnumValue, RegGetValue, RegOpenKeyEx, RegQueryValueEx
- .text .ini dosyalarına erişim (kernel32.dll): GetPrivateProfileString
- İletişim kutuları (user32.dll): MessageBox, MessageBoxEx, CreateDialog, SetDlgItemText, GetDlgItemText
- Kaynak erişimi : (user32.dll): LoadMenu.

- TCP/IP ağ iletişimi (ws2_32.dll): WSARecv, WSASend
- Dosya erişimi (kernel32.dll): CreateFile, ReadFile, ReadFileEx, WriteFile, WriteFileEx
- İnternete yüksek düzeyde erişim (wininet.dll): WinHttpOpen
- Yürütülebilir bir dosyanın (wintrust.dll) dijital imzasını kontrol etme: WinVerifyTrust
- Standart MSVC kitaplığı (dinamik olarak bağlıysa) (msvcr*.dll): assert, itoa, ltoa, open, printf, read, strcmp, atol, atoi, fopen, fread, fwrite, memcmp, rand, strlen, strstr, strchr.

3.2.3 İş parçacığı ortamı bloğu (thread environment block – TEB)

İş Parçacığı Ortam Bloğu (TEB), işlemdeki çalışan iş parçacıkları hakkında bilgi depolayan bir veri yapısıdır. Farklı işletim sistemlerinde TEB yapısının morfolojisi farklıdır. Şekil 3.11.'de TEB yapısı gösterilmektedir.

```
struct _TEB {
    0x000 _NT_TIB NtTib;
    0x01c void* EnvironmentPointer;
    0x020 _CLIENT_ID ClientId;
    0x028 void* ActiveRpcHandle;
    0x02c void* ThreadLocalStoragePointer;
    0x030 _PEB* ProcessEnvironmentBlock;  /* In WinNT (incl. h
    0x034 DWORD LastErrorValue;
    0x038 DWORD CountOfOwnedCriticalSections;
    0x03c void* CsrClientThread;
    0x040 void* Win32ThreadInfo;
    0x044 DWORD User32Reserved[26];
    0x0ac DWORD UserReserved[5];
    0x0c0 void* WOW32Reserved;
    0x0c4 DWORD CurrentLocale;
    0x0c8 DWORD FpSoftwareStatusRegister;
    0x0cc void* SystemReserved1[54];
    0x1a4 int ExceptionCode;
    0x1a8 _ACTIVATION_CONTEXT_STACK ActivationContextStack;
    0x1bc DWORD SpareBytes1[24];
    0x1d4 _GDI_TEB_BATCH GdiTebBatch;
    0x6b4 _CLIENT_ID RealClientId;
    0x6bc void* GdiCachedProcessHandle;
    0x6c0 DWORD GdiClientPID;
    0x6c4 DWORD GdiClientTID;
    0x6c8 void* GdiThreadLocalInfo;
    0x6cc DWORD Win32ClientInfo[62];
    0x7c4 void* glDispatchTable[233];
    0xb68 DWORD glReserved1[29];
    0xbdc void* glReserved2;
    0xbe0 void* glSectionInfo;
    0xbe4 void* glSection;
    0xbe8 void* glTable;
    0xbec void* glCurrentRC;
    0xbf0 void* glContext;
    0xbf4 DWORD LastStatusValue;
    0xbf8 _UNICODE_STRING StaticUnicodeString;
    0xc00 WORD StaticUnicodeBuffer[261];
    0xe0c void* DeallocationStack;
    0xe10 void* TlsSlots[64];
    0xf10 _LIST_ENTRY TlsLinks;
    0xf18 void* Vdm;
    0xf1c void* ReservedForNtRpc;
    0xf20 void* DbgSsReserved[2];
    0xf28 DWORD HardErrorsAreDisabled;
    0xf2c void* Instrumentation[16];
    0xf6c void* WinSockData;
    0xf70 DWORD GdiBatchCount;
    0xf74 UChar InDbgPrint;
    0xf75 UChar FreeStackOnTermination;
    0xf76 UChar HasFiberData;
    0xf77 UChar IdealProcessor;
    0xf78 DWORD Spare3;
    0xf7c void* ReservedForPerf;
    0xf80 void* ReservedForOle;
    0xf84 DWORD WaitingOnLoaderLock;
    0xf88 _Wx86ThreadState Wx86Thread;
    0xf94 void** TlsExpansionSlots;
    0xf98 DWORD ImpersonationLocale;
    0xf9c DWORD IsImpersonating;
    0xfa0 void* NlsCache;
    0xfa4 void* pShimData;
    0xfa8 DWORD HeapVirtualAffinity;
    0xfac void* CurrentTransactionHandle;
    0xfb0 _TEB_ACTIVE_FRAME* ActiveFrame;
};
```

Şekil 3.11. İş parçacığı ortamı bloğu yapısı

3.2.4 İşlem ortamı bloğu (process environment block – PEB)

İşlem Ortamı Bloğu (PEB), uygulamalar tarafından yüklenen modüllerin listesi, işlem başlatma argümanları, yığın adresi, programın hata ayıklanıp ayıklanmadığını kontrol etme, içe aktarılan DLL'lerin görüntü temel adresini bulma gibi bilgileri almak için kullanılabilen kullanıcı modu veri yapısıdır. Şekil 3.12.'de 32-bitlik bir Windows için PEB yapısı gösterilmektedir.

```
struct _PEB {
0x000 BYTE InheritedAddressSpace;
0x001 BYTE ReadImageFileExecOptions;
0x002 BYTE BeingDebugged;
0x003 BYTE SpareBool;
0x004 void* Mutant;
0x008 void* ImageBaseAddress;
0x00c _PEB_LDR_DATA* Ldr;
0x010 _RTL_USER_PROCESS_PARAMETERS* ProcessParameters;
0x014 void* SubSystemData;
0x018 void* ProcessHeap;
0x01c _RTL_CRITICAL_SECTION* FastPebLock;
0x020 void* FastPebLockRoutine;
0x024 void* FastPebUnlockRoutine;
0x028 DWORD EnvironmentUpdateCount;
0x02c void* KernelCallbackTable;
0x030 DWORD SystemReserved[1];
0x034 DWORD ExecuteOptions:2;          // bit offset: 34, len=2
0x034 DWORD SpareBits:30;       // bit offset: 34, len=30
0x038 _PEB_FREE_BLOCK* FreeList;
0x03c DWORD TlsExpansionCounter;
0x040 void* TlsBitmap;
0x044 DWORD TlsBitmapBits[2];
0x04c void* ReadOnlySharedMemoryBase;
0x050 void* ReadOnlySharedMemoryHeap;
0x054 void** ReadOnlyStaticServerData;
0x058 void* AnsiCodePageData;
0x05c void* OemCodePageData;
0x060 void* UnicodeCaseTableData;
0x064 DWORD NumberOfProcessors;
0x068 DWORD NtGlobalFlag;
0x070 _LARGE_INTEGER CriticalSectionTimeout;
0x078 DWORD HeapSegmentReserve;
0x07c DWORD HeapSegmentCommit;
0x080 DWORD HeapDeCommitTotalFreeThreshold;
0x084 DWORD HeapDeCommitFreeBlockThreshold;
0x088 DWORD NumberOfHeaps;
0x08c DWORD MaximumNumberOfHeaps;
0x090 void** ProcessHeaps;
0x094 void* GdiSharedHandleTable;
0x098 void* ProcessStarterHelper;
0x09c DWORD GdiDCAttributeList;
0x0a0 void* LoaderLock;
0x0a4 DWORD OSMajorVersion;
0x0a8 DWORD OSMinorVersion;
0x0ac WORD OSBuildNumber;
0x0ae WORD OSCSDVersion;
0x0b0 DWORD OSPlatformId;
0x0b4 DWORD ImageSubsystem;
0x0b8 DWORD ImageSubsystemMajorVersion;
0x0bc DWORD ImageSubsystemMinorVersion;
0x0c0 DWORD ImageProcessAffinityMask;
0x0c4 DWORD GdiHandleBuffer[34];
0x14c void (*PostProcessInitRoutine)();
0x150 void* TlsExpansionBitmap;
0x154 DWORD TlsExpansionBitmapBits[32];
0x1d4 DWORD SessionId;
0x1d8 _ULARGE_INTEGER AppCompatFlags;
0x1e0 _ULARGE_INTEGER AppCompatFlagsUser;
0x1e8 void* pShimData;
0x1ec void* AppCompatInfo;
0x1f0 _UNICODE_STRING CSDVersion;
0x1f8 void* ActivationContextData;
0x1fc void* ProcessAssemblyStorageMap;
0x200 void* SystemDefaultActivationContextData;
0x204 void* SystemAssemblyStorageMap;
0x208 DWORD MinimumStackCommit;
};
```

Şekil 3.12. İşlem ortamı bloğu yapısı

3.3 Endianness

Bellekteki değerleri temsil etmenin bir yoludur. İşlemciler baytları saklarken önemli baytın solda veya sağda olmasına göre sınıflandırılır. Big-endian ve Little-endian olmak üzere iki çeşit sıralama vardır.

3.3.1 Big-endian

Önemli baytın solda olduğu sıralama çeşididir. 0x12345678 değeri bellekte Tablo 3.2.'deki gibi temsil edilir. Motorola 68k, IBM POWER, Sun Sparc işlemcileri Big-endian biçimini kullanır.

Tablo 3.2. Big-endian temsili

Bellekteki Adres	Bayt Değeri
+0	0x12
+1	0x34
+2	0x56
+3	0x78

3.3.2 Little-endian

Önemli baytın sağda olduğu sıralama çeşididir. 0x12345678 değeri bellekte Tablo 3.3.'teki gibi temsil edilir. Intel IA-32(i386 veya x86-32) ve x86-64 serisi işlemciler, Little-endian biçimini kullanır.

Tablo 3.3. Little-endian temsili

Bellekteki Adres	Bayt Değeri
+0	0x78
+1	0x56
+2	0x34
+3	0x12

3.4 İşaretli Sayı Gösterimleri

İşaretli sayıları temsil etmenin birkaç yöntemi vardır ve en popüler olanı "ikinin tümleyeni" yöntemidir. Tablo 3.4.'te bazı bayt değerlerinin işaretli ve işaretsiz sayılarının farkı gösterilmiştir.

Tablo 3.4. İşaretli ve işaretsiz sayı gösterimleri

İkili	Onaltılık	İşaretsiz	İşaretli
01111111	0x7f	127	127
01111110	0x7e	126	126
00000011	0x3	3	3
00000010	0x2	2	2
00000001	0x1	1	1
00000000	0x0	0	0
11111111	0xff	255	-1
11111110	0xfe	254	-2
11111101	0xfd	253	-3
10000001	0x81	129	-127
10000000	0x80	128	-128

0xFFFFFFFE ve 0x00000002 işaretsiz sayı olarak temsil edildiğinde, ilk sayının (4294967294) ikinciden (2) daha büyük olduğu görülmektedir. Her iki sayı da işaretli olarak temsil edilirse, birinci sayı -2 olmakta ve ikinci sayıdan (2) daha küçük olduğu görülmektedir. Koşullu atlamaların hem işaretli (JG, JL) hem de işaretsiz (JA, JB) işlemler için mevcut olmasının nedeni budur.

3.5 Çağırma Kuralları (Calling Conventions)

Çağırma kuralı, işlev çağrılarının makine düzeyinde nasıl çalıştığını belirleyen bir dizi kuraldır. Belirli bir sistem için Uygulama İkili Arayüzü (ABI) tarafından tanımlanır. Çağırma kuralları, argümanların bir fonksiyona nasıl iletildiğini, bir fonksiyondan dönüş değerlerinin nasıl geri iletildiğini, fonksiyonun nasıl çağrıldığını ve fonksiyonun yığını ve yığın çerçevesini nasıl yönettiğini belirtir. Kısacası, çağırma kuralı, işlev çağrısının nasıl assembly diline dönüştürüleceğini belirtir. Pek çok çağrı kuralı vardır, ancak popüler olanları *__cdecl, __stdcall, __thiscall* ve *__fastcall*'dır. Derleyici ayrıca kendi özel çağrı kurallarını da oluşturabilir. Tablo 3.5.'te bazı çağırma kurallarının özellikleri görülmektedir.

Tablo 3.5 Çağırma kuralları ve özellikleri

	CDECL	**STDCALL**	**FASTCALL**
Parametreler	Veriler, yığına sağdan sola doğru kaydedilir. Çağıran, çağrı yapıldıktan sonra yığını temizler.	Veriler, yığına sağdan sola doğru kaydedilir. Çağrılan, çağrı yapıldıktan sonra yığını temizler.	İlk iki parametre ECX ve EDX'e kaydedilir. Geriye kalanlar yığına kaydedilir.
Geri Dönüş Değeri	EAX yazmacında saklanır.	EAX yazmacında saklanır.	EAX yazmacında saklanır.
Uçucu olmayan(Non-volatile) Yazmaçlar	EBP, ESP, EBX, ESI, EDI	EBP, ESP, EBX, ESI, EDI	EBP, ESP, EBX, ESI, EDI

3.6 Kesme Noktaları (Breakpoints)

Kesme noktası, hata ayıklayıcıya belirli bir noktada bir işlemin çalışmasını durdurmasını söyleyen bir sinyaldir. Hata ayıklayıcılar, hedef uygulamalarda çeşitli şekillerde kesme noktaları oluşturabilir. Bunu kullanarak, tersine mühendisler Windows API'leri gibi alanlara kesme noktaları yerleştirebilir.

Yazılım kesme noktaları ve donanım kesme noktaları, benzer sonuçları kolaylaştırır ancak farklı şekilde çalışır. Bir işleme bir yazılım kesme noktası eklendiğinde, hata ayıklayıcı kesme noktası konumundaki talimatı okur, bu talimatın ilk baytını kaldırır ve onu, INT 3 talimatı anlamına gelen 0xCC kesme noktası işlem koduyla değiştirir. Orijinal bayt bir tabloya kaydedilir ve kesme noktasına isabet edildiğinde değiştirilir, böylece yürütmenin devam etmesine izin verilir.

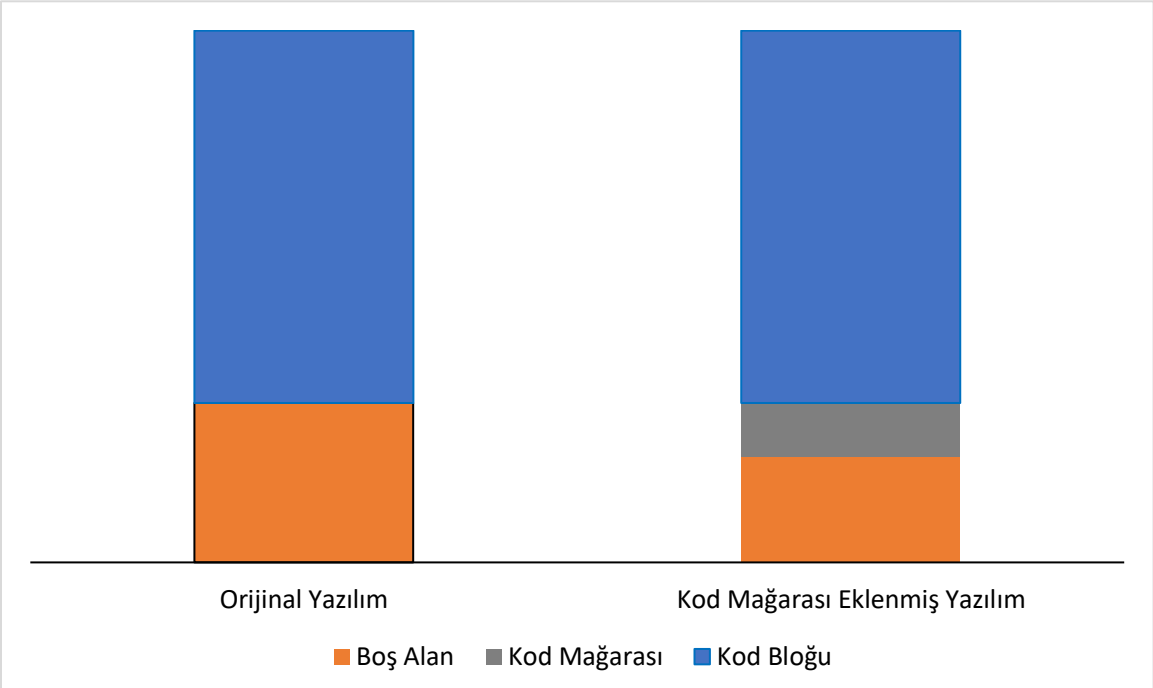
Donanım kesme noktaları, işlemci donanımının kendisinde uygulanır. Donanım, program adres veri yolunda belirli adreslerin belirlenmesine ayrılmış kayıtlar içerir. Veri yolundaki adres, hata ayıklama kayıtlarında saklanan adreslerle eşleştiğinde, bir kesme noktası sinyali (INT 1) gönderilir ve CPU işlemi durdurur. x86 mimarisinde DR0 ila DR7 olarak adlandırılan sekiz adet hata ayıklama kaydı vardır. DR0 ila DR3 kayıtları, kesme noktası istenilen adresleri içerirken, DR7, DR0 ila DR3 kesme noktalarının her birini etkinleştirmek veya devre dışı bırakmak için bitler içerir. DR6, bir hata ayıklayıcının hangi

hata ayıklama kaydının tetiklendiğini bilmesini sağlamak için bir durum kaydı olarak kullanılır. Çipten hata ayıklama kaydı bilgilerini okumak için GetCurrentThreadContext çağrısı kullanılır.

3.7 Code Caves (Kod Mağaraları)

Kod mağaraları, derlenmiş bir yazılıma yeni kodların eklenmesi için kullanılan veya oluşturulan boş alanlara verilen isimdir. Derlenmiş bir yazılıma yeni işlevler eklemek, programın çalışma şeklini değiştirmek ve cracking işlemlerinde keygen (anahtar oluşturucu) yaratmak için kullanılabilir.

Kod mağaraları ile işlem yapmak için öncelikle yazılımda kullanılmayan bir alan bulunur. Daha sonra, bu boş alana yeni kodlar eklenir. Eklenen kodun yürütülmesi için orijinal kod akışından kod mağarasına bir atlama komutu ile geçiş yapılır. Kod mağarasının sonuna, normal kod akışının devam etmesi için geri atlama komutu eklenir. Şekil 3.13.'te Orijinal yazılım ve kod mağarası eklenmiş yazılım alanları görülmektedir.



Şekil 3.13. Orijinal yazılım ve kod mağarası eklenmiş yazılım

Bazen hedef yazılım, eklenmesi düşünülen kodun sığabileceği kadar boş alana sahip değildir. Bu durumda, yazılıma tamamen yeni bir bölüm eklenir, ayrıcalıkları ayarlanır ve bu yeni bölüm kod mağarası olarak kullanılır.

4. TERSİNE MÜHENDİSLİĞİ ÖNLEME YÖNTEMLERİ VE BUNLARI AŞMAK

Tersine mühendisliği önleme, hedef dosyada tersine mühendisliği ve hata ayıklama girişimlerini engelleyen bir veya daha fazla tekniğin uygulanmasıdır. Kendini işine adamış bir tersine mühendisi engellemek için kusursuz bir çözüm yoktur; ancak, görevi mümkün olduğunca çetin ve zor hale getirmek, yazılımların tam analizi için gereken süreyi arttırır ve kişinin tersine mühendislikteki bilgi birikimi ve uzmanlık seviyesine bağlı olarak ulaşmak istediği sonuca varmasını zorlaştırır veya engeller.

4.1 Hata Ayıklama Bayrakları

İşlem belleğinde bulunan ve işletim sisteminin ayarladığı sistem tablolarındaki özel bayraklar, işlemde hata ayıklanıp ayıklanmadığını belirtmek için kullanılabilir. Bu bayrakların durumları, belirli API işlevleri kullanılarak veya bellekteki sistem tablolarının incelenmesiyle doğrulanabilir.

4.1.1 BeingDebugged / IsDebuggerPresent

En temel hata ayıklayıcı algılama tekniği, İşlem Ortamı Bloğundaki (PEB) BeingDebugged bayrağının kontrol edilmesi ile gerçekleşir. IsDebuggerPresent() fonksiyonu PEB'nin adresini almak için İş Parçacığı Ortam Bloğuna (TEB) erişir ve ardından işlemin bir kullanıcı modu hata ayıklayıcı tarafından hata ayıklanıp ayıklanmadığını belirlemek için PEB'nin 0x02 ofsetindeki BeingDebugged bayrağını kontrol eder[8]. Bazı paketleyiciler, IsDebuggerPresent()'i çağırmak yerine, PEB'yi BeingDebugged bayrağı için manuel olarak kontrol eder. Bu teknik, PEB.BeingDebugged bayrağının değeri 0x00 bayt değeriyle değiştirilerek kolayca atlanabilir. Şekil 4.1.'de BeingDebugged bayrağınınm direkt kontrolü gösterilmiştir.

```
mov     eax,dword ptr fs:[0x30h] ; EAX = TEB.ProcessEnvironmentBlock
movzx   eax,byte ptr [eax+0x02] ; AL=PEB.BeingDebugged
test eax,eax
jnz ,debugger found
```

Şekil 4.1. BeingDebugged bayrağının kontrolü

4.1.2 CheckRemoteDebuggerPresent / NtQuerryInformationProcess

Kernel32!CheckRemoteDebuggerPresent() işlevi, geçerli işleme bir hata ayıklayıcının (aynı makinede farklı bir işlemde) eklenip eklenmediğini kontrol eder. Bu işlev, ProcessInformationClass parametresinin ProcessDebugPort(7) olarak ayarlanmış bir şekilde ntdll!NtQueryInformationProcess() işlevini dahili olarak çağırır. Ayrıca, çekirdeğin içinde, NtQueryInformationProcess(), EPROCESS çekirdek yapısının DebugPort alanını sorgular. DebugPort alanındaki sıfır olmayan bir değer, işlemin kullanıcı modu hata ayıklayıcı tarafından hata ayıklandığını gösterir ve ProcessInformation, 0xFFFFFFFF değeriyle ayarlanır.

Bu hata ayıklamayı önleme tekniğini aşmak için, NtQueryInformationProcess() işlevinin geri döndüğü kod bloğuna bir kesme noktası ayarlanır, ardından kesme noktasına ulaşıldığında ProcessInformation, 0x00 değeri ile değiştirilerek atlanır.

4.1.3 NTGlobalFlag

İşlem Ortamı Bloğu, NTGlobalflag adlı bir alana da sahiptir ve bu alan programlarda hata ayıklama işlemi yapılıp yapılmadığını belirlemek için kullanılabilir. 32-bit Windows'ta offseti 0x68 ve 64-bit Windows'ta offseti 0xBC olan NtGlobalFlag alanının değeri varsayılan olarak 0x00'dır. Hata ayıklayıcı eklemek, NtGlobalFlag değerini değiştirmez. Ancak işlem bir hata ayıklayıcı tarafından oluşturulmuşsa, Tablo 4.1.'deki bayraklar ayarlanır. Bir hata ayıklayıcının varlığı, bu bayrakların kombinasyonunu kontrol ederek tespit edilebilir.

Tablo 4.1. NTGlobalFlag bayrakları ve değerleri

Bayrak	**Değer**
FLG_HEAP_ENABLE_TAIL_CHECK	0x10
FLG_HEAP_ENABLE_FREE_CHECK	0x20
FLG_HEAP_VALIDATE_PARAMETERS	0x40
Toplam	0x70

NtGlobalFlag kontrolünü atlamak için kontrolden önce yapılan işlemleri tersine çevirmek yeterlidir; başka bir deyişle, hata ayıklanmış işlemin PEB yapısının NtGlobalFlag alanını, kontrol edilmeden önce 0x00 olarak ayarlamak yeterlidir.

4.1.4 NtQuerySystemInformation

Ntdll!NtQuerySystemInformation() işlevi, Windows Nt'den beri var olan SystemKernelDebuggerInformation (0x23) sınıfını içerir. SystemKernelDebuggerInformation sınıfı Al yazmacı için KdDebuggerEnabled ve Ah yazmacı için KdDebuggerNotPresent bayrak değerini döndürür. İşlem bir çekirdek hata ayıklayıcısı tarafından ayıklanıyorsa, Ah yazmacının dönüş değeri sıfırdır.

Bu hata ayıklamayı önleme tekniğini aşmanın yolu, döndürülen SYSTEM_KERNEL_DEBUGGER_INFORMATION ::DebuggerEnabled değerinin 0 ile değiştirilmesi ve SYSTEM_KERNEL_DEBUGGER_INFORMATION::DebuggerNotPresent değerinin 1 ile değiştirilmesidir.

4.2 Nesne Tanıtıcıları (Object Handles)

Hata ayıklama başladığında işletim sistemi tarafından oluşturulan belirli çekirdek nesneleri vardır. Çekirdek nesne tanıtıcılarını parametreleri olarak kabul eden bazı WinAPI işlevleri hata ayıklama altında farklı davranabilir. Aşağıdaki teknikler kümesi, hata ayıklayıcının varlığını algılamak için çekirdek nesneleri tanıtıcılarını kullanan denetimleri temsil eder.

4.2.1 OpenProcess / SeDebugPrivilege

Varsayılan olarak, bir işlemin erişim belirtecinde SeDebugPrivilege ayrıcalığı devre dışıdır. Ancak, işlem bir hata ayıklayıcı tarafından yüklendiğinde, SeDebugPrivilege ayrıcalığı etkinleştirilir. Bu hata ayıklayıcılar belirteçlerini söz konusu ayrıcalığı etkinleştirecek şekilde ayarlamaya çalıştıklarından ve hata ayıklanan işlem yüklendiğinde SeDebugPrivilege ayrıcalığı devralındığından durum böyledir.

Bazı paketleyiciler dolaylı olarak "CSRSS.EXE" işlemini açmaya çalışarak işlemin hata ayıklanıp ayıklanmadığını belirlemek için SeDebugPrivilege kullanır. Bir işlem CSRSS.EXE işlemini açabiliyorsa; işlemin erişim belirtecinde etkin SeDebugPrivilege ayrıcalığına sahip olduğu ve böylece işlemin hata ayıklandığını öne sürdüğü anlamına gelir. Bu denetim, CSRSS.EXE işleminin güvenlik tanımlayıcısının yalnızca sistemin söz konusu işleme erişmesine izin vermesi nedeniyle çalışır, ancak bir işlem SeDebugPrivilege ayrıcalığına sahipse; güvenlik tanımlayıcısından bağımsız olarak başka bir işleme erişebilir. Bu ayrıcalık varsayılan olarak yalnızca yönetici (Administrator) grubunun üyelerine verilir.

```
;CSRSS.EXE PID
call    [CsrGetProcessId]

;CSRSS.EXE erişim
push    eax
push    FALSE
push    PROCESS_QUERY_INFORMATION
call    [OpenProcess]

;eğer OpenProcess() başarılıysa muhtemelen hata ayıklanıyor
test        eax,eax
jnz .debugger found
```

Şekil 4.2. OpenProcess kullanımı

Şekil 4.2.'de, CSRSS.EXE'nin PID'sini almak için ntdll!CsrGetProcessId()'i kullanır. (Paketleyiciler, işlem numaralandırma yoluyla CSRSS.EXE'nin PID'sini manuel olarak alabilir.) OpenProcess() başarılı olursa, SeDebugPrivilege ayrıcalığının etkinleştirildiği anlamına gelir; bu da işlemin muhtemelen hata ayıklandığı anlamına gelir.

Bu tekniği aşmak için, ntdll!NtOpenProcess() işlevinin geri döndüğü kod bloğuna bir kesme noktası ayarlanır. Kesme noktasına ulaşıldığında, geçerli PID CSRSS.EXE ise EAX değeri 0xC0000022 (STATUS_ACCESS_DENIED) olarak ayarlanır.

4.2.2 CloseHandle

Bir işlem bir hata ayıklayıcı altında çalışıyorsa ve ntdll!NtClose() veya kernel32!CloseHandle() işlevine geçersiz bir tanıtıcı geçirilirse, EXCEPTION_INVALID_HANDLE (0xC0000008) özel durumu ortaya çıkar ve özel durum işleyicisi tarafından önbelleğe alınır. Denetimin, özel durum işleyicisine geçirilmesi, bir hata ayıklayıcının mevcut olduğunu göstermektedir.

4.2.3 LoadLibrary / CreateFile

kernel32!LoadLibrary işlevini kullanarak belleği işlemek için bir dosya yüklendiğinde, LOAD_DLL_DEBUG_EVENT olayı meydana gelir. Yüklenen dosyanın tanıtıcısı LOAD_DLL_DEBUG_INFO yapısında saklanacaktır. Bu nedenle, hata ayıklayıcılar bu dosyadan hata ayıklama bilgilerini okuyabilir. Bu tanıtıcı, hata ayıklayıcı tarafından kapatılmazsa, dosya özel erişimle açılmaz.

Hata ayıklayıcının varlığını kontrol etmek için kernel32!LoadLibrary işlevi kullanılarak dosya yüklenir ve kernel32!CreateFile işlevi kullanılarak dosya açılmaya çalışılır. kernel32!CreateFile çağrısı başarısız olursa, hata ayıklayıcının mevcut olduğunu gösterir.

4.2.4 NtQuerryObject

Bir uygulamada hata ayıklandığında, çekirdekte hata ayıklama oturumu için DebugObject türünde bir nesne oluşturulur. DebugObject sayısı, ntdll!NtQueryObject() işlevi ile tüm nesne türleri hakkında bilgi sorgulanarak elde edilebilir. NtQueryObject 5 parametre kabul eder ve tüm nesne türlerini sorgulamak amacıyla ObjectHandle parametresi NULL olarak ve ObjectInformationClass ise ObjectAllTypeInformation (3) olarak ayarlanır. Söz konusu işlev, NumberOfObjectsTypes alanının ObjectTypeInformation dizisindeki toplam nesne türlerinin sayısı olduğu bir OBJECT_ALL_INFORMATION yapısı döndürür. Algılama rutini ObjectTypeInformation dizisi aracılığıyla yinelenir. TypeName alanı daha sonra UNICODE dizesi "DebugObject" ile karşılaştırılır ve ardından TotalNumberOfObjects veya TotalNumberOfHandles alanında sıfır olmayan bir değer kontrol edilir. Bu tekniği aşmak için NtQueryObject() öğesinin döndüğü kod bloğuna bir kesme noktası ayarlanır. Ardından, döndürülen OBJECT_ALL_INFORMATION yapısı yamalanır. Özellikle paketleyicilerin ObjectTypeInformation dizisi aracılığıyla yinelenmesini önlemek için NumberOfbjectsTypes alanı 0'a ayarlanabilir.

4.3 Zaman Temelli Yöntemler

Hata ayıklamanın başka bir sınıfı da zamanlamaya dayalı algılamalardır. Bu yöntemler, satırlar veya kod bölümleri arasındaki yürütmedeki gecikmeyi algılamak için zamanlamaya dayalı işlevleri kullanır. Bir hata ayıklayıcı içinde kod çalıştırırken, kodu tek adımda yürütmek çok yaygındır. Tek adım, hata ayıklayıcının tek bir satır (step into) veya tek işlev (step over) yürütmesine ve ardından denetimi hata ayıklayıcıya geri döndürmesine izin verir. Hata ayıklayıcı, kodu tek adımda yürütürken, yürütmedeki gecikme, uygulamaya zaman veya tik sayıları döndüren işlevler kullanılarak algılanabilir. Art arda iki zamanlı fonksiyon çağrısı yapılabilir ve farkları(delta) tipik bir değerle karşılaştırılabilir. Alternatif olarak, bir bölüm veya kod bloğu, zamanlayıcı çağrıları ve yine tipik bir yürütme süresi değeriyle karşılaştırılan delta ile çevrelenebilir[9].

4.3.1 RDTSC (read time-stamp counter)

Zaman damgası sayacı, Intel Pentium'un yaratılmasından bu yana tüm x86 işlemcilerin bir parçası olan 64 bitlik bir kayıttır. Bu kayıt, sistemin en son yeniden başlatılmasından bu yana geçen saat döngüsünü yani işlemci zaman damgasını içerir. Bu değere C kodundan erişmek için *__rdtsc* yönergesi kullanılır. Tek adımlı hata ayıklamayı algılamak için *__rdtsc*'ye yapılan iki çağrının sonuçları alınır ve delta, sabit bir değer ile karşılaştırılır.

Şekil 4.3.'te *__rdtsc* yönergesinin, hata ayıklamayı önleme kapsamında kullanımı görülmektedir ve gecikme eşiği için 0x2000 değeri kullanılmaktadır.

```
i = __rdtsc();
j = __rdtsc();

if (j-i < 0x2000) {
 MessageBox(NULL, L"Hata Ayklayıcı Algılanmadı", L"No Debugger", MB_OK);
} else {
 MessageBox(NULL, L"Hata Ayıklayıcı Algılandı !", L"Debugger", MB_OK);
}
```

Şekil 4.3. RDTSC kullanımı

4.3.2 QueryPerormanceCounter

Modern işlemciler ayrıca donanım performans sayaçlarını da içerir. Bu performans sayaçları, işlemci içinde donanımla ilgili etkinliklerin sayılarını depolayan kayıtlardır. Donanım performansının değeri, QueryPerformanceCounter işlevi kullanılarak sorgulanabilir. Diğer zamanlama yöntemleriyle neredeyse aynı teknik kullanılır. QueryPerformanceCounter'a yapılan iki çağrının deltası hesaplanır ve sabit bir değer ile karşılaştırılır. Şekil 4.4.'te QueryPerformanceCounter işlevinin, hata ayıklamayı önleme kapsamında kullanımı görülmektedir ve gecikme eşiği için 0xFF değeri kullanılmaktadır.

```
QueryPerformanceCounter(&li);
QueryPerformanceCounter(&li2);
if ((li2.QuadPart-li.QuadPart) > 0xFF)
{
 MessageBox(NULL, L"Hata Ayıklayıcı Algılandı !", L"Debugger", MB_OK);
}
else
{
 MessageBox(NULL, L"Hata Ayıklayıcı Algılanmadı", L"No Debugger", MB_OK);
}
```

Şekil 4.4. QueryPerformanceCounter kullanımı

4.3.3 GetTickCount

kernel32.dll tarafından sağlanan GetTickCount işlevi, sistemin en son yeniden başlatılmasından bu yana geçen zaman değerini milisaniye cinsinden döndürür. Bu değer 49.7 günde tamamlanır[10]. Bu ve diğer zamanlama yöntemleri arasındaki temel fark, dönüş değerinin milisaniye cinsinden olması nedeniyle eşik değerin çok daha düşük olmasıdır.

4.3.4 timeGetTime

timeGetTime işlevi, son sistem yeniden başlatılmasından bu yana geçen sistem saatini döndürür. GetTickCount gibi, timeGetTime işlev çağrısı da zaman değerini milisaniye cinsinden döndürür.

4.4 İşlem Belleği

Bu yöntemler işlemin, hata ayıklayıcının varlığını algılamak veya hata ayıklayıcıya müdahale etmek için kendi belleğini incelemesi üzerinedir.

4.4.1 ReadFile

Yöntem, dönüş adresindeki kodu yamalamak için kernel32!ReadFile() işlevini kullanır. Buradaki fikir, mevcut işlemin yürütülebilir dosyasını okumak ve dönüş adresini çıktı arabelleği olarak kernel32!ReadFile() işlevine iletmektir. Dönüş adresindeki bayt, 'M' karakteriyle (PE dosyasının ilk baytı) yamalanacak ve işlem muhtemelen çökecektir.

4.4.2 Kod checksum hesaplaması ile yama tespit etme

Yama tespit işlemi, paketleyici kodunun bir bölümünün değiştirilip değiştirilmediğini belirlemeye çalışır; bu, hata ayıklama önleme yordamlarının devre dışı bırakılmış olabileceğini ve ikinci bir amaç olarak yazılım kesme noktalarının ayarlanıp ayarlanmadığını belirleyebilir. Yama tespiti kod checksum ile gerçekleştirilir ve checksum hesaplaması basitten karmaşık algoritmalarına kadar değişebilir. Yazılım kesme noktaları bir kod checksum yordamı tarafından tanımlanıyorsa, bunun yerine donanım kesme noktaları kullanılabilir. Yama işlemi checksum yordamı tarafından tanımlanıyorsa, tersine mühendis, yamalanmış adreste bir erişim kesme noktası ayarlayarak checksum yordamının nerede olduğunu belirleyebilir ve checksum yordamı bulunduğunda, checksum değerini beklenen değer ile değiştirebilir veya karşılaştırmadan sonra uygun bayrakları değiştirerek bu tekniği aşabilir.

4.5 Hata Ayıklayıcı ile Etkileşim

Bu yöntemler, çalışan işlemin bir kullanıcı arabirimini yönetmesine veya hata ayıklanmış bir işlemdeki tutarsızlıkları keşfetmek için üst işlemiyle etkileşime geçmesine izin verir.

4.5.1 NtSetInformationThread / ThreadHideFromDebugger

Bu yöntem, genellikle bir iş parçacığının önceliğini ayarlamak için kullanılan ntdll!NtSetInformationThread() API'sini kullanır. Ancak söz konusu API, hata ayıklama olaylarının hata ayıklayıcıya gönderilmesini önlemek için de kullanılabilir.

NtSetInformationThread() parametreleri Şekil 4.5.'te gösterilmektedir. Bu yöntemi gerçekleştirmek için, ThreadInformationClass parametresi olarak ThreadHideFromDebugger (0x11) geçirilir, ThreadHandle genellikle geçerli iş parçacığı tanıtıcısına (0xfffffffe) ayarlanır.

```
C++

__kernel_entry NTSYSCALLAPI NTSTATUS NtSetInformationThread(
  [in] HANDLE          ThreadHandle,
  [in] THREADINFOCLASS ThreadInformationClass,
  [in] PVOID           ThreadInformation,
  [in] ULONG           ThreadInformationLength
);
```

Şekil 4.5. NtSetInformationThread() parametreleri

Dahili olarak, ThreadHideFromDebugger, ETHREAD çekirdek yapısının HideThreadFromDebugger alanını ayarlayacaktır. Bir kez ayarlandığında, asıl amacı hata ayıklayıcıya olaylar göndermek olan dahili çekirdek işlevi DbgkpSendApiMessage() hiçbir zaman çağrılmaz. Bu tekniği aşmak için ntdll!NtSetInformationThread() içinde bir kesme noktası ayarlanabilir ve kesme noktasına isabet edildiğinde, fonksiyonun çekirdeğe ulaşmasını önlemek için EIP değiştirilebilir.

4.5.2 BlockInput

Tersine mühendisin hata ayıklayıcıyı denetlemesini önlemek için, paketleyiciler, ana paket açma rutini yürütülürken klavye ve fare girişini engellemek için user32!BlockInput() fonksiyonunu kullanabilir.

Tipik bir örnek olarak, GetProcAddress() içinde bir kesme noktası ayarlayan ve ardından kesme noktasına ulaşılana kadar paketten çıkarma(unpacking) kodunu yürüten bir tersine mühendis, BlockInput fonksiyonunun çağrılması sonrasında hata ayıklayıcıyı kontrol edemeyecek hale gelecektir. Bu hata ayıklamayı önleme yönteminin basit çözümü,

BlockInput fonksiyonunun sadece RETN kodunu işletecek şekilde yamalanması olarak gösterilebilir.

4.6 Anti-Dumping

Tersine mühendislik alanında kullanılan özel bir terim olan "Dumping", korunan bir yürütülebilir dosyanın alınması ve yürütülebilir dosyanın şifresi çözüldükten sonra, programın esasen anlık görüntüsünün alınması ve diske kaydedilmesi sürecini tanımlar.

4.6.1 Nanomitler (nanomites)

Bu teknik, belirli atlama yönergelerini “int 3h” kesme noktaları ile değiştirerek çalışır. Değiştirilen atlama yönergeleri, şifrelenmiş bir tabloya yerleştirilir. Bu tablo, adres, tür (JCC, JMP, JLE vb.), uzaklık, nanomit veya normal hata ayıklama özel durumu gibi verileri içerir. Daha sonra, bu kesme noktalarından birine isabet edildiğinde, hata ayıklama işlemi, hata ayıklama özel durumunu işler ve hata ayıklama sonu hakkında belirli bilgileri arar. Bu bilgiler, kesme noktasının bir nanomit mi yoksa gerçek bir hata ayıklama kesme noktası mı olduğu ve EFLAGS kayıtlarının atlama türüne uygun şekilde karşılaştırılması sonucunda atlamanın yapılıp yapılmayacağıdır. Bu teknik doğru yerlerde kullanıldığında çok güçlüdür ve performans üzerinde neredeyse hiç negatif etkisi yoktur.

4.6.2 Stolen bytes (stolen codes)

Bu teknikte paketleyici tarafından korunan orijinal işlemdeki kod veya bayt genellikle OEP'den kaldırılır ve paketlenmiş kodun içinde bir yerde şifrelenir. Baytların bulunduğu alan daha sonra orijinal koddan "çalınan" şifresi çözülmüş baytları içeren dinamik olarak ayrılmış bir arabelleğe atlayacak kodla değiştirilir; Bu arabellek ayrıca uygun yürütme adresine geri atlama içerir. Çoğu zaman, hem baytların kaldırıldığı alan hem de orijinal baytların bulunduğu dinamik olarak ayrılmış arabellek, önemsiz kod ve hatta daha fazla tersine mühendisliği önleme tekniği ile doldurulur.

4.6.3 Kendi kendine eşlemeyi kaldırma (self-unmapping)

Yürütülebilir dosya yürütülmek üzere yüklendiğinde, bölümlerdeki tüm veriler adres alanına eşlenir. Buna, dosyanın eşlenmiş görüntüsü denir. Bu eşlenen dosya görünümünün eşlemesi, UnmapViewofFile() işlevi kullanılarak kaldırılabilir. Ancak yüklenen yürütülebilir dosyanın eşlemesini kaldırmadan önce, tüm verilerin ayrı bir konuma

aktarılması gerekmektedir. Çünkü dosya eşleştirildikten sonra, görüntüdeki çeşitli bölümler tarafından işgal edilen adres aralıkları geçersiz hale gelir. Görüntü yeniden yerleştirildikten sonra, tüm mutlak referansların yeni BaseAdress'e göre ayarlanması gerekmektedir. Bu işlem yer değiştirme tablosu (relocation table) kullanılarak yapılır. Tüm yer değiştirmeler düzeltildikten sonra görüntünün önceki görünümü açılabilir. Bu yöntem, VirtualAlloc() ve bunu gibi çağrılarla dinamik olarak oluşturulan belleği değil, yalnızca bölümler ve üstbilgiler gibi statik bellek dökümü çıkartan otomatik araçlar kullanılıyorsa, bellek dökümünü önlemede etkilidir.

4.6.4 Sanallaştırma

Sanallaştırma, tersine mühendisliği önleme yöntemlerinin geleceği olarak kabul edilir ve günümüzde kullanımı daha yaygın hale gelmiştir. Örneğin OREANS firması tarafından satışı yapılan Themida koruyucusu, sanal makine korumasını kullanan her korumalı yürütülebilir dosya için benzersiz bir sanal makine oluşturan bir teknoloji kullanır. Themida, sanal makineleri bu şekilde uygulayarak sanallaştırma korumasına karşı geliştirilebilecek genel bir saldırının ortaya çıkmasını önler.

4.6.5 PE başlığını kaldırma

Yürütülebilir dosyanın PE başlığını çalışma zamanında bellekten kaldıran basit bir anti-dumping tekniğidir. Bu teknik ile dump edilmiş bir görüntüde, önemli tabloların (Reloc, İçe Aktarma, Dışa Aktarma vb.) RVA'sı, giriş noktası ve Windows yükleyicisinin bir dosya yüklerken kullanması gereken önemli bilgileri eksik olacaktır.

4.6.6 Guard pages

Paketleyiciler ve koruyucular, şifre çözme/açma sistemi uygulamak için guard pages kullanabilir. Yürütülebilir dosya, çalışma zamanında dosyanın tüm içeriğinin sıkıştırmasını açmak/şifresini çözmek yerine belleğe yüklendiğinde, koruyucu, hemen ihtiyaç duyulmayan tüm sayfaları guard pages olarak işaretleyecektir. Bu yapıldıktan sonra, başka bir kod veya veri bölümüne ihtiyaç duyulduğunda, bir EXCEPTION_GUARD_PAGE (0x80000001) özel durumu ortaya çıkar ve verilerin dosyadan veya bellekteki şifrelenmiş / sıkıştırılmış içeriklerden şifresi çözülebilir.

4.7 Obfuscation

Yazılım geliştirmede obfuscation, insanların anlaması zor olan kaynak veya makine kodu oluşturma eylemidir. Yazılım mühendisleri, tersine mühendisliği ve kurcalamayı önlemek için düşük maliyetli bir yaklaşım olan obfuscation yöntemlerine başvurur[11-12].

4.7.1 Kontrol akışı gizleme (control flow obfuscation)

Kontrol Akışı Gizleme, metotların içindeki kodu spagetti koduna dönüştürür, bu da kodun işlevini korurken insan gözünün ve kod çözücülerin program mantığını izlemesini son derece zorlaştırır.

4.7.2 Opak yüklemler (opaque predicates)

Akademik literatürde, opak bir yüklem, her zaman bir yönde çalışan, programın yaratıcısı tarafından bilinen ve analizör tarafından önceden bilinmeyen bir dal olarak adlandırılır[13].

Normal bir koşullu atlamada, kodun devam edebileceği iki yol vardır ve ayrıştırıcı her iki koşul için de kodu çözmesi gerekir. Opak bir yüklemde, gerçekten tek bir yol olsa bile, ayrıştırıcıya kodun devam edebileceği iki yol olduğu düşündürülür. Teknik, bu koşulsuz atlamalardan birini kurmaktır ve atlamanın yapılmayacağı yola gereksiz ve kafa karıştırıcı kodları, atlamanın her zaman yapılacağı yola ise gerçek kodları eklemektir. Opak isimli Şekil 4.6.'daki alt programda görüldüğü gibi EAX akümülatörüne 4 değeri atanmış ve cmp komutu kullanılarak 5 ile karşılaştırılmıştır. Karşılaştırmanın sonucu hiçbir zaman eşitlik koşulunu sağlamayacağından je (Jump Equal) komutu ile "sahte" isimli etikete atlama işlemi gerçekleşmeyecek ve programın akışı, jne (Jump Not Equal) komutu ile her zaman "gerçek" isimli etiketten devam edecektir.

```
opak proc

mov eax,4
cmp eax,5
je sahte
jne gercek

sahte :
decoy db "1234-ABCD-DEFG-5678";

gercek :
mov eax 3
ret

opak endp
```

Şekil 4.6. opak isimli alt program

Opak yüklemlerin güzel yanı, kodu olduğundan çok daha karmaşık hale getirmesi ve tersine mühendisin kodu anlamak için çok daha fazla zaman harcamasına neden olmasıdır.

4.7.3 Talimat permutasyonu (instruction permutation)

Talimat permutasyonu da bir obfuscation yöntemidir ve basit talimatları, sonunda aynı sonucu veren bir dizi başka talimatla değiştirerek, bir kod parçasını analiz etmek için gereken süreyi artırmayı amaçlar. Şekil 4.7.'de "add eax, ebx" talimatının karşılığı gösterilmektedir.

```
;add eax, ebx
push eax
add [esp], ebx
pop eax
```

Şekil 4.7. add eax, ebx talimatının karşılığı

4.7.4 Şifreleme

Genellikle paketleyicilerin kullandığı bu teknik, hem koruyucu kodu hem de korunan yürütülebilir dosyayı şifrelemek üzerinedir. Şifreleme algoritması, çok basit XOR döngülerinden çok karmaşık hesaplamalar yapan döngülere kadar değişen bir skalaya sahiptir ve paketleyiciler arasında bu algoritmalar büyük farklılıklar gösterir. Polimorfik paketleyicilerde, şifreleme algoritması da oluşturulan örnekler arasında değişiklik gösterir ve şifre çözme kodu, oluşturulan her örnekte çok farklı görünecek şekilde değiştirilir ve paketleyici tanımlayıcı aracının (Packer Identifier Tool) paketleyiciyi doğru bir şekilde tanımlamasını engelleyebilir.

Şifre çözme rutinleri, veri alma, hesaplama ve depolama işlemlerini gerçekleştiren döngüler olarak kolayca tanınır. Şekil 4.8.'de, şifrelenmiş bir DWORD değeri üzerinde birkaç XOR işlemi gerçekleştiren basit bir şifre çözme rutini örneği verilmiştir.

```
0040A07C LODS DWORD PTR DS:[ESI]
0040A07D XOR EAX,EBX
0040A07F SUB EAX,12338CC3
0040A084 ROL EAX,10
0040A087 XOR EAX,799F82D0
0040A08C STOS DWORD PTR ES:[EDI]
0040A08D INC EBX
0040A08E LOOPD SHORT 0040A07C ;Şifre Çözme rutini
```

Şekil 4.8. Şifre çözme rutini

4.8 TLS Callback

TLS Callback, var olan en ilginç hata ayıklamayı önleme yöntemlerinden biridir, çünkü yıllardan beri bilinmesine rağmen bu yöntemi kullanmak için hala yeni yollar keşfedilmektedir. Thread Local Storage (TLS), iş parçacığı çalıştırılmadan önce iş parçacığına özgü verileri başlatmak için vardır. Her işlem en az bir iş parçacığı içerdiğinden, TLS, ana iş parçacığı çalıştırılmadan önce verileri başlatma özelliğini içerir. Başlatma, dinamik olarak ayrılan belleğe kopyalanan statik bir arabellek belirtilerek ve / veya bellek içeriğini dinamik olarak başlatmak için bir "callback" dizisindeki kod yürütülmesiyle yapılabilir. Hata ayıklayıcının varlığını tespit etmek için iyi bir yöntemdir çünkü "callback" işlevi, yürütülebilir dosyanın giriş noktası çağrılmadan önce çağrılır.

4.9 Mutasyon

Anti-virüs teknolojisi ilerledikçe, kötü amaçlı yazılımları gizleme işi daha zor hale geldi. Sonuç olarak, kötü amaçlı yazılım geliştiricileri, algılamayı önlemek için mutasyon teknikleri geliştirmiştir. Şifrelenmiş virüsler, koda sahte talimatlar eklemek için değişken şifre çözme şemaları kullanır[14]. Bu talimatlar, önceden bilinen bir örneğin yeni görünmesini sağlar. Oligomorfik teknikler, virüse ek şifre çözücüler ekleyerek şifrelenmiş virüslerin karmaşıklığını artırır. Bir oligomorfik virüs, olası varyasyonlara ek olarak çalışma zamanında mevcut şifre çözücülerden rastgele seçecektir. Polimorfik virüsler ise, milyonlarca benzersiz virüs örneği oluşturabilen mutasyona uğramış şifre çözücüleri içerir. Metamorfik virüsler ise yürütülebilir dosyanın içindeki modülleri değiştirerek yeni formlar oluşturur. Metamorfik virüsler N faktöriyel (n!) yaratma yeteneğine sahiptir (n alt rutin için).

4.10 İşlem Enjeksiyonu

İşlem enjeksiyonu bazı paketleyicilerde kullanılan bir özelliktir. Bu özellikle, paket açma saplaması seçilen bir ana bilgisayar sürecini (örneğin: kendisi, explorer.exe, iexplorer.exe, vb.) ortaya çıkarır ve ardından paketten çıkartılmış yürütülebilir dosyayı farklı bir bilgisayar işlemine enjekte eder. Zararlı yazılımlar, harici ağ bağlantıları gerçekleştirmek ve işlemin izin verilen uygulamalar listesinde olup olmadığını kontrol eden bazı güvenlik duvarlarını atlamalarına izin vermek için bu paketleyici özelliğini kullanır.

Paketleyicilerin işlem enjeksiyonu gerçekleştirmek için kullandığı yöntemlerden biri aşağıdaki adımlardan oluşmaktadır.

- Host işlemi askıya alınmış bir alt işlem olarak oluşturulur. Bu, kernel32!CreateProcess()'e iletilen CREATE_SUSPENDED işlem oluşturma

bayrağı kullanılarak yapılır. Bu noktada bir başlatma iş parçacığı oluşturulur ve askıya alınır, yükleyici yordamı (ntdll!LrdInitializeThunk) hala çağrılmadığı için DLL'ler hala yüklenmez.

- kernel32!GetThreadContext() kullanılarak, alt işlemin başlatma iş parçacığının durumu alınır.
- Alt işlemin PEB adresi CONTEXT.EBX aracılığıyla alınır.
- Alt işlemin ImageBase'i, PEB.ImageBase (PEB + 0x8) okunarak alınır.
- Alt işlemdeki orijinal host görüntüsünün eşlemesi ntdll!NtUnmapViewOfSection() kullanılarak kaldırılır.
- Paket açma stub'ı, paketlenmemiş yürütülebilir dosyanın görüntü boyutuna eşit dwSize parametresi ile kernel32!VirtualAllocEx() kullanarak alt işlem içinde bellek tahsis eder.
- kernel32!WriteProcessMemory() kullanılarak, PE başlığı ve paketlenmemiş yürütülebilir dosyanın bölümlerinin her biri alt işleme yazılır.
- Alt işlemin PEB.ImageBase'i daha sonra paketlenmemiş yürütülebilir dosyanın ImageBase'i eşleşecek şekilde güncellenir.
- Alt işlemin başlatma iş parçacığının durumu daha sonra, kernel32!SetThreadContext() aracılığıyla güncellenir.
- Alt işlemin yürütülmesine kernel32!ResumeThread() aracılığıyla devam edilir.

Oluşturulan alt işlemin giriş noktasından itibaren hatalarını ayıklamak için, tersine mühendis WriteProcessMemory() içinde bir kesme noktası ayarlayabilir ve giriş noktasını içeren bölüm alt işleme yazılmak üzereyken, giriş noktası koduna "kendi kendine atlama" talimatı (0xEB 0xFE) eklenir. Alt işlemin ana iş parçacığı devam ettirildiğinde, alt işlem giriş noktasında sonsuz bir döngüye girer. Bu noktada tersine mühendis, alt işleme bir hata ayıklayıcı ekleyebilir, değiştirilen talimatları geri yükleyebilir ve normal bir şekilde hata ayıklamaya devam edebilir.

5. YÜRÜTÜLEBİLİR PAKETLEYİCİLER VE KORUYUCULAR (EXECUTABLE PACKERS AND PROTECTORS)

Başlangıçta, yürütülebilir bir dosyanın boyutunu 1.44 MB'lık bir diskete sığacak şekilde küçültmek için kullanılan yürütülebilir paketleyiciler veya sıkıştırıcılar (executable packers - compressors), günümüzde kötü amaçlı yazılım üreten bilgisayar korsaları için kod gizlemenin önemli bir parçası haline geldi. Tipik bir paketleyici, hedef dosyanın kod ve veri bölümlerini sıkıştırır ve giriş noktasını bir açıcı (decompressor) ile değiştirir. Dosya yürütüldüğünde, orijinal dosyayı belleğe açan ve ardından dosyanın orijinal giriş noktasına (OEP) atlayan açıcı çalışır. OEP'ye ulaşıldığında, ikili normal olarak yürütülmeye başlar. Tersine mühendis, paketlenmiş bir yürütülebilir dosya ile karşı karşıya kaldığında, içinde bulunan gerçek yürütülebilir dosyayı etkili bir şekilde analiz etmesi için öncelikle paketleyiciden kurtulmalıdır.

Koruyucu, hata giderme, kod şifreleme, kod gizleme, kod sanallaştırma vb. gibi çeşitli tersine mühendislik tekniklerini kullanarak paketleme yapan programlara denir. Themida ve VMProtect Microsoft Windows ortamları için yaygın olarak kullanılan ticari koruyuculardan bazılarıdır[15].

5.1 Paketleyici Algılama Yöntemleri

Paketleyici algılama yöntemleri[16, 17, 18] paketlenmiş ve paketlenmemiş yürütülebilir dosyaları ayırt etmek ve kullanılan paketleyicilerin türünü bulmak için tasarlanmıştır. Bazı açıcılar, paketleyici algılama yeteneğine sahiptir. Paketlenmiş ve paketlenmemiş yürütülebilir dosyalar arasında ayrım yapma sorununun kararsız olduğu gösterilmiştir[19], bu nedenle paketleyici tespiti yapmak için bazı buluşsal politikalar önerilmiştir.

5.1.1 İmza tabanlı paketleyici algılama

Tıpkı kötü amaçlı yazılımın imza tabanlı bir anti-virüs motoru kullanılarak tespit edilebilmesi gibi, paketten çıkarma işlemini gerçekleştirecek stub'ın bayt dizilerinin (imza) karakteristiği, paketleyicileri tespit etmek için kullanılabilir. İmza tabanlı paketleyici algılama yöntemi basit ve etkilidir, ancak imzası olmayan bir paketleyiciyi algılayamaz. PEiD, DIE, eklenti desteğine sahip en popüler imza tabanlı paketleyici algılama araçlarıdır.

5.1.2 Bilgi yoğunluğu (entropi) eşikleri

Entropi veya bilgi yoğunluğu, bir dizi sembol içindeki her sembolün taşıdığı bilgi miktarını tanımlayan bilgi teorisinden bir terimdir. Bir çalışma zamanı paketleme yöntemi, yeni bayt dizisinin tipik olarak orijinalinden daha yüksek entropiye sahip olduğu bir bayt dizisini diğerine dönüştürür. Bu özellik, paketlenmiş yürütülebilir dosyaları paketlenmemiş olanlardan ayırt etmek için kullanılabilir. Bu yaklaşım, PE dosyasını bayt dizileri olarak temsil eder ve bir sembol olarak bireysel baytı kullanır[20]. Bir X sembolünün entropisi log(1/px) olarak tanımlanır, burada *px* X'in frekansıdır. Bir PE dosyasının entropisi ne kadar yüksek olursa, paketlenmiş olması o kadar olasıdır. Bazı eşikler, belirli bir paketlenmiş ikili veriye ait veritabanından statik olarak çıkarılır. Algılanan bir PE dosyasının entropisinin ilgili eşiklerin üzerinde olduğu tespit edilirse, paketlenmiş olarak sınıflandırılacaktır. Entropi eşiği yöntemi, paket açıcıların çoğunda yaygın olarak kullanılır.

5.1.3 Şablon tanıma

Şablon tanıma, tipik bir istatistik ve denetimli öğrenme yöntemidir. İlk olarak, yazılabilir-yürütülebilir bölümlerin sayısı, kod ve veri entropisi vb. gibi yürütülebilir ikili dosyanın bazı yapı özellikleri ikili statik analiz teknikleri kullanılarak çıkarılır ve bir model vektörüne çevrilir. Ardından, paketlenmiş ve paketlenmemiş yürütülebilir dosyaları ayırt etmek için örüntü tanıma teknikleri uygulanır. Örüntü tanıma yöntemi, paketlenmiş ve paketlenmemiş yürütülebilir dosyalar arasında ayrım yapmak için paketlenmiş yürütülebilir dosyalardan oluşan bir veritabanında eğitilmiş bir sınıflandırıcı kullanır, bu nedenle sonucu hatalı bulma olasılığı vardır.

5.2 Paketten Çıkarma (Unpacking) ve Orijinal Giriş Noktasına (OEP) Ulaşma Yöntemleri

Her paketleyicinin OEP'ye ulaşmak için farklı yöntemler kullandığı doğru olsa bile, OEP'yi hızlı bir şekilde bulmak için kullanılan bazı yöntemler vardır.

Şifre çözme rutini sona erdikten sonra paketleyici, içe aktarma tablosunu (import table) çözecek ve geri yükleyecektir. Bu nedenle, yordamın sonunda LoadLibraryA (bazı durumlarda GetModuleHandle) ve GetProcAddress işlevleri çağrılır ve bu işlevler üzerinde kesme noktaları ayarlamak oldukça yararlı olabilir. Bu yöntem için normal süreç aşağıdaki gibidir:

- LoadLibraryA'da bir kesme noktası ayarlanır.
- Program, kesme noktasında duruncaya kadar çalıştırılır.

- Tüm kitaplıklar yüklenene kadar çalıştırılır.
- Kod, belleğin paketlenmiş bölümüne kadar izlenir.
- Kod, son return/jump talimatına kadar izlenir.

LoadLibraryA'da kesme noktaları ayarlamak her zaman mantıklı olmayabilir, çünkü kod akışı her zaman paket açma rutininin sonuna dönmeyebilir. Daha rafine bir sonuç elde etmek için, yüklenen son işlevi aramak gerekir, bu da GetProcAddress üzerinde kesme noktasının ayarlanması anlamına gelmektedir. Tablo geri yüklendiğinden ve belleğin paketlenmemiş bölümleri erişim için hazır olduğundan, program artık bilgisayarın belleğinde çalışacaktır.

5.2.1 ESP trick

GetProcAddress fonksiyonunun kesilmesinin başarısız olduğu durumlarda, tersine mühendisler “ESP Trick” olarak bilinen yöntemden yararlanır. Çoğu paketleyici, tüm kayıtları yığına yerleştirmek için giriş noktasının (EP) yakınında "PUSHAD" talimatını içerir. Bunun nedeni, şifre çözme rutini başladığında, işlem sırasında değerler değişse bile kayıtlar kullanılabilir. Bu nedenle, bir bellek bölümü yazıldığında, kayıt defteri yığını, paketleyici tarafından kullanılan ilk değerleri tutar. ESP kaydı, yığının en üstüne işaret eden dolaylı bir bellek işleneni olarak kullanılır. Bir "PUSHAD" komutu yürütüldüğünde ESP, yığını içeren adrese işaret edecek şekilde değiştirilir.

Şifre çözme işlemi tamamlandıktan sonra, paketleyici bellekteki tüm kayıtları yığından geri çağırır (POPAD) ve uygulama bunları normal çalışma zamanı için kullanabilir. ESP yazmacına bir donanım kesme noktası ayarlanır, yazmaçlar yığından geri yüklendiğinde kod bloğu orijinal giriş noktasından (OEP) devam eder ve böylece tam konum bulunur.

5.2.2 Özel durum (exception) sayımı ve belleğe erişimde kesme noktası

Bazı paketleyiciler, tersine mühendisi ve araçlarını kandırmak için birden fazla istisna türü kullanır. Uygulama gerçekten çalışmadan önce kaç istisna oluştuğunu sayılarak, OEP bulunabilir.

Örneğin, uygulamanın dört istisnadan sonra çalıştığı biliniyorsa, o zaman dört istisna sayıldıktan sonra, yürütülebilir dosyanın ilk bölümü için belleğe erişimde bir kesme noktası ayarlanır. Bu normalde OEP'nin bulunduğu yerdir ve kesme noktası tetiklenene kadar yürütmeye devam edilir.

5.2.3 Yığında geri izleme

Yığın, dinamik analiz sırasında doğru yürütmeyi takip etmek için kullanıldığından, hangi işlevin daha önce çağrıldığı, şu anda içinde bulunulan işlevin hangi adresten çağrıldığı ve bunun gibi birçok şey hakkında ipucu almak için de kullanılır.

5.2.4 Yapılandırılmış özel durum işleme (structured exception handling / SEH) izleme

Birçok paketleyici, atlama veya çağrı talimatı gibi belirgin yöntemler kullanmadan yürütmeyi istedikleri yere yönlendirmek için istisnaları, tersine mühendisliği önleme yöntemi olarak kullanır. Bu nedenle, özel durum yönetimlerini (exception handlers) ve yürütmenin nereye aktarılacağını izlemek, OEP'yi bulmaya yardımcı olabilir.

5.3 Bilinen Giriş Noktaları

Amaç, paketleyici tarafından eklenen kod katmanlarının altında gizlenmiş olan korumalı uygulamadan çalışan bir yürütülebilir dosyayı almaktır. Bunu başarmak için önce uygulama için orijinal giriş noktasının (OEP) bulunması gerekir. Bu, işlem Windows Loader tarafından oluşturulduktan sonra normal olarak yürütülecek olan uygulama içindeki kullanıcı kodunun ilk talimatıdır.

MS Visual C ++ , Borland C ++ Builder, Visual Basic tabanlı uygulamalar ve bunun gibi tanınmış derleyiciler tarafından oluşturulan ortak giriş noktalarını (EP) tanıyabilmek çok yardımcı olacaktır. Bu sayede bellekteki belirli bayt kalıpları aranabilir ve daha da ötesi, belirli bir derleyiciden oluşturulan kod yürütmesinin başlangıcında her zaman çağrılması gerektiği bilinen belirli API'lere yapılan çağrılar izlenebilir.

Şekil 5.1.'de Microsoft Visual C++ 6.0 ile derlenmiş bir programın giriş noktası gösterilmektedir.

```
Microsoft Visual C++ 6.0
PUSH    EBP
MOV     EBP, ESP
PUSH    -1
PUSH    005C7C70
PUSH    005035E0 ; SEH
MOV     EAX, DWORD PTR FS:[0]
PUSH    EAX
MOV     DWORD PTR FS:[0], ESP
SUB     ESP, 58h
PUSH    EBX
PUSH    ESI
PUSH    EDI
MOV     DWORD PTR SS:[EBP-18], ESP
CALL    DWORD PTR DS:[&KERNEL32.GetVersion] ; kernel32.GetVersion—ilk API çağrısı
```

Şekil 5.1. Microsoft Visual C++ 6.0 ile derlenmiş bir programın giriş noktası

Şekil 5.2.'de Microsoft Visual C++ 7.x ile derlenmiş bir programın giriş noktası gösterilmektedir.

```
Microsoft Visual C++ 7.x
004023FD      6A 74       PUSH  74
004023FF      68 A03A4000    PUSH    00403AA0
00402404      E8 F3010000      CALL    004025FC
00402409      33DB      XOR     EBX, EBX
0040240B      895D E0    MOV     DWORD PTR SS:[EBP-20], EBX
0040240E      53            PUSH     EBX     ;  /pModule = > NULL
0040240F      8B3D 3C304000 MOV     EDI, DWORD PTR
DS:[&KERNEL32.GetModuleHandleA] ; kernel32.GetModuleHandleA
00402415      FFD7      CALL     EDI     ; \GetModuleHandleA ←ilk API çağrısı
```

Şekil 5.2. Microsoft Visual C++ 7.x ile derlenmiş bir programın giriş noktası

Şekil 5.3.'te Microsoft Visual C++ 8.0 – 9.0 ile derlenmiş bir programın giriş noktası gösterilmektedir.

```
Microsoft Visual C++ 8.0-9.0
0047DF0E  E8  F7730000      CALL    b.00485330A ; ← çağrılan işlevde çağrılacak ilk API GetSystemTimeAsFileTime
0047DF13  E9  17FEFFFF      JMP       b.0047DD2F ; ←Bu atlama aşağıdaki gibi görünen bir kod bloğuna götürür
0047DD2F   6A 60       PUSH     60
0047DD31    68  787B4E00       PUSH     004E7B78
0047DD36    E8  B9400000       CALL     00481DF4
0047DD3B   8365 FC 00     AND    DWORD PTR SS:[EBP-4], 0
0047DD3F    8D45 90     LEA     EAX, DWORD PTR SS:[EBP-70]
0047DD42    50         PUSH     EAX     ;/pStartupinfo
0047DD43      FF15 78824B00 CALL     DWORD  PTR
DS:[&KERNEL32.GetStartupInfoW] ; \GetStartupInfoW ←ikinci API çağrısı
```

Şekil 5.3. Microsoft Visual C++ 8.0 – 9.0 ile derlenmiş bir programın giriş noktası

Şekil 5.4.'te Borland C++ Builder ile derlenmiş bir programın giriş noktası gösterilmektedir.

```
Borland C++ Builder (any recent version)
0040154C    EB 10       JMP    SHORT b.0040155E
0040154E    66           DB     66             ; CHAR 'f'
0040154F    62           DB     62             ; CHAR 'b'
00401550    3A          DB     3A            ; CHAR ':'
00401551    43           DB     43             ; CHAR 'C'
00401552    2B           DB    2B            ; CHAR '+'
00401553    2B           DB    2B            ; CHAR '+'
00401554    48           DB     48             ; CHAR 'H'
00401555    4F           DB     4F             ; CHAR 'O'
00401556    4F           DB     4F             ; CHAR 'O'
00401557    4B           DB     4B            ; CHAR 'K'
00401558    90           NOP
00401559    E9           DB     E9
0040155A   AC404F00     DD     OFFSET    CPPdebugHook
0040155E   A1 9F404F00    MOV     EAX, DWORD PTR DS:[4F409F]
00401563   C1E0 02       SHL     EAX, 2
00401566    A3 A3404F00    MOV     DWORD    PTR DS:[4F40A3], EAX
0040156B    52          PUSH     EDX
0040156C    6A 00       PUSH      0         ; /pModule = NULL
0040156E    E8 BD110F00     CALL    [JMP.&KERNEL32.GetModuleHandleA] ; \GetModuleHandleA ←ilk API çağrısı
```

Şekil 5.4. Borland C++ Builder ile derlenmiş bir programın giriş noktası

Şekil 5.5.'te Dev C++ ile derlenmiş bir programın giriş noktası gösterilmektedir.

```
Dev-C++ 4.9.9.2
00401220      55       PUSH     EBP
00401221      89E5      MOV     EBP, ESP
00401223      83EC 08      SUB    ESP, 8
00401226      C70424 01000000 MOV    DWORD PTR SS:[ESP], 1
0040122D     FF15 D0504000   CALL  DWORD PTR
DS:[&msvcrt.__set_app];  msvcrt.__set_app_type ← ilk API çağrısı
00401223      E8 C8FEFFFF    CALL   00401100
```

Şekil 5.5. Dev C++ ile derlenmiş bir programın giriş noktası

Şekil 5.6.'da Borland Delphi 6.0 – 7.0 ile derlenmiş bir programın giriş noktası gösterilmektedir.

```
Borland Delphi 6.0-7.0
0046488C      55       PUSH     EBP
0046488D      8BEC      MOV     EBP, ESP
0046488F      83C4 F0     ADD    ESP, -10
00464892      B8 0C474600    MOV     EAX,  0046470C
00464897     E8 341DFAFF   CALL    004065D0; ← çağrılan işlevde çağrılacak ilk API GetModuleHandleA
```

Şekil 5.6. Borland Delphi 6.0 – 7.0 ile derlenmiş bir programın giriş noktası

Şekil 5.7.'de Microsoft Visual Basic 5.0 – 6.0 ile derlenmiş bir programın giriş noktası gösterilmektedir.

```
Microsoft Visual Basic 5.0-6.0
00401258    FF25 7410400  JMP   DWORD PTR
DS:[&MSVBVM60.#100>] ;  MSVBVM60.ThunRTMain
0040125E      00         DB     00
0040125F      00         DB     00
00401260      68 289F4C00    PUSH     004C9F28←entry point
00401265      E8 EEFFFFFF   CALL    [JMP.&MSVBVM60.#100>] ;← MSVBVM60.ThunRTMain 'e ilk çağrı
```

Şekil 5.7. Microsoft Visual Basic 5.0 – 6.0 ile derlenmiş bir programın giriş noktası

Şekil 5.8.'de MASM32/TASM32 ile derlenmiş bir programın giriş noktası gösterilmektedir.

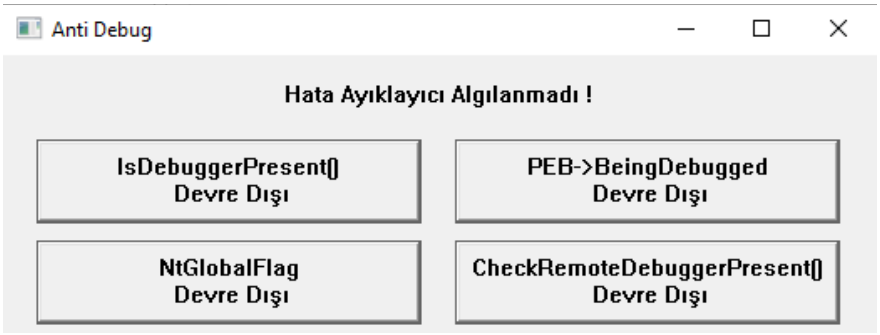
```
MASM32/TASM32
00401000     6A 00      PUSH     0
00401002     E8 D7020000      CALL
[JMP.&KERNEL32.GetModuleHandleA] ;  \GetModuleHandleA ← ilk API çağrısı
00401007      A3 6C30400      MOV    DWORD PTR DS:[40306C], EAX
0040100C     E8  C7020000   CALL
[JMP.&KERNEL32.GetCommandLineA]  ;  GetCommandLineA
```

Şekil 5.8. MASM32/TASM32 ile derlenmiş bir programın giriş noktası

6. YAPILAN ÇALIŞMALAR

6.1 Örnek 1

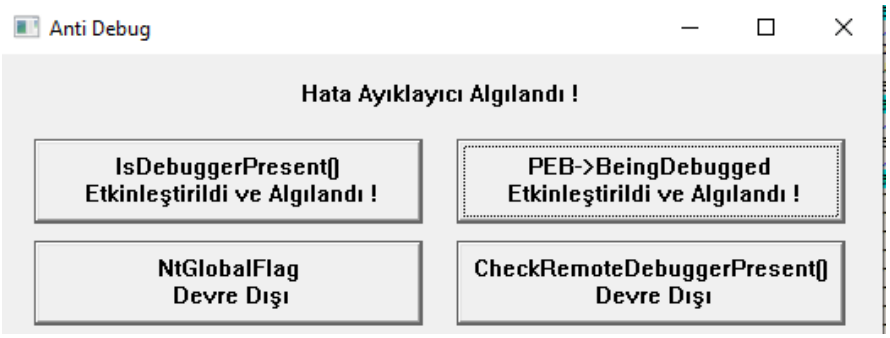
Bu çalışmada “Anti Debug” isimli programda 4 hata ayıklamayı önleme yönteminin aşılması gösterilmiştir. “Anti Debug” isimli program Şekil 6.1.’de gösterildiği gibi 4 butona sahiptir ve her butona tıklandığında üzerinde yazan hata ayıklamayı önleme yöntemini aktive etmektedir.



Şekil 6.1. Anti Debug Arayüzü

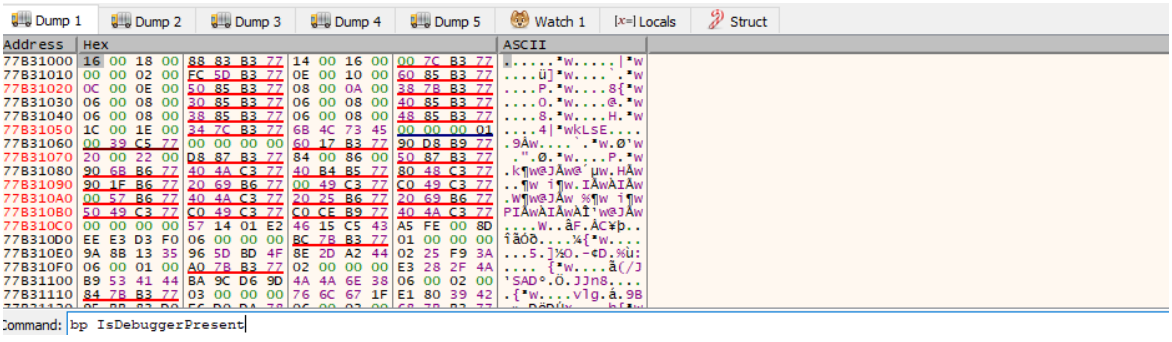
6.1.1 IsDebuggerPresent ve PEB->BeingDebugged yöntemlerini aşma

Program x64dbg isimli hata ayıklayıcı ile başlatılmıştır. IsDebuggerPresent() ve PEB->BeingDebugged butonlarının aktif hale getirilmiş halleri Şekil 6.2.’de gösterilmektedir.



Şekil 6.2. IsDebuggerPresent() ve PEB->BeingDebugged butonları etkin

Şekil 6.3.’te IsDebuggerPresent işlevine kesme noktası ayarlamak için yazılan komut gösterilmektedir.

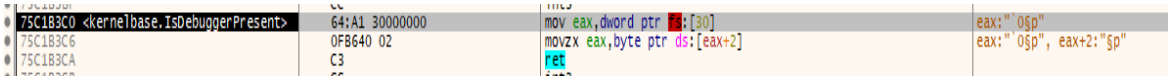


Şekil 6.3. IsDebuggerPresent() kesme noktası komutu

Kesme noktası için Enter tuşuna basıldığında program kesme noktasına isabet etmiş ve Şekil 6.4.'te gösterilen 778A20D0 adresinde kesmeye uğramıştır.

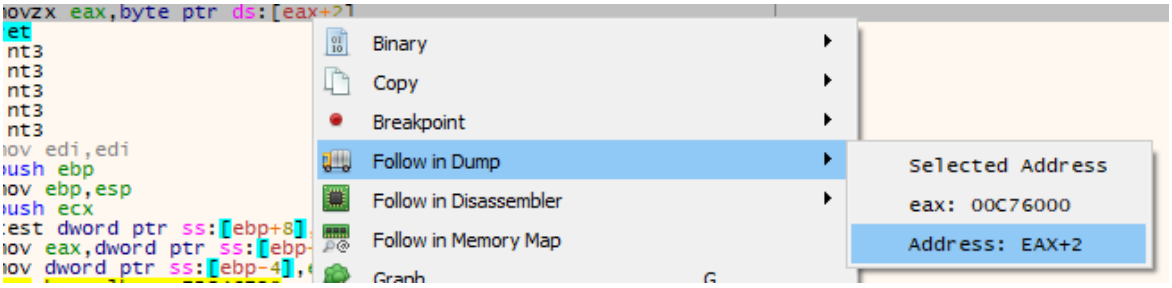
Şekil 6.4. 778A20D0 kesmesi

Tek adımda içe yürütme için F7 tuşuna basılmış ve Şekil 6.5.'te IsDebuggerPresent işlevinin kod bloğuna ulaşılmıştır.



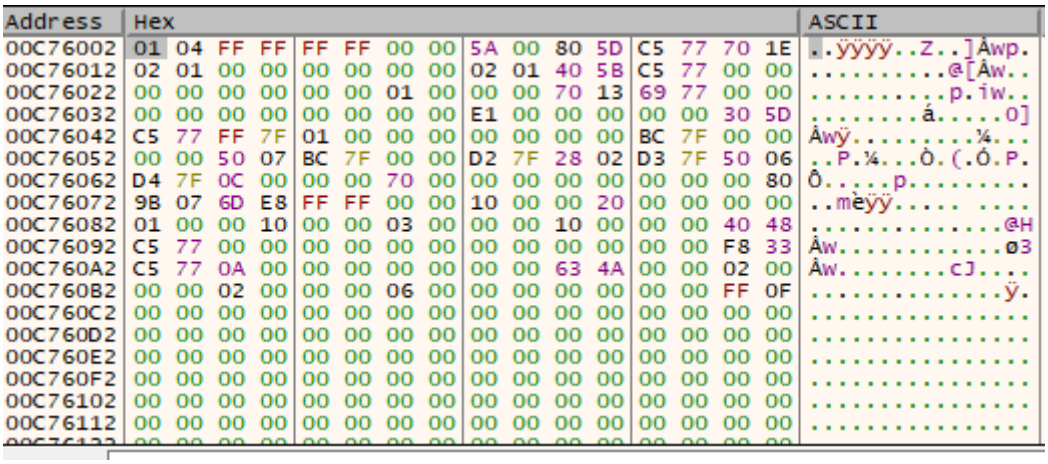
Şekil 6.5. IsDebuggerPresent() kod bloğu

Şekilde PEB ve oradan BeingDebugged alanına erişim görülmektedir. Tek adımda yürütmeye 1 kere devam edildikten sonra sağ tıklanmış ve Şekil 6.6.'da gösterilen Follow in Dump > Address: EAX+2 seçenekleri seçilmiştir. EAX'in değeri 00C76000'dir.



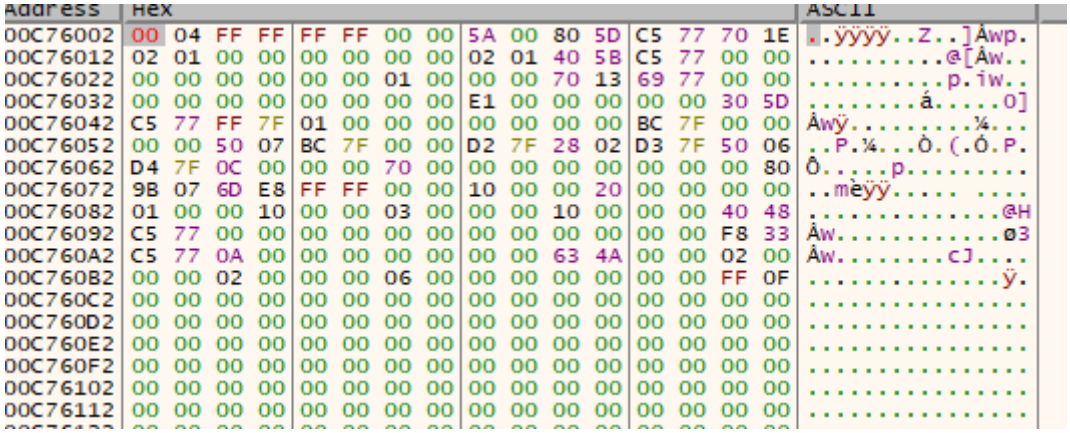
Şekil 6.6. Follow in Dump > Address: EAX+2 seçenekleri

Şekil 6.7.'de EAX+2 (00C76002) adresindeki değer dump sekmesinde gösterilmektedir.



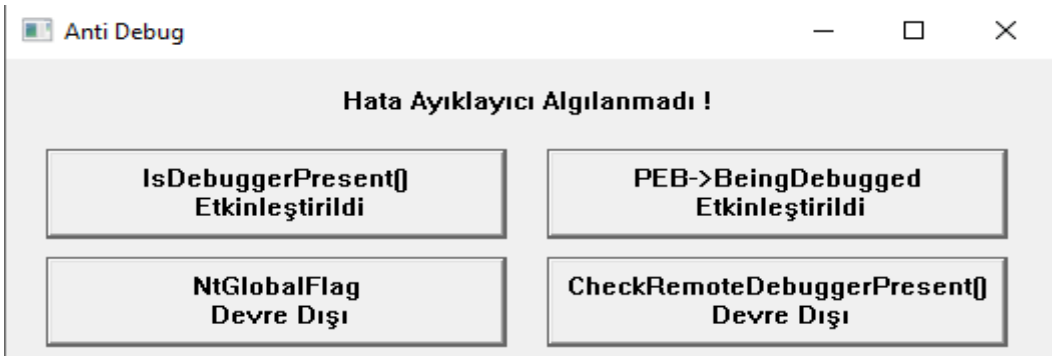
Şekil 6.7. 00C76002 adresinin dump sekmesinde gösterimi

Değerin 1 olması hata ayıklayıcının varlığını işaret etmektedir. Değeri 0 bayt ile değiştirmek bu hata ayıklama yöntemlerini aşmak için yeterlidir. 1 değerine çift tıklanıp 0 değeri ile değiştirilmiş hali Şekil 6.8.'de gösterilmektedir.



Şekil 6.8. Dump sekmesinde 0 bayt ile değişim

IsDebuggerPresent işlevindeki kesme noktası devre dışı bırakıldıktan sonra program yürütülmeye devam edildiğinde değişikliğe uğraya arayüz Şekil 6.9.'daki gibidir ve ilk iki hata ayıklamayı önleme yönteminin aşıldığı görülmektedir.



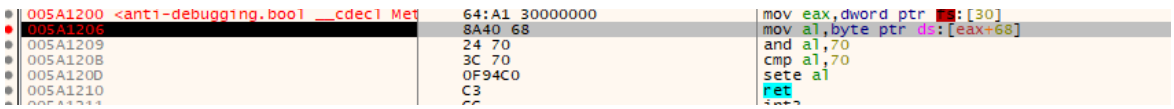
Şekil 6.9. İlk değişikliğe uğramış anti debug arayüzü

6.1.2 NtGlobalFlag yöntemini aşma

NtGlobalFlag işlevini aktive etmek için butona tıklanmış hali Şekil 6.10.'da gösterilmektedir.

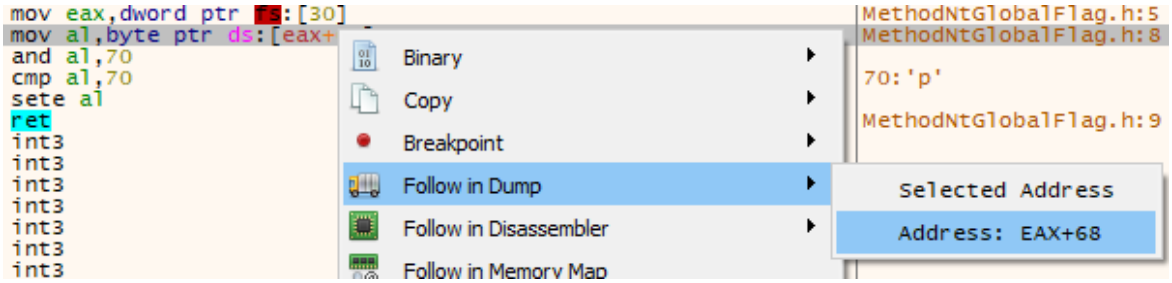
Şekil 6.10. NtGlobalFlag butonu etkin

Bu yöntem önce PEB'e sonra offseti 0x68h olan NtGlobalFlag'e erişimin sağlanması ile devam etmektedir. NtGlobalFlag alanına erişimin kod bloğu bulunmuş ve Şekil 6.11.'deki gibi bir kesme noktası yerleştirilmiştir.



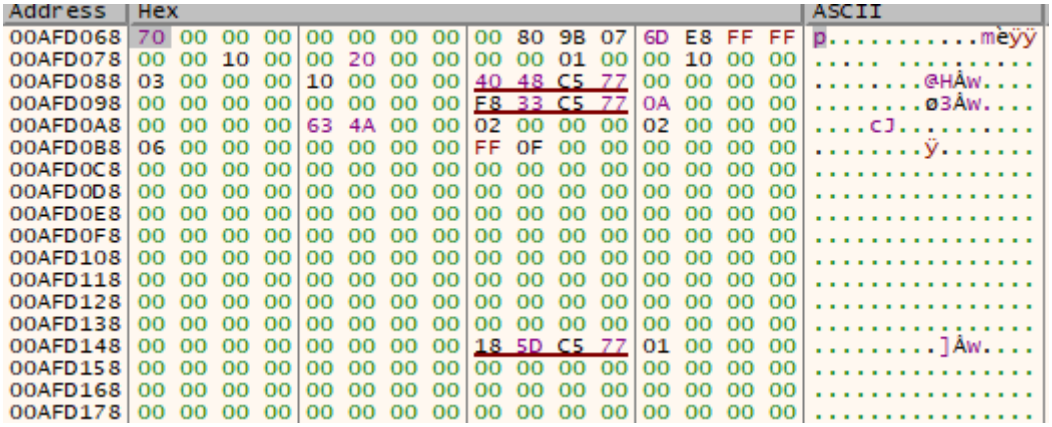
Şekil 6.11. NtGlobalFlag kesme noktası

005A1206 adresinde kesmeye uğramış programın NtGlobalFlag alanının değerini görmek için sağ tıklanmış ve Follow in Dump > Address : EAX+68 seçenekleri seçilmiştir.



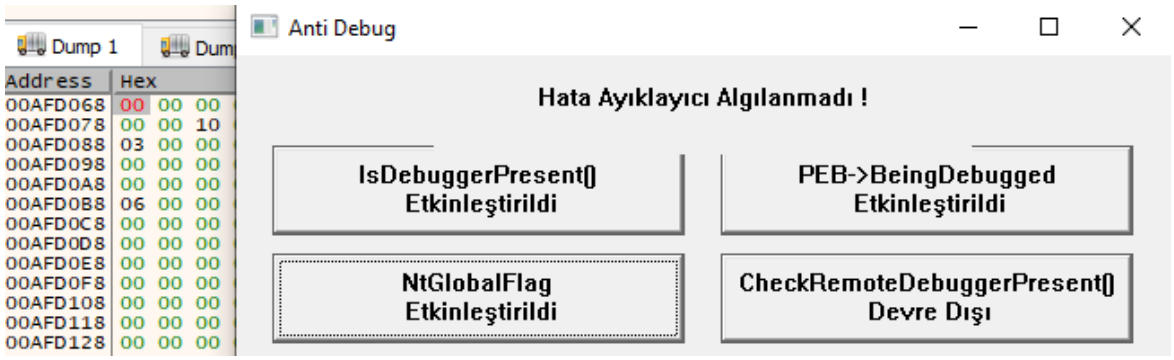
Şekil 6.12. Follow in Dump > Address : EAX+68 seçenekleri

Şekil 6.13.'te dump penceresinde NtGlobalFlag alanının değeri 70 gösterilmektedir.



Şekil 6.13. NtGlobalFlag alanın değeri

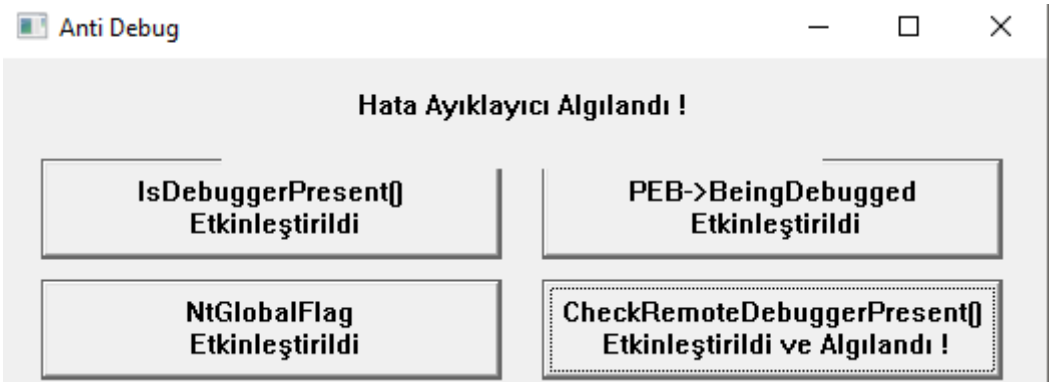
NtGlobalFlag alanının değerini 0 bayt ile değiştirmek bu hata ayıklama yöntemini aşmak için yeterlidir. Şekil 6.14.'te NtGlobalFlag alanının değeri değiştirilmiş ve kesme noktası devre dışı bırakıldıktan sonra programın yürütülmesine devam edilmiştir. Şekil 6.14.'te programın arayüzüne bakıldığında bu hata ayıklama yönteminin de aşıldığı görülmektedir.



Şekil 6.14. İkinci değişikliğe uğramış anti debug arayüzü

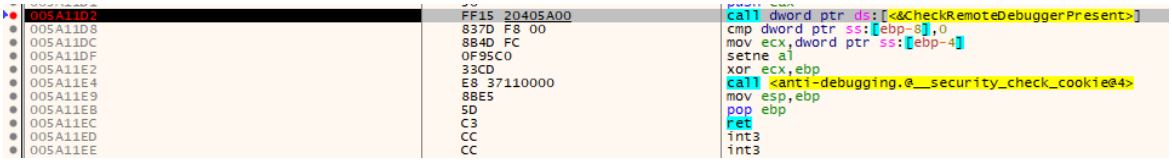
6.1.3 CheckRemoteDebuggerPresent yöntemini aşma

CheckRemoteDebuggerPresent() işlevini aktive etmek için butona tıklanmıştır ve Şekil 6.15.'te gösterilmiştir.



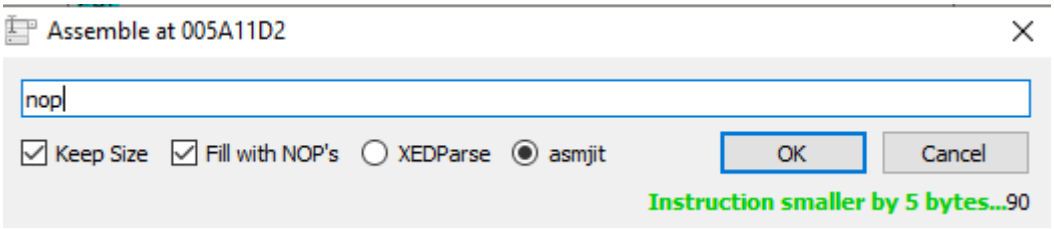
Şekil 6.15. CheckRemoteDebuggerPresent butonu etkin

Bu hata ayıklama yöntemini aşmada NtQueryInformationProcess() işlevinin geri döndüğü kod bloğuna bir kesme noktası ayarlayıp ProcessInformation parametresini 0x00 değeri ile değiştirmekten ziyade, işlev çağrısının kapladığı bellek alanını NOP talimatları ile değiştirme yöntemi seçilmiştir. CheckRemoteDebuggerPresent işlevine bir kesme noktası ayarlanmış ve program Şekil 6.16.'da gösterildiği gibi kesmeye uğramıştır.



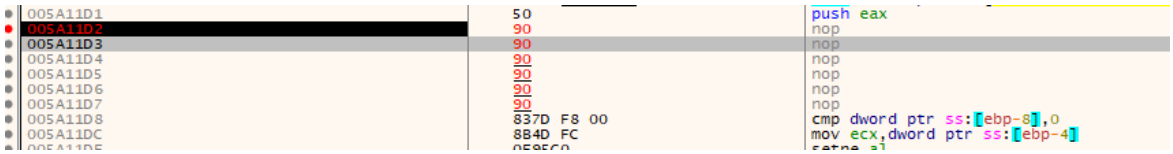
Şekil 6.16. CheckRemoteDebuggerPresent kesme noktası

Space tuşuna basıldıktan sonra ekrana gelen penceredeki talimat nop ile değiştirilmiş, Keep Size ve Fill with NOP's seçenekleri seçilmiştir. Şekil 6.17.'de yapılan değişiklikler gösterilmektedir.



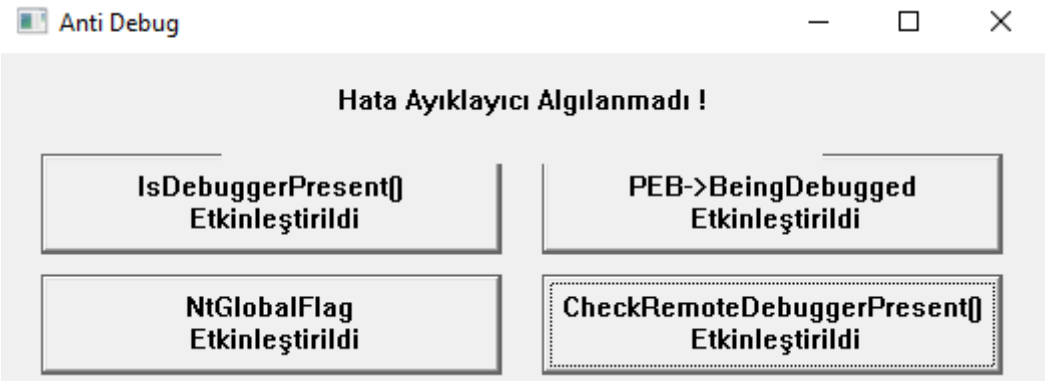
Şekil 6.17. CheckRemoteDebuggerPresent kesme noktası

CheckRemoteDebuggerPresent çağrısının kapladığı tüm hafıza alanı nop talimatı ile Şekil 6.18.'deki gibi değiştirilmiştir.



Şekil 6.18. Nop talimatı ile doldurma

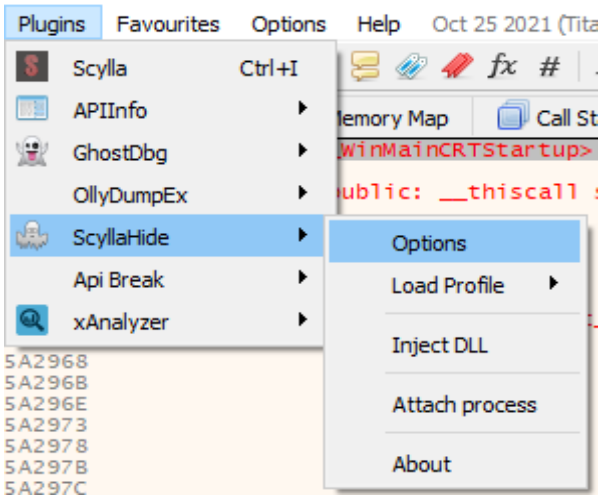
Kesme noktası devre dışı bırakılıp program yürütüldüğünde Şekil 6.19.'da gösterildiği gibi hata ayıklama yönteminin aşıldığı görülmektedir.



Şekil 6.19. Üçüncü değişikliğe uğramış anti debug arayüzü

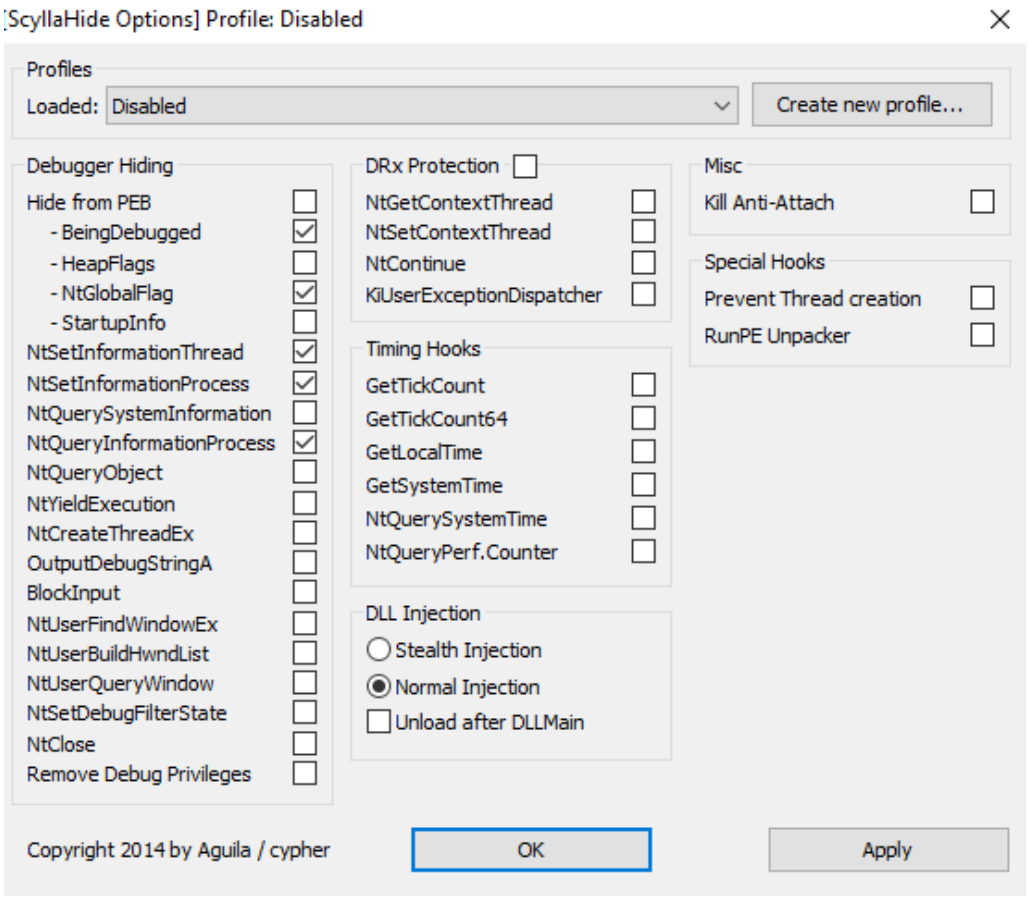
6.1.4 ScyllaHide eklentisi

Bu tip hata ayıklamayı önleme yöntemlerini daha kolay bir şekilde aşmak için ScyllaHide ve benzeri eklentiler de kullanılmaktadır. Program, hata ayıklayıcı ile tekrar başlatılır. Şekil 6.20'de gösterildiği gibi Plugins > ScyllaHide > Options seçenekleri seçilir.



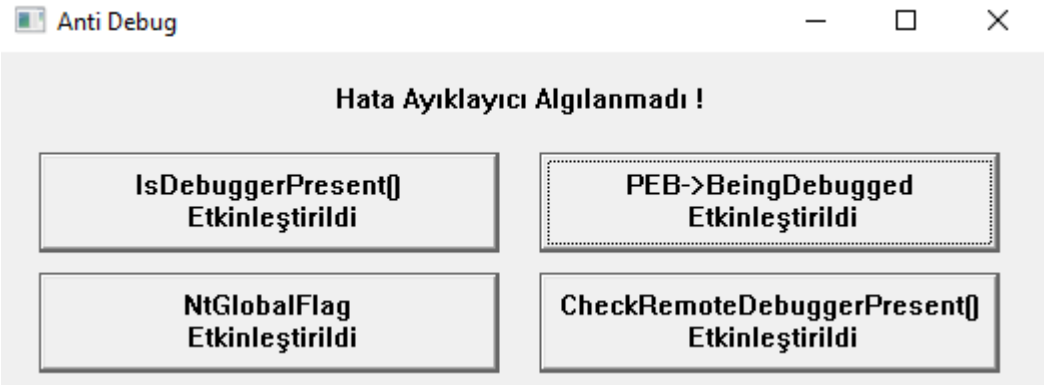
Şekil 6.20. Plugins > ScyllaHide > Options seçenekleri

ScyllaHide arayüzü Şekil 6.21.'de gösterilmiş ve "Anti Debug" isimli programdan hata ayıklayıcıyı saklamak için seçilmesi gereken seçenekler seçilmiştir.



Şekil 6.21. ScyllaHide arayüzü ve seçenekler

Anti Debug programındaki tüm butonlar aktif hale getirilmiş ve Şekil 6.22.'de görüldüğü gibi hata ayıklayıcı algılanmamıştır.

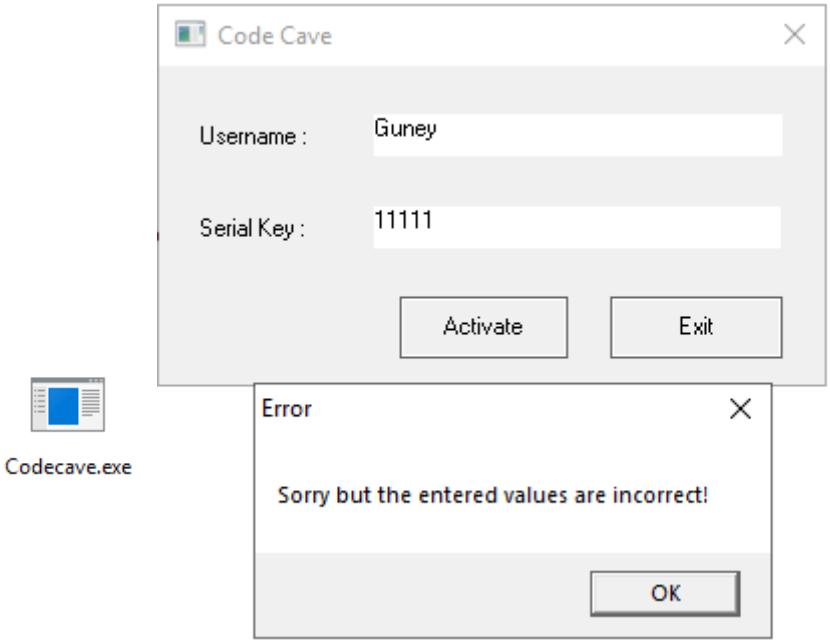


Şekil 6.22. ScyllaHide ve anti debug arayüzü

6.2 Örnek 2

Bu çalışmada kod mağaraları kullanılarak yazılımın kullanıcı adları için ürettiği seri anahtarı ekrana bastırılmış ve bu sayede programın aktivasyon için gereken seri anahtarı elde edilmiştir. Bu program teknik olarak yeni bir bölüm eklenmesini gerektirmemektedir ancak manuel olarak bir bölümün nasıl ekleneceğini anlamak ve aynı zamanda bir PE dosyasının nasıl yapılandırıldığına dair fikir vermek için kod mağarası, yeni bir bölüm eklenerek oluşturulmuştur.

Şekil 6.23'te Code Cave isimli programa kullanıcı adı ve seri numarası girildiğinde ekrana gelen hata mesajı gösterilmektedir.



Şekil 6.23. Code cave arayüzü ve hata mesajı

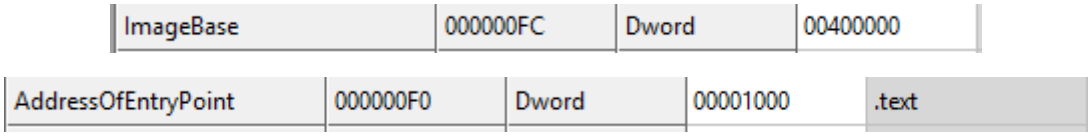
CFF Explorer ile içeriğine bakılan programın Şekil 6.24'te bölümleri ve bu bölümlere ait bilgiler gösterilmektedir.

Codecave.exe

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	0000030A	00001000	00000400	00000400	00000000	00000000	0000	0000	60000020
.rdata	00000196	00002000	00000200	00000800	00000000	00000000	0000	0000	40000040
.data	00000138	00003000	00000200	00000A00	00000000	00000000	0000	0000	C0000040
.rsrc	000001B0	00004000	00000200	00000C00	00000000	00000000	0000	0000	40000040

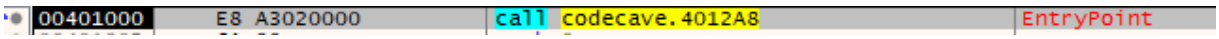
Şekil 6.24. CFF Explorer ile bölüm bilgileri

Bu programın .text .rdata .data ve .rsrc adında 4 bölümü bulunmaktadır. Başlangıç adresleri ve boyutları, Raw Size ve Raw Adress sütunlarında görülmektedir. Bu sütunlar dosya bölümünün dosyadaki verilerinin ham boyutunu ve verilerin ham haliyle nerede olduğunu göstermek için kullanılır. Şekil 6.24'te text bölümünün 0400'den başladığı ve 0400 boyutunda olduğu görülmektedir. Windows yükleyici programı belleğe yüklediğinde, bu bölümleri diskten kopyalar ve belleğe yerleştirir. Virtual Size ve Virtual Address ise verilerin belleğe nerede kopyalanacağını ve alanın ne kadar büyük olacağını gösterir. .text bölümünde başlangıç adresinin 01000 ve kopyalanan verinin boyutunun 030A olacağı şekil 6.24.'te gösterilmektedir. Yükleyici, programı diskten yüklediğinde, programın 0400'den başlayan verileri 01000 bellek adresinden başlayarak belleğe yüklenmektedir. 01000 bu bellek alanının boyutudur, 30A verinin boyutudur. 01000 – 030A = 0CF6 matematiksel hesabı göz önünde bulundurulduğunda bu bölümün sonunda içinde sıfır bulunan 0CF6 bayt boyutunda bir alan kalmaktadır. Programın AdressOfEntryPoint ve ImageBase değeri şekil 6.25'te görüldüğü gibidir.



Şekil 6.25. AdressOfEntryPoint ve ImageBase değeri

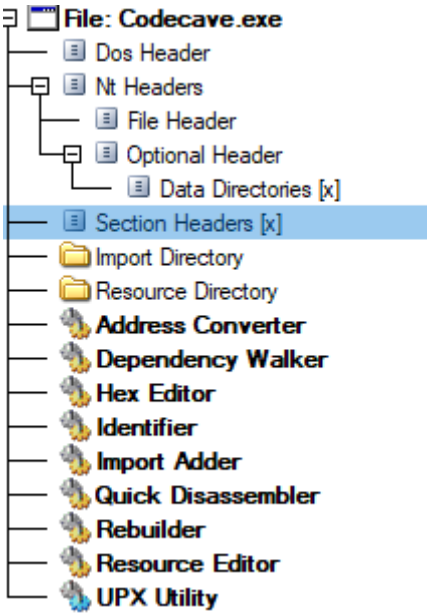
Programı hata ayıklayıcı ile başlatıldığında giriş noktasının Şekil 6.26'daki gibi 401000 olduğu görülmektedir.



Şekil 6.26. Code cave giriş noktası

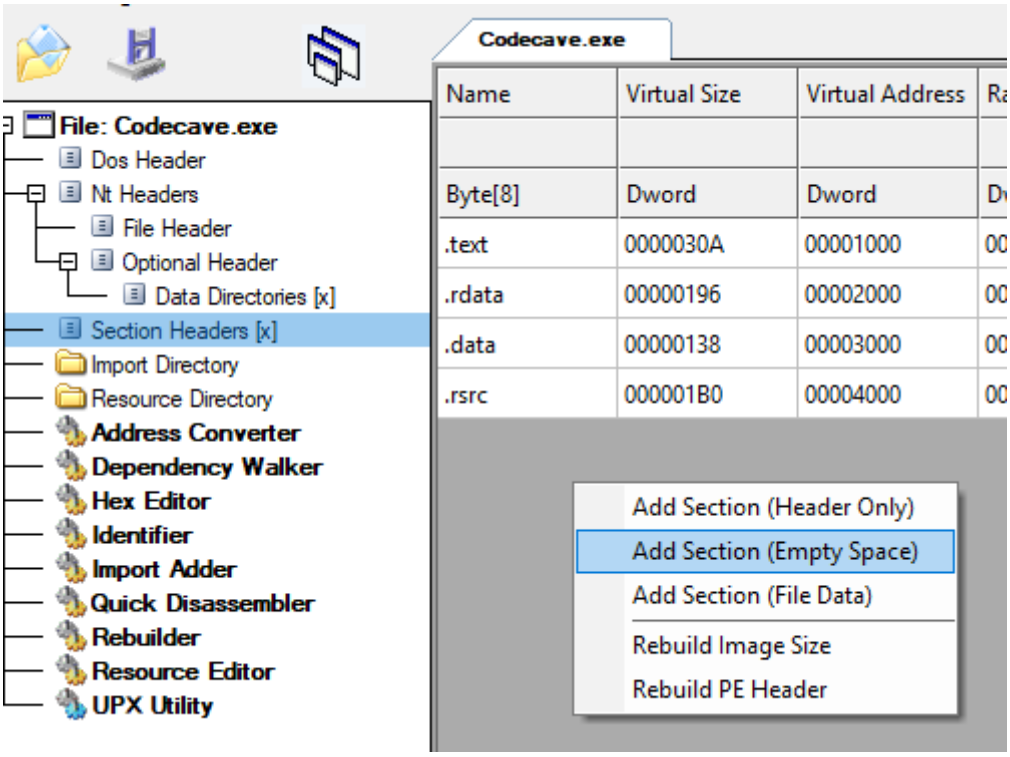
6.2.1 Manuel olarak yeni bölüm ekleme

CFF Explorer ile görüntülenen programa yeni bölüm eklemek için Şekil 6.27.'de bulunan Section Headers sekmesi seçilmiştir.



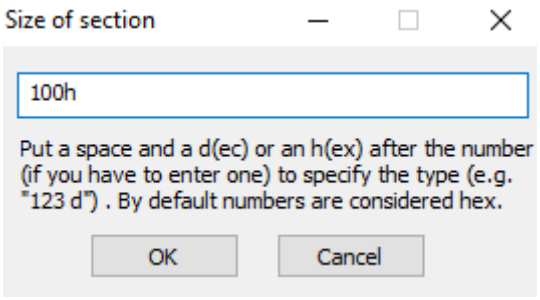
Şekil 6.27. Section headers sekmesi

Section Headers sekmesine tıklandıktan sonra çıkan ekrana Şekil 6.28.'de gösterildiği gibi sağ tıklanarak Add Section (Empty Space) seçeneği seçilmiştir.



Şekil 6.28. Add section (empty space) seçeneği

Eklenecek bölümün boyutu Şekil 6.29.'da gösterildiği gibi 100h yani 256 bayt olarak girilmiştir.



Şekil 6.29. eklenecek bölümün boyutu

Eklenen yeni bölüm Şekil 6.30.'da gösterildiği gibi .ornekcc olarak adlandırılmıştır.

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
00000260	00000268	0000026C	00000270	00000274	00000278	0000027C	00000280	00000282	00000284
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	0000030A	00001000	00000400	00000400	00000000	00000000	0000	0000	60000020
.rdata	00000196	00002000	00000200	00000800	00000000	00000000	0000	0000	40000040
.data	00000138	00003000	00000200	00000A00	00000000	00000000	0000	0000	C0000040
.rsrc	000001B0	00004000	00000200	00000C00	00000000	00000000	0000	0000	40000040
.ornekcc	00000100	00005000	00000200	00000E00	00000000	00000000	0000	0000	C0000000

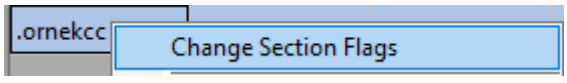
Şekil 6.30. .ornekcc yeni bölümü

Optional Header sekmesine girildiğinde Şekil 6.31.'de gösterildiği gibi bir bölümün varsayılan boyutunun bellekte 01000 bayt olduğu "SectionAligment" alanında belirtilmiştir. FileAlignment alanında ise bölümün diskte kaplayabileceği minimum alan belirtilmiştir.

SectionAlignment	00000100	Dword	00001000
FileAlignment	00000104	Dword	00000200

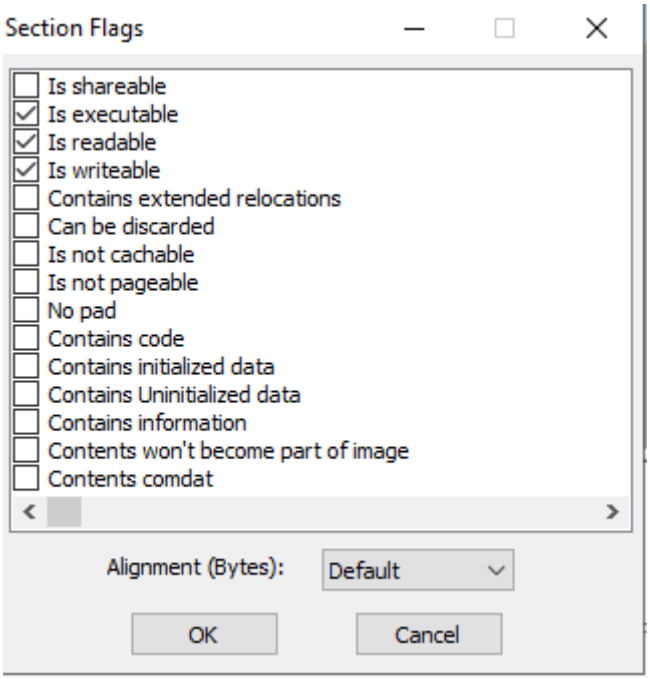
Şekil 6.31. SectionAlignment ve FileAlignment alanları

Yeni bölümü çalıştırılabilir hale getirmek yani kodun içinde çalışmasına izin vermek için eklenen bölüme sağ tıklanıp Şekil 6.32.'de görülen "Change Section Flags "seçeneği seçilmiştir.



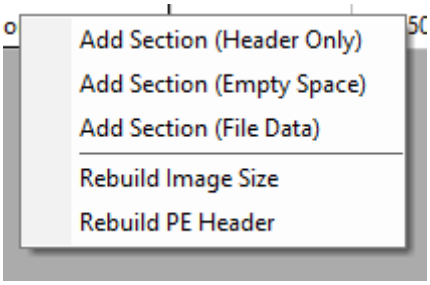
Şekil 6.32. Change section flags seçeneği

Şekil 6.33.'de gösterilen Section Flags isimli pencerede "Is executable" özelliğine tik koyularak kod mağarasının bellekte çalıştırılabilir olması sağlanmıştır.



Şekil 6.33. Section flags penceresi

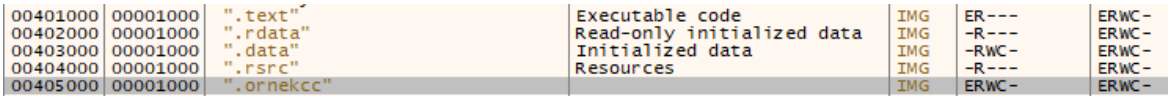
Yeni bölüm eklendiği için programın boyutunun ve başlığının güncellenmesi gerekmektedir. Şekil 6.34'te gösterildiği gibi eklenilen yeni bölüme sağ tıklanıp "Rebuild Image Size" seçeneği seçilmiştir. Bu, programın toplam boyutuna 0200 bayt ekleyecektir. Ardından, eklenilen yeni bölüme sağ tıklanarak "Rebuild PE Header" seçeneği seçilmiştir. Bu, bölüm sayısı alanını ve yeni bölümü yüklemek için gerekli diğer alanları güncellemiştir.



Şekil 6.34. Rebuild image size seçeneği

File sekmesinden "Save As" seçeneği seçilerek yeni dosya Codecaveyeni.exe ismi ile kaydedilmiştir.

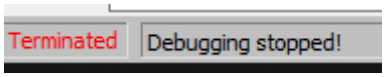
x64dbg hata ayıklayıcısı ile yeni program açıldığında “Memory Map” sekmesinde eklenilen bölüm Şekil 6.35.’deki gibi görüntülenmektedir. Executable, Read, Write and Copy kelimelerinin baş harflerinin temsil ettiği gösterim(ERWC-) mevcut bölümün hangi işlemleri yapabileceğini ifade etmektedir.



Şekil 6.35. Yeni bölümün memory map sekmesindeki görünüşü

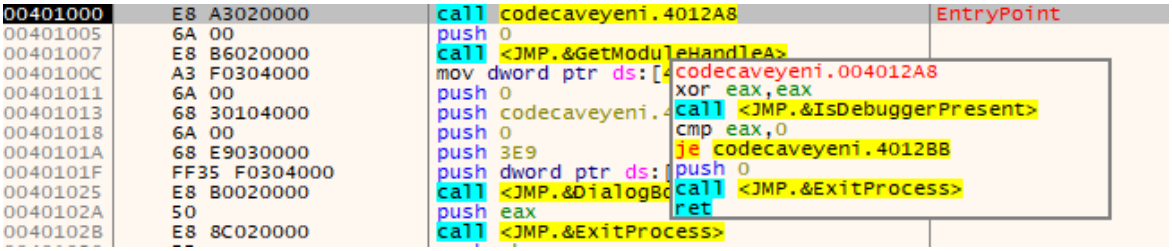
6.2.2 Araştırma ve hata ayıklamayı önlemeden kurtulma aşaması

Program, hata ayıklayıcı ile yürütülmek istendiğinde Şekil 6.36.’da gösterilen hata ile karşılaşılmakta ve program yürütülmeden kapanmaktadır.



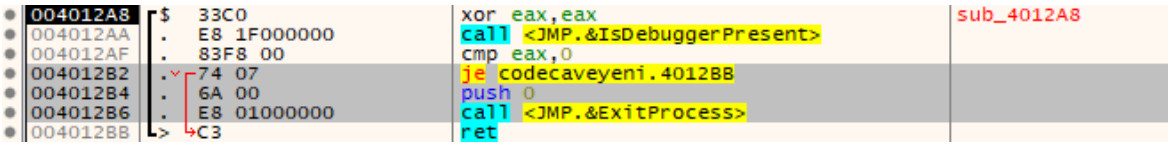
Şekil 6.36. Hata ayıklayıcıda karşılaşılan hata

Program tek adımlı hata ayıklama yöntemi ile izlendiğinde programın giriş noktası (Entry Point) Şekil 6.37.’de görülen 004012A8 adresindeki xor eax,eax kodu ile başlayan kod bloğunu çağırmaktadır ve o kod bloğu içinde API tabanlı hata ayıklamayı önleme işlevlerinden biri olan IsDebuggerPresent işlevi çağrılarak eax değeri 0 ile karşılaştırılmaktadır. Karşılaştırmanın sonucunda eax 0’a eşit ise 4012BB adresine atlanılacak, eşit değilse ExitProcess işlevi ile programdan çıkış yapılacaktır.

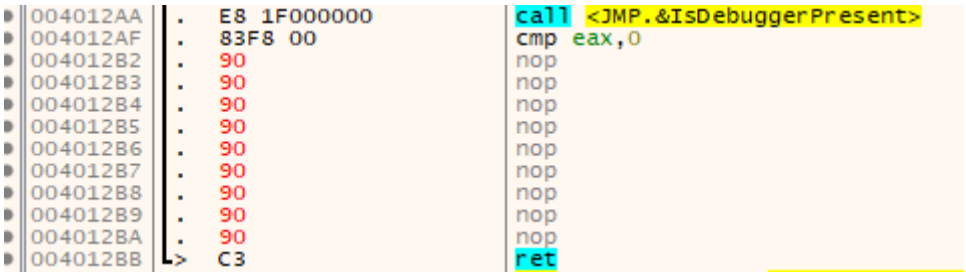


Şekil 6.37. AdressOfEntryPoint ve ImageBase değeri

IsDebuggerPresent hata ayıklamayı önleme işlevi birçok teknik ile kolayca atlanabilir. Kod mağarası örneğine hızlıca devam etmek için Şekil 6.38.’de seçili olarak görülen koyu alandaki talimatlar Şekil 6.39.’da gösterilen NOP talimatı ile değiştirilmiştir.

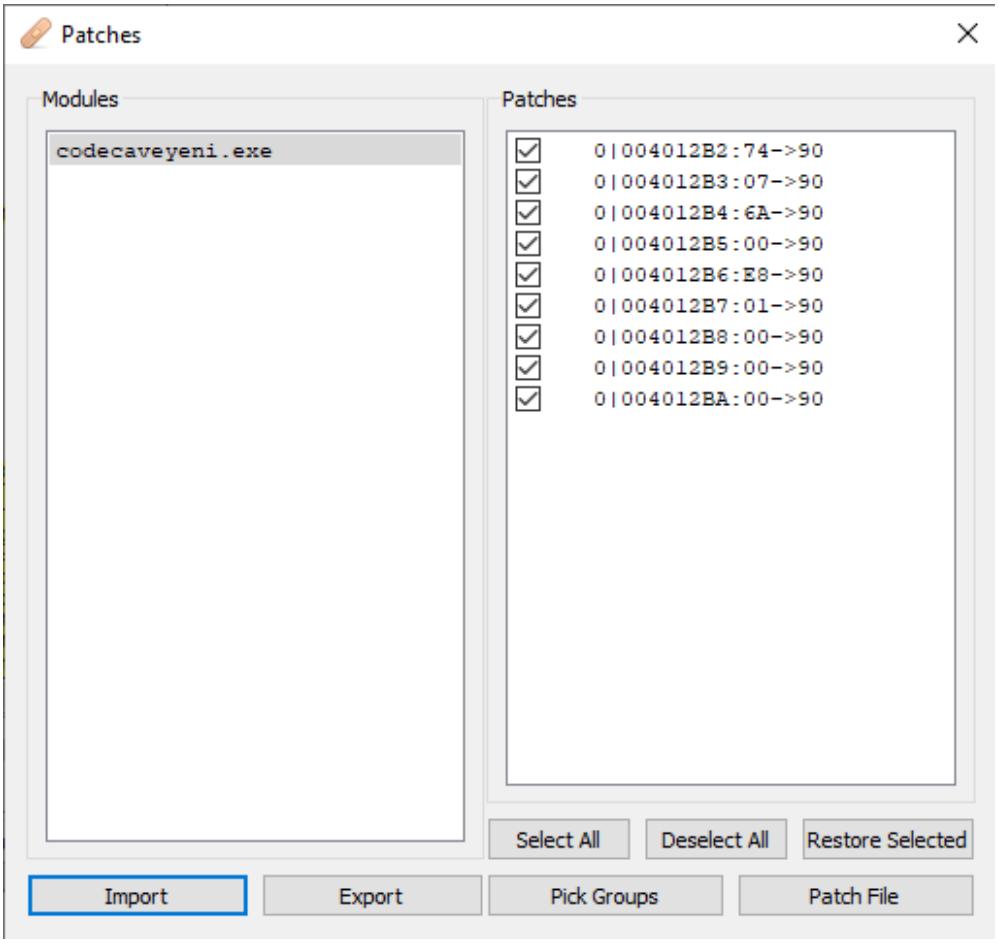


Şekil 6.38. Seçili kod bloğu



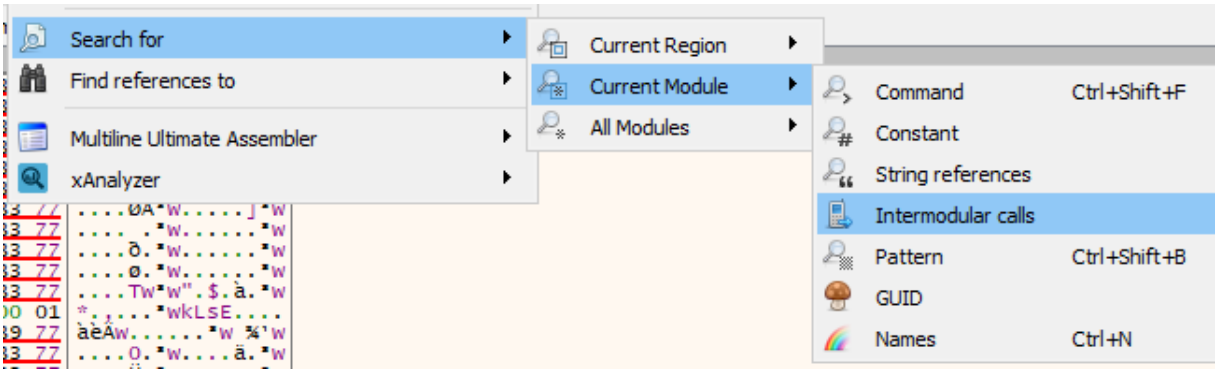
Şekil 6.39. Nop talimatları ile değişim

File sekmesinden Patch file seçeneği seçilmiş ve Şekil 6.40.'da açılan pencereden Patch File butonuna tıklanmıştır. Yeni dosya Codecavetemiz.exe ismiyle kaydedilmiştir.



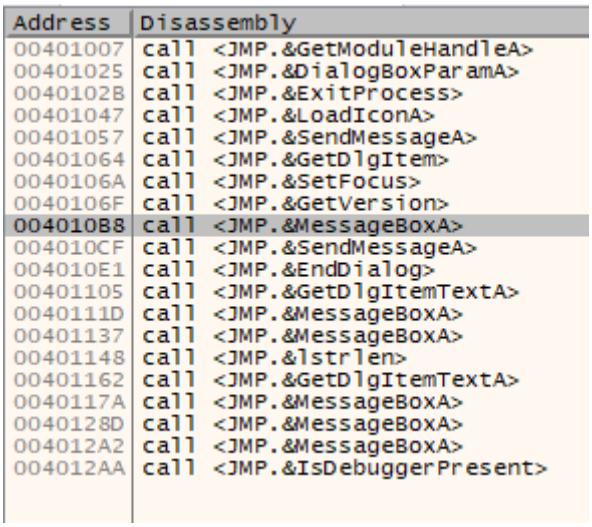
Şekil 6.40. Patch Penceresi

Yeni dosya hata ayıklayıcısı ile açıldıktan sonra ana modüle sağ tıklanarak Şekil 6.41.'de gösterilen Search for > Current Module > Intermodular Calls seçeneği seçilmiştir.

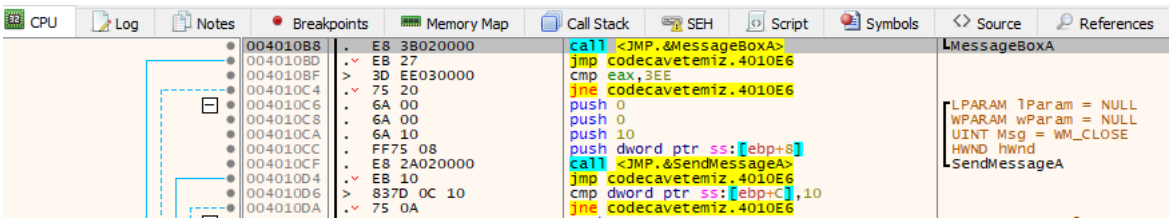


Şekil 6.41. Search for > Current Module > Intermodular Calls seçeneği

Codecavetemiz isimli programın mesaj kutusu bastırdığı bilindiğinden Şekil 6.42.'de gösterilen MessageBoxA fonksiyonuna çift tıklanarak CPU ekranında fonksiyonun çağrıldığı Şekil 6.43.'teki kod bloğuna gidilir.

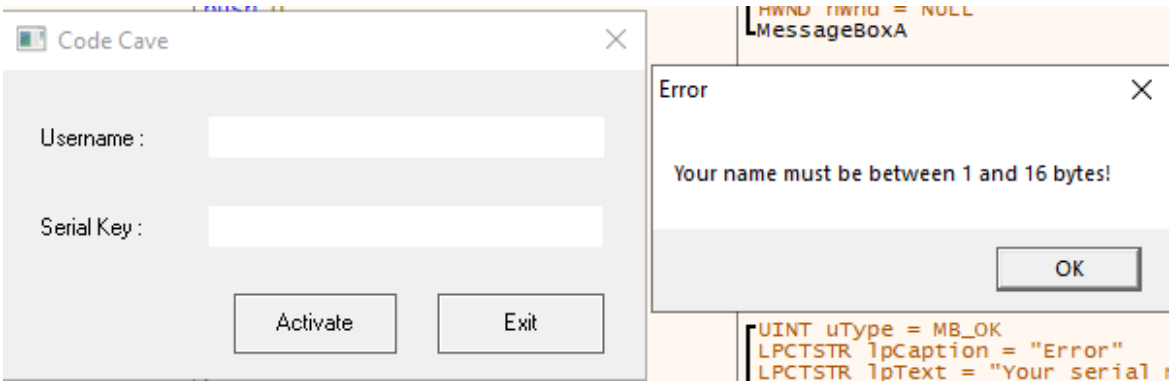


Şekil 6.42. MessageBoxA fonksiyonu



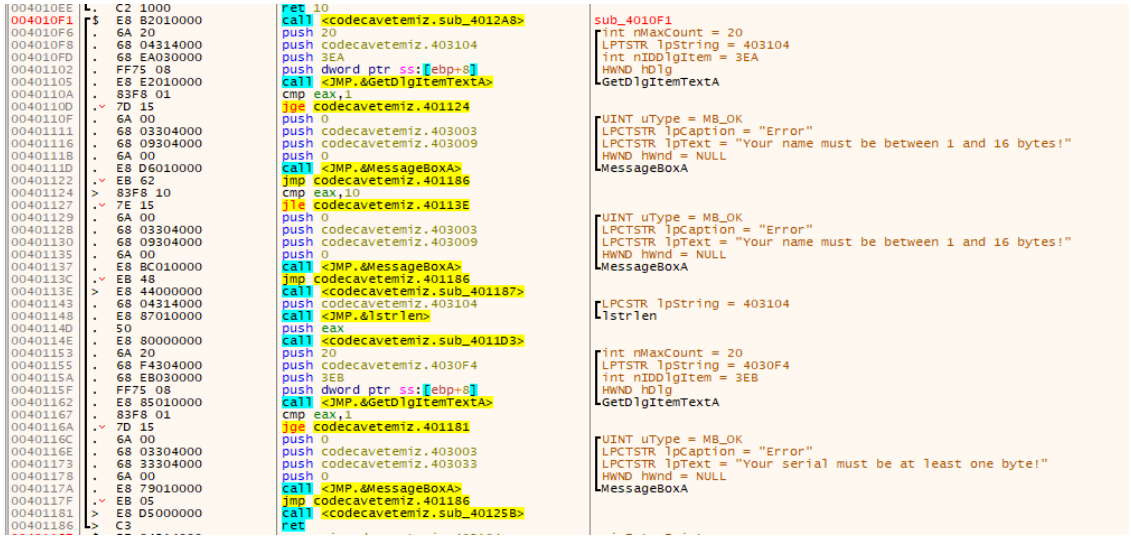
Şekil 6.43. MessageBoxA fonksiyonu

Programa hiçbir giriş yapmadan Activate butonuna tıklandığında karşılaşılan hata Şekil 6.44.'te gösterildiği gibidir.



Şekil 6.44. Hata mesajı

Şekil 6.45.'te kod bloğunda açıkça görüldüğü gibi 401105 adresindeki GetDlgItemTextA fonksiyonu ile kullanıcı adı alınmakta, bu kullanıcı adı 403105 adresinde saklanmakta ve eax yazmacına, karakterlerin bayt cinsinden boyutu yazılmaktadır. 40110A adresinde eax yazmacının değeri 1 ile karşılaştırılmaktadır. Bu karşılaştırmanın sonucunda eğer eax 1'e eşit veya 1'den büyükse ilk MessageBoxA, 40110D adresindeki jge talimatı ile atlanacaktır. 2. Kod bloğunda eax değerinin 16 ya eşit veya 16'dan küçük olması halinde 401127 adresindeki jle talimatı ile 2. MessageboxA fonksiyonu atlanacak ve 40113E adresinden program yürütülmeye devam edecektir.



Şekil 6.45. Ana kod bloğu

Şekil 6.45.'te gösterilen 401162 adresindeki GetDlgItemTextA fonksiyonu seri anahtarı girdisini almak için kullanılmakta ve eax yazmacına karakterlerin toplam bayt

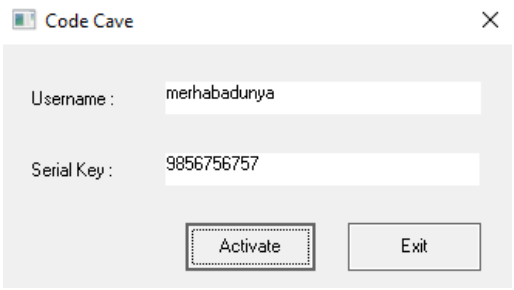
boyutu yazılmaktadır. Takip eden kod akışında eax yazmacının değeri 1 ile karşılaştırılmakta ve eğer 1'e eşit veya 1'den büyükse seri anahtarı için olan MesssageBoxA fonksiyonu atlanmaktadır. Bu karşılaştırmanın yapılabilmesi için seri anahtarının üretilmiş olması veya yazılımın içine gömülmüş olması gerekmektedir.

0040113E adresindeki alt rutin çağrısı 00401187 adresinden 004011B6 adresine kadar olan Şekil 6.46.'da gösterilen kod bloğunu çağırmakta ve bu kod bloğu seri anahtarının kullanıcı adına göre üretilmesini sağlamaktadır.

```
00401187  $  BE 04314000    mov esi,codecavetemiz.403104        sub_401187
0040118C  .  BF 14314000    mov edi,codecavetemiz.403114
00401191  .  B9 10000000    mov ecx,10                          ecx:sub_77192700+C
00401196  .  8B1D 34314000  mov ebx,dword ptr ds:[403134]
0040119C  >  8A06           mov al,byte ptr ds:[esi]
0040119E  .  3C 00          cmp al,0
004011A0  .v 75 02          jne codecavetemiz.4011A4
004011A2  .  8BC1           mov eax,ecx                         ecx:sub_77192700+C
004011A4  >  32C3           xor al,bl
004011A6  .  86DF           xchg bh,bl
004011A8  .  32C3           xor al,bl
004011AA  .  50             push eax
004011AB  .  E8 07000000    call <codecavetemiz.sub_4011B7>
004011B0  .  8807           mov byte ptr ds:[edi],al
004011B2  .  46             inc esi
004011B3  .  47             inc edi
004011B4  .^ E2 E6          loop codecavetemiz.40119C
004011B6  L. C3             ret
```

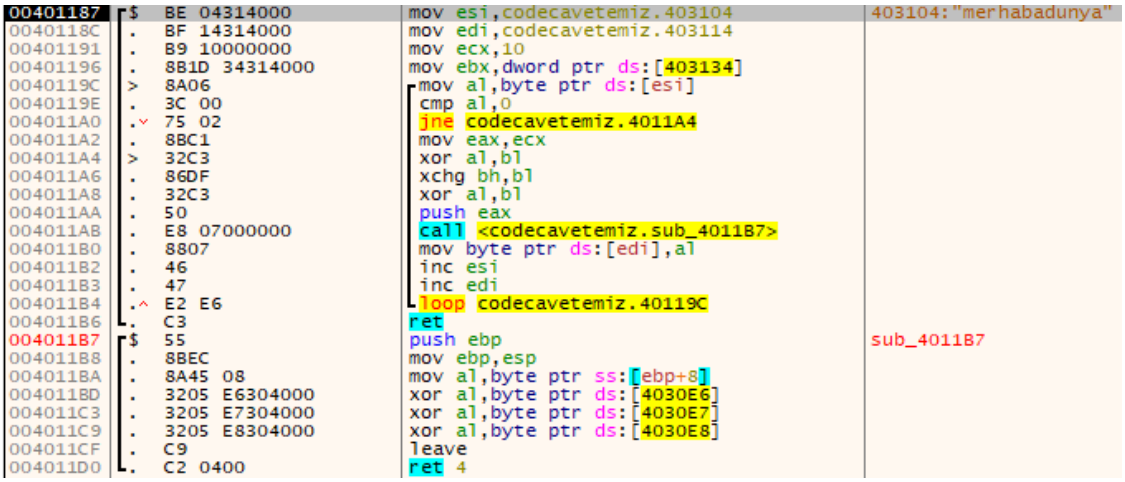
Şekil 6.46. Kullanıcı adına göre seri numarası üreten kod bloğu

0040113E adresine bir kesme noktası yerleştirildikten sonra programdaki Username ve Serial Key bölümlerine Şekil 6.47.'deki gibi her hangi bir kullanıcı adı ve seri numarası girilir ve Activate butonuna tıklanır. Program kesme noktasından dolayı 0040113E adresinde duraklar.



Şekil 6.47. Kullanıcı adı ve seri numarası girdisi

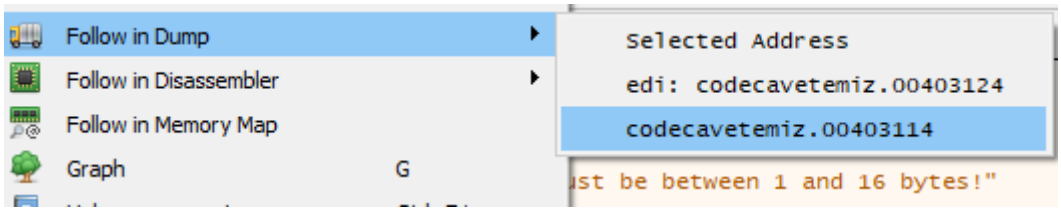
0040113E adresindeki Call talimatı ile 00401187 alt rutini çağrılır ve bu rutin Şekil 6.48.'de görüldüğü gibi bir döngüye sahiptir. Bu döngünün içinde 004011AB adresinden yapılan bir çağrı daha mevcuttur ve bu çağrı 004011B7 – 004011D0 adres aralığındaki alt rutini çağırır.



Şekil 6.48. 00401187 alt rutini

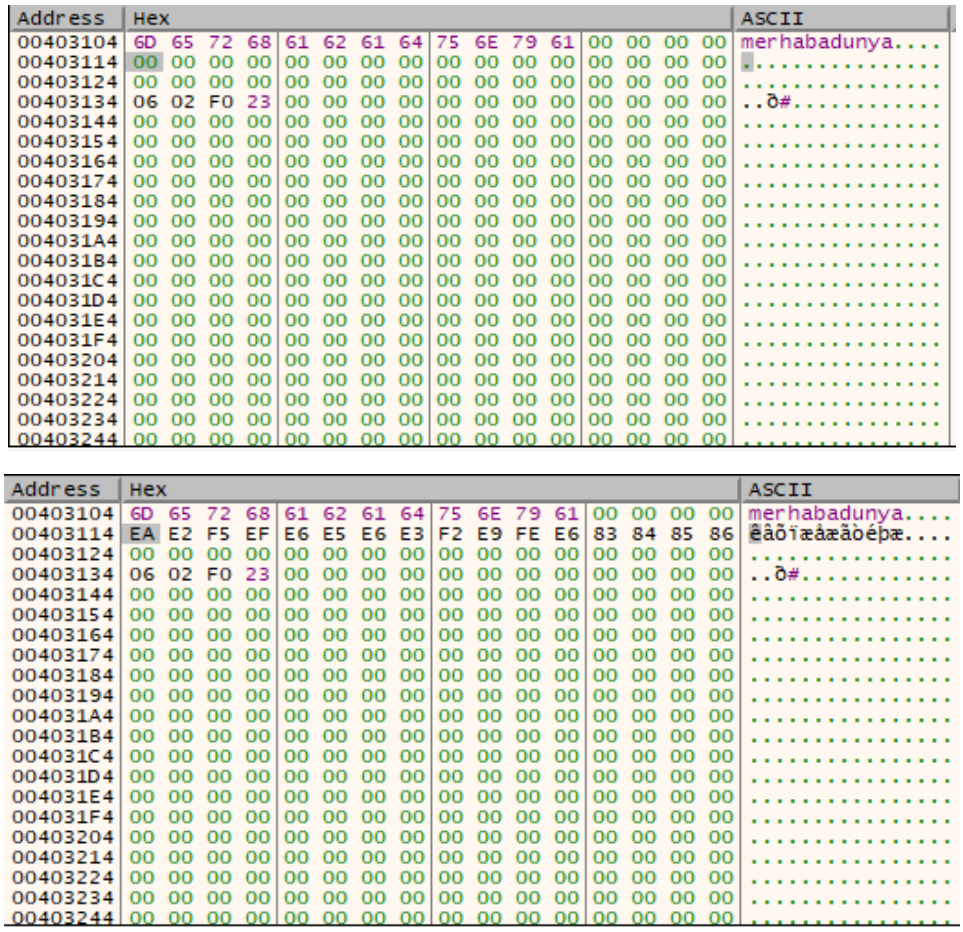
004011B7 ile başlayan alt rutinde 3 kere üst üste işletilen bir XOR Şifreleme algoritması olduğu açıkça görülmektedir. 00401187 isimli alt rutinin aslında Şifreleme yani encryption rutini olduğu ve kullanıcı adını kullanarak şifrelenmiş bir seri anahtarı ürettiği gösterilmektedir.

Şifrelenmiş seri anahtarı bellek dökümünde görmek için öncelikle belleğin hangi adresine kopyalanacağını bulmak gerekmektedir. Yapılan araştırmada 00403104 adresinden itibaren girilen kullanıcı adı bilgisi saklanmakta ve 00403114 adresinden itibaren ise şifrelenmiş seri anahtarı bilgisi saklanmaktadır. Şifrelenmiş seri anahtarı bilgisini görmek için Şekil 6.49.'da gösterildiği gibi sağ tıklanır. Follow in Dump seçeneği seçildikten sonra codecavetemiz.00403114 adlı seçenek seçilir.



Şekil 6.49. Follow in dump seçeneği

00401187 alt rutininin çağrıldığı 0040113E adresinden sonraki adrese alt rutindeki şifreleme işleminin sonucunu görmek için bir kesme noktası yerleştirilir. Şifreleme işleminin başladıktan ve bittikten sonra Dump sekmesindeki 00403114 adresinden itibaren oluşan değişiklikler 6.50'de gösterildiği gibidir.



Address	Hex	ASCII
00403104	6D 65 72 68 61 62 61 64 75 6E 79 61 00 00 00 00	merhabadunya....
00403114	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403124	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403134	06 02 F0 23 00 00 00 00 00 00 00 00 00 00 00 00	..ð#............
00403144	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403154	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403164	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403174	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403184	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403194	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004031A4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004031B4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004031C4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004031D4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004031E4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004031F4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403204	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403214	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403224	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403234	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403244	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

Address	Hex	ASCII
00403104	6D 65 72 68 61 62 61 64 75 6E 79 61 00 00 00 00	merhabadunya....
00403114	EA E2 F5 EF E6 E5 E6 E3 F2 E9 FE E6 83 84 85 86	êâõïæåæãòéþæ....
00403124	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403134	06 02 F0 23 00 00 00 00 00 00 00 00 00 00 00 00	..ð#............
00403144	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403154	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403164	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403174	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403184	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403194	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004031A4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004031B4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004031C4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004031D4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004031E4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004031F4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403204	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403214	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403224	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403234	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403244	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

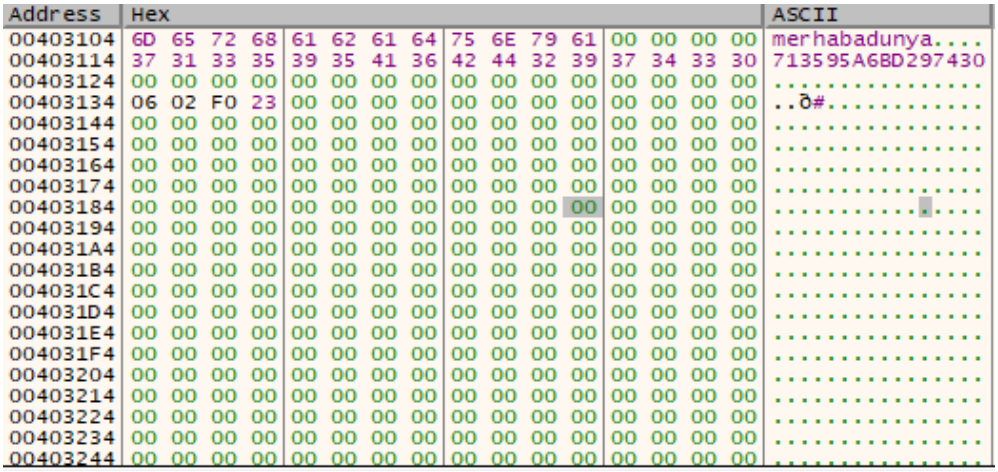
Şekil 6.50. Dump sekmesindeki 00403114 adresinden itibaren oluşan değişiklikler

0040114E adresinde çağrılan ve 004011D3 adresi ile başlayan Şekil 6.51.'de gösterilen alt rutin ise şifre çözme yani decrypt işlemini temsil etmektedir.

```
004011D3  r$  55              push ebp                          sub_4011D3
004011D4  .   8BEC            mov ebp,esp
004011D6  .   83C4 FC         add esp,FFFFFFFC
004011D9  .   BE 04314000     mov esi,codecavetemiz.403104      403104:"merhabadunya"
004011DE  .   BF 14314000     mov edi,codecavetemiz.403114
004011E3  .   B9 10000000     mov ecx,10
004011E8  .   C645 FF 00      mov byte ptr ss:[ebp-1],0
004011EC  >   8A06            mov al,byte ptr ds:[esi]
004011EE  .   3C 00           cmp al,0
004011F0  .v  75 02           jne codecavetemiz.4011F4
004011F2  .   B0 3F           mov al,3F                         3F:'?'
004011F4  >   F6E1            mul cl
004011F6  .   51              push ecx
004011F7  .   33C9            xor ecx,ecx
004011F9  .   8B4D 08         mov ecx,dword ptr ss:[ebp+8]
004011FC  >   8A5C31 FF       mov bl,byte ptr ds:[ecx+esi-1]
00401200  .   32C3            xor al,bl
00401202  .   49              dec ecx
00401203  .^  75 F7           jne codecavetemiz.4011FC
00401205  .   59              pop ecx
00401206  .   8A5D FF         mov bl,byte ptr ss:[ebp-1]
00401209  .   80FB 01         cmp bl,1
0040120C  .v  7C 07           jl codecavetemiz.401215
0040120E  .   24 F0           and al,F0
00401210  .   C0E8 04         shr al,4
00401213  .v  EB 02           jmp codecavetemiz.401217
00401215  >   24 0F           and al,F
00401217  >   F6DB            neg bl
00401219  .   885D FF         mov byte ptr ss:[ebp-1],bl
0040121C  .   3C 09           cmp al,9                          9:'\t'
0040121E  .v  7E 02           jle codecavetemiz.401222
00401220  .   04 07           add al,7
00401222  >   04 30           add al,30
00401224  .   8807            mov byte ptr ds:[edi],al
00401226  .   47              inc edi
00401227  .   46              inc esi
00401228  .^  E2 C2           loop codecavetemiz.4011EC
0040122A  .   C9              leave
0040122B  L.  C2 0400         ret 4
```

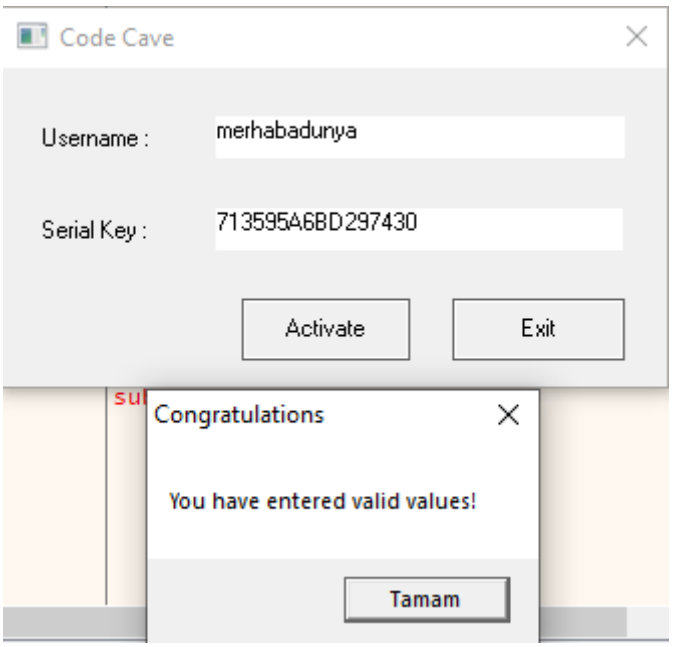
Şekil 6.51. Şifre çözme alt rutini

Şifre çözme işlemi bittikten sonra Dump sekmesindeki 00403114 adresinden itibaren oluşan değişiklikler Şekil 6.52'de gösterildiği gibidir.



Şekil 6.52. Şifre çözme işleminden sonra oluşan değişiklikler

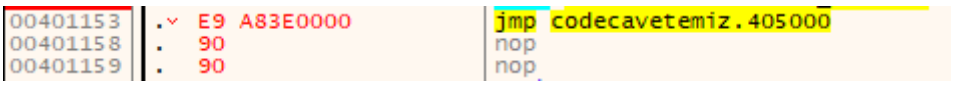
Kullanıcı adı olarak "merhabadunya", seri anahtarı olarak "9856756757" değeri girilmiş, şifreleme işleminde kullanıcı adından şifrelenmiş "êâõïæåæãòéþæ...." seri anahtarı türetilmiş ve bu seri anahtarına şifre çözme işlemi uygulanarak gerçek seri anahtarı olan "713595A6BD297430" bulunmuştur. "merhabadunya" kullanıcı ismine özel üretilen seri anahtarının denemesi yapıldığında Şekil 6.53'te gösterildiği gibi başarılı bir sonuç elde edilmiştir.



Şekil 6.53.Seri anahtarı ile deneme ve başarılı sonuç

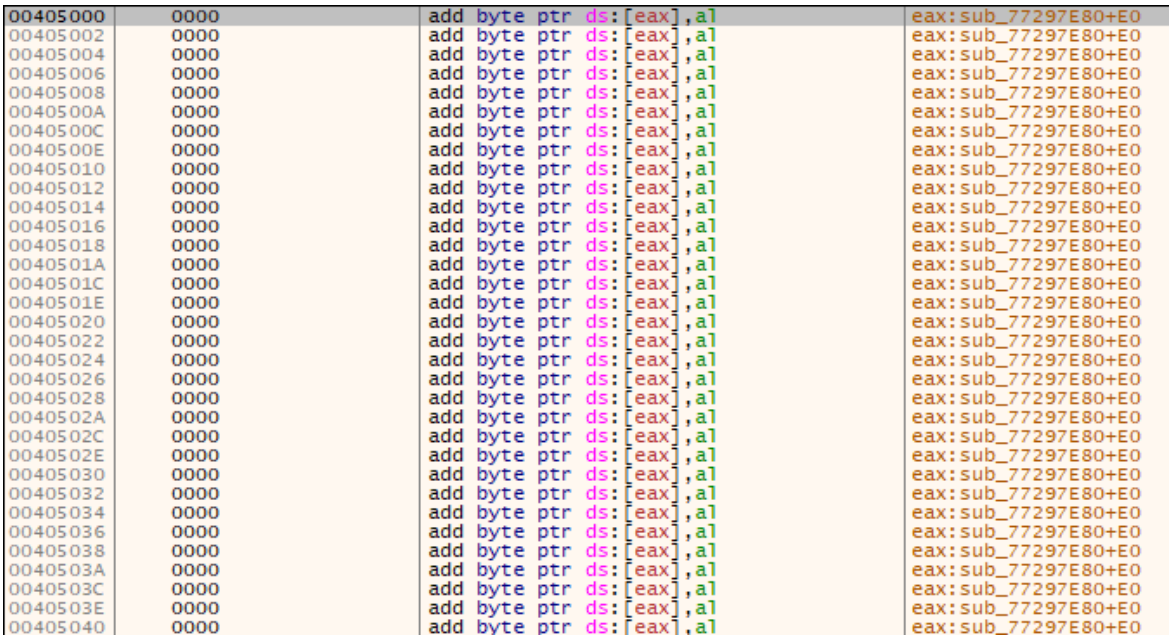
6.2.3 Kod mağarası kullanarak seri anahtarı oluşturma

Buradaki asıl amaç kod mağarası kullanılarak girilen her kullanıcı adı için üretilen seri anahtarını ekrana bastırmaktır. 0040114E adresinden çağrılan alt rutinin şifre çözme işlemi bittikten sonra program 00401153 adresinden devam etmektedir. Seri anahtarı girdisi yapılmasına gerek kalmayacağından seri anahtarı girdisine bağlı getDlgItemTextA işlevine müdahale etmek sorun çıkarmayacaktır. Kod mağarasının başlangıcı 00405000 olarak bilindiğinden 00401153 adresindeki talimat Şekil 6.54.'te gösterildiği gibi jmp 0x00405000 olarak değiştirilerek kod akışının kod mağarasından devam edilmesi sağlanmıştır.



Şekil 6.54. Kod akışı değiştirme

00405000 adresinden başlayan yeni bölümün görüntüsü Şekil 6.55'te gösterildiği gibidir.



Şekil 6.55. Yeni bölümün görüntüsü

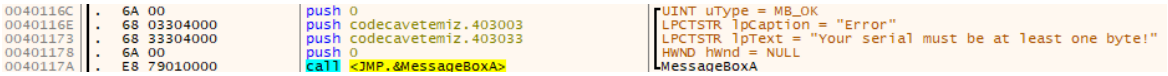
Mesaj kutusunun ekrana, girilen kullanıcı adı için üretilen seri anahtarını bastırması istendiğinden kod mağarasına gerekli talimatların eklenmesi gerekmektedir. 0040117A adresindeki MessageBoxA fonksiyon çağrısı kullanılarak ekrana bastırma işlemi gerçekleşeceğinden MessageBoxA fonksiyonunun sözdizimini bilmek önemlidir. MessageBoxA fonksiyonunun C++ dilindeki sözdizimi Şekil 6.56'da gösterildiği gibidir.

```
C++

int MessageBoxA(
  [in, optional] HWND   hWnd,
  [in, optional] LPCSTR lpText,
  [in, optional] LPCSTR lpCaption,
  [in]           UINT   uType
);
```

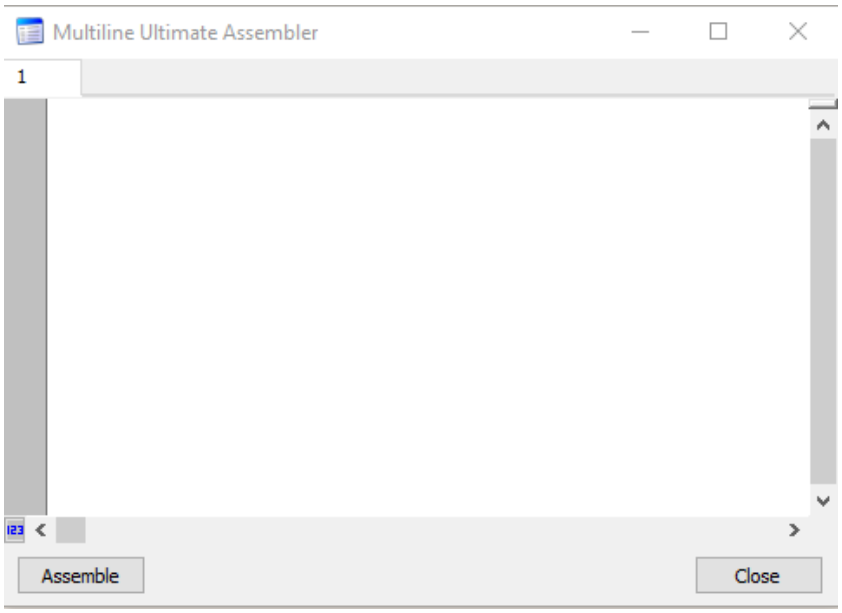
Şekil 6.56. MessageBoxA fonksiyonunun C++ dilindeki sözdizimi

Programdaki MessageBoxA fonksiyonunun hata ayıklayıcıda gösterimi Şekil 6.57'de gösterildiği gibidir. Sözdiziminin C++ diline göre ters bir şekilde gözükmesi stack kullanımındaki "last in first out / LIFO" yani son giren ilk çıkar yaklaşımından kaynaklıdır.



Şekil 6.57. MessageBoxA fonksiyonunun hata ayıklayıcıda gösterimi

Talimatları teker teker eklememek için Şekil 6.58'de gösterilen "Multiline Ultimate Assembler" isimli eklenti kullanılmaktadır.



Şekil 6.58. Multiline ultimate assembler eklentisi

Multiline Ultimate Assembler isimli eklenti ile talimatları yazmadan önce bir başlangıç adresi girmek gerekmektedir. Bu durumda başlangıç adresi <00405000> olacaktır.

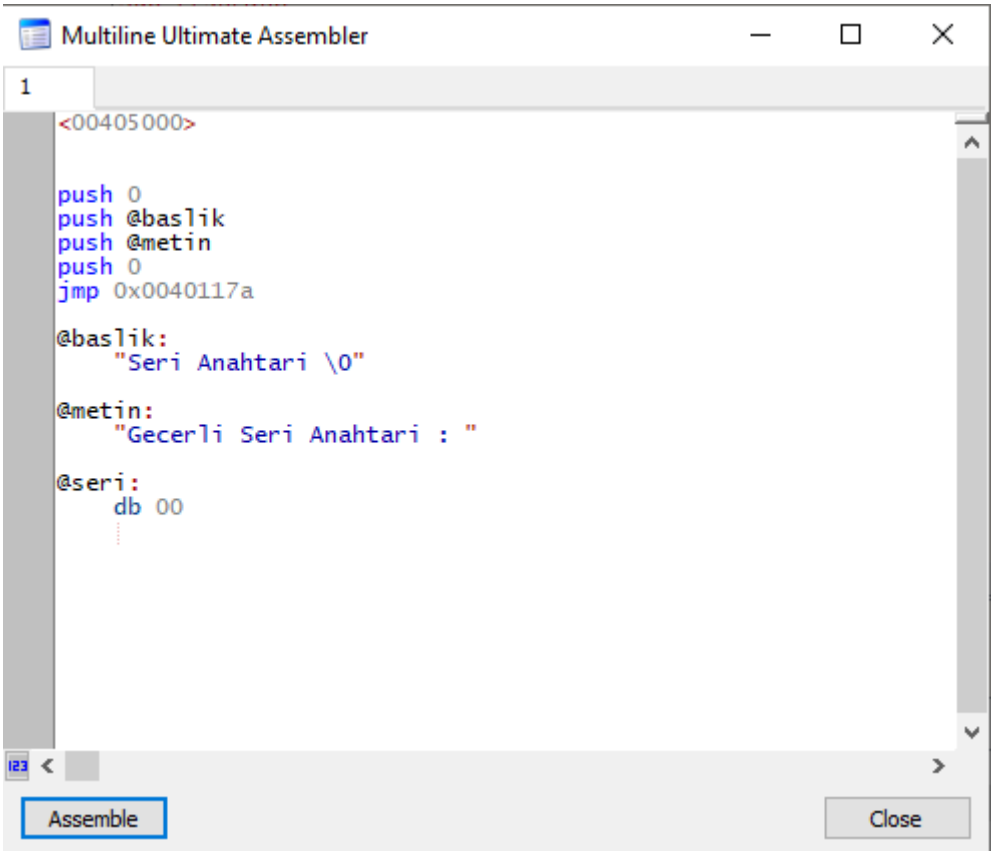
İkinci olarak “uType” parametresinin belirlenmesi gerekmektedir. Bu parametre mesaj kutusunun içeriği ve davranışı yani mesaj kutusunun stilidir.

Üçüncü olarak “lpCaption” parametresi ile mesaj kutusunun başlığı belirlenmelidir.

Dördüncü olarak “lpText” parametresi ile mesaj kutusunda görüntülenecek mesaj belirlenmelidir.

Beşinci olarak “hWnd” parametresi ile mesaj kutusunun sahip olduğu pencere belirlenmelidir.

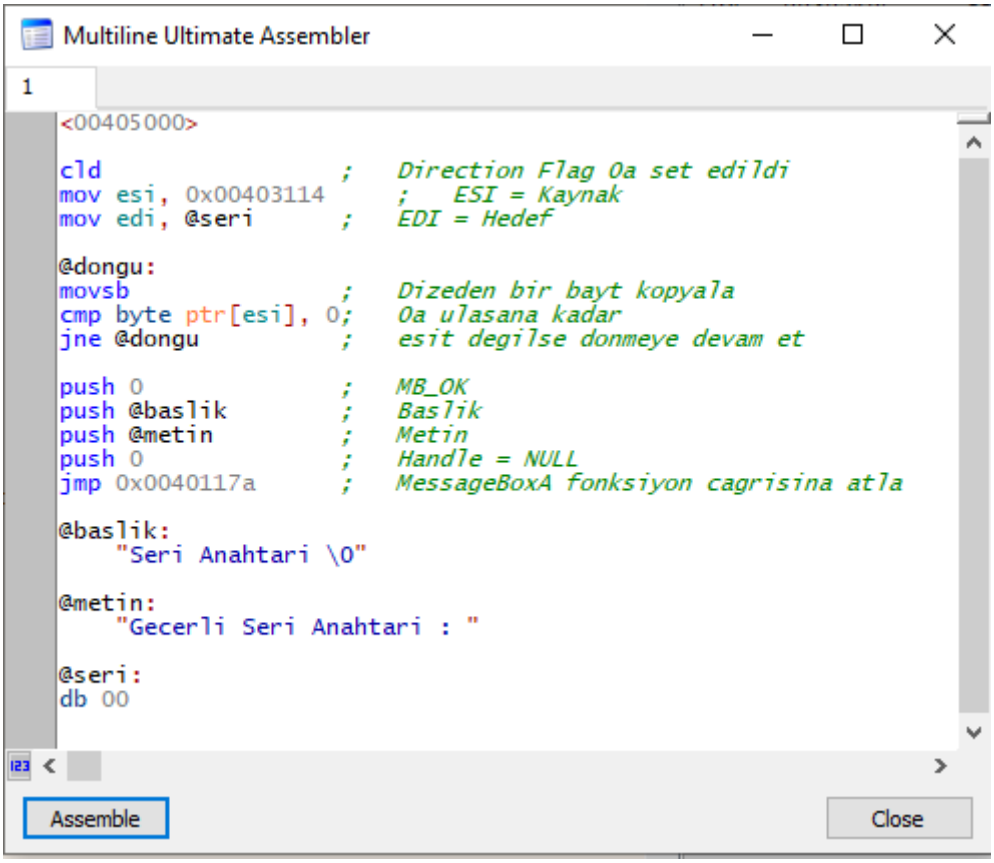
Şekil 6.59’da görüldüğü gibi tüm parametreler belirlenmiştir. @baslik dizesinde, sonlandırmayı belirtmek için “/0” ifadesi kullanılmıştır. @metin dizesinin sonuna sıfır eklenmemiştir. Bunun nedeni, seri anahtarını metnin devamına eklemek içindir. Mesaj kutusu bu dizenin başından başlayacak ve sıfıra ulaşana kadar devam edecektir. Bu şekilde, mesaj kutusu metni görüntülerken, eklediğimiz dizeye girmeye devam edecek ve doğru seri anahtarı görüntülenene kadar durmayacaktır. Bunun için dizenin sonuna seri anahtarını manuel olarak eklemek ve bunu sıfır ile bitirmek gereklidir. Seri anahtarını kopyalamak için bir alan oluşturulmuş ve dize “@seri” olarak tanımlanmıştır.



Şekil 6.59. Parametreler

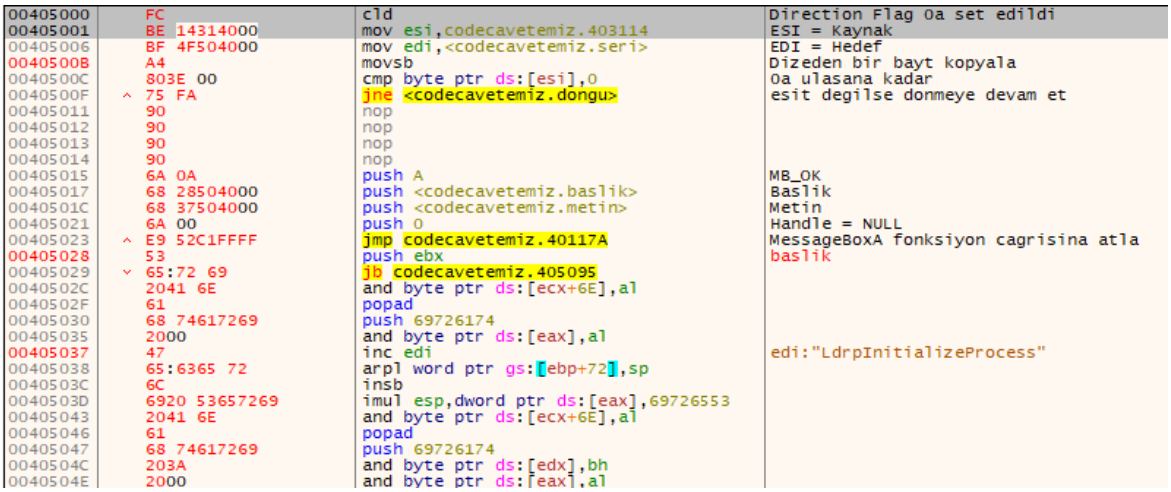
db komutu, veriler için bellek ayırma işleminde kullanılır. Seri anahtarının uzunluğu bilinmediği için bir bayt eklenmiştir. Seri anahtarının baytlarının dinamik olarak eklenmesi gerekmektedir.

Şekil 6.60.'ta tamamlanmış talimatlar görülmektedir. Dize komutlarının dizedeki bir sonraki baytı almak için bellek adreslerini otomatik olarak artırması için Cld komutu ile Direction Flag 0'a set edilmiştir. Dizi talimatı kaynak işaretçisi (ESI) 403114 adresini temsil etmekte ve dizi talimatı hedef işaretçisi (EDI), seri anahtarını kopyalamak için oluşturulmuş "@seri" alanını temsil etmektedir. Seri anahtarının baytlarının dinamik olarak eklenmesi için bir döngü oluşturulmuştur. Movsb komutunun görevi DS:SI ikilisinin gösterdiği adresteki baytı ES:DI ikilisinin gösterdiği adrese aktarmaktır.



Şekil 6.60. Tamamlanmış talimatlar

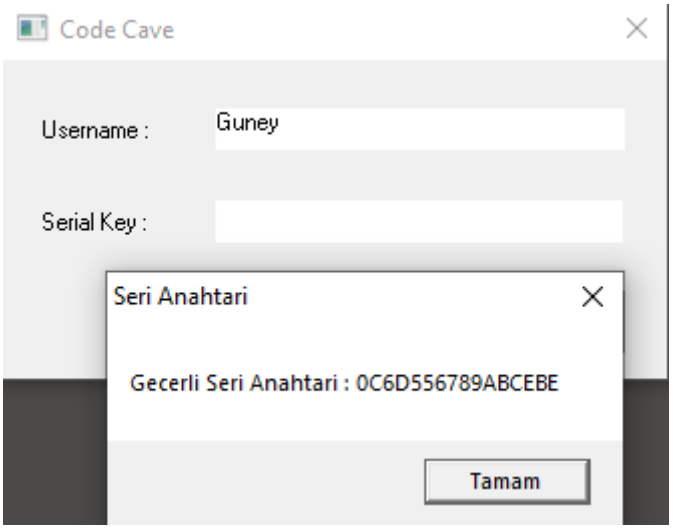
Assemble butonuna tıklandığında Şekil 6.61'de görüldüğü gibi 00405000 adresinden itibaren talimatlar eklenmiştir. Kod mağarasının sonu tanımlanmış dizelerle bittiğinden hata ayıklayıcı bunu analiz edememiş ve verileri kod olarak göstermiştir.



Şekil 6.61. Talimatların eklenmiş hali

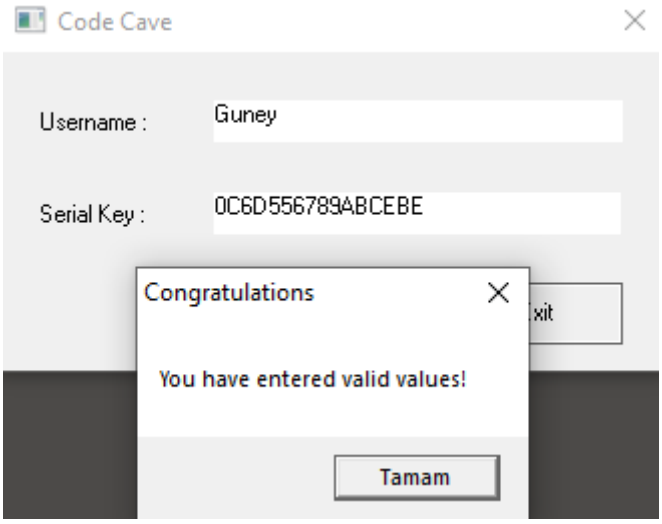
Yapılan değişiklikleri kaydetmek ve yeni bir çalıştırılabilir dosya oluşturmak için File sekmesinden Patch file seçeneği seçilmiş ve açılan pencereden Patch File butonuna tıklanmıştır. Yeni dosya cckeygen.exe ismiyle kaydedilmiştir.

Yeni dosya çalıştırıldığında ve bir kullanıcı ismi girilip Activate butonuna tıklandığında, ekrana Şekil 6.62.'de gösterilen mesaj kutusu belirmektedir. Bu mesaj kutusunun içeriği, @metin olarak tanımlanan dize ve girilen kullanıcı adı için geçerli olan seri anahtarıdır.



Şekil 6.62. Geçerli seri anahtarı mesaj kutusu

Örneğin en başındaki hiçbir tersine mühendislik işlemlerine uğramamış Codecave.exe isimli programa bu bilgilerin girişi yapıldığında Şekil 6.63'te gösterilen doğru değerlerin girildiğine dair bir mesaj kutusu ekrana gelmektedir.



Şekil 6.63. Doğru değerlerin girildiğine dair mesaj kutusu

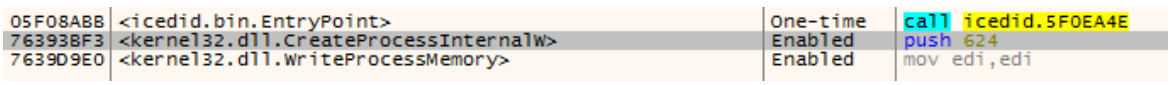
6.3 Örnek 3

Bu çalışmada, çevrimiçi bankacılık oturumlarından kimlik bilgilerini çalmak ve sahte işlemler başlatmak için man-in-the-browser saldırılarını kullanan ve son zamanlarda özellikle fidye yazılımları olmak üzere diğer kötü amaçlı yazılımlar için bir dağıtım mekanizması olarak kullanılan, paketlenmiş ve işlem enjeksiyonu yöntemini kullanan IcedID (BokBot) isimli bankacılık truva atı incelenmiştir.

VMware ile oluşturulan yalıtılmış sanal makinada x64dbg isimli hata ayıklayıcı ile zararlı yazılım örneğinin oluşturduğu süreçler incelenmiş, PE-Bear programı ile eşleme işlemleri yapılmış ve yine x64dbg isimli hata ayıklayıcı ile paketten çıkarma işlemleri yapılmış ve IDA Pro ile yüzeysel bir statik analiz gerçekleştirilmiştir.

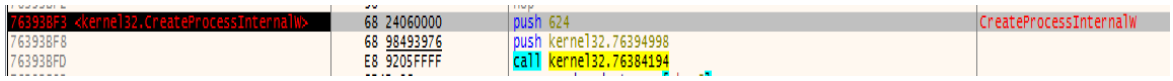
6.3.1 İşlemin yaratılma süreci, bölümler ve eşleme işlemleri

X64dbg hata ayıklayıcısı ile açılan zararlı yazılımda işlemin yaratılma sürecini gözlemlemek için CreateProcessInternalW çağrısına ve işlemin bellek alanına yazılacak verileri gözlemlemek için WriteProcessMemory çağrısına kesme noktaları yerleştirilmiştir. Kesme noktaları Şekil 6.64.'te gösterilmektedir.



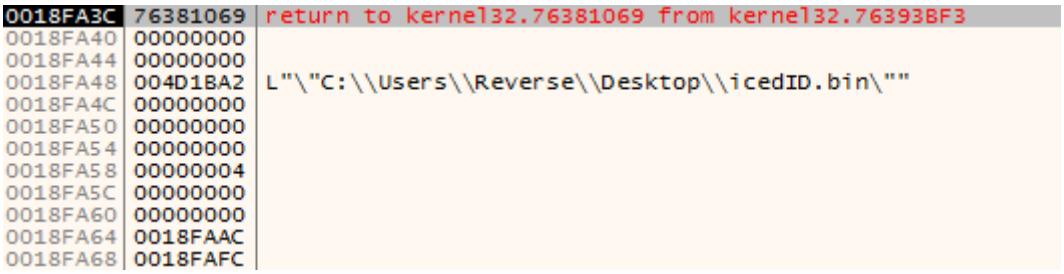
Şekil 6.64. CreateProcessInternalW ve WriteProcessMemory kesme noktaları

Zararlı yazılım hata ayıklayıcısında yürütüldüğünde ilk olarak giriş noktasında kesmeye noktasına isabet edilmiştir, yürütülmeye devam edildiğinde ise Şekil 6.65.'te görülen CreateProcessInternalW çağrısında kesme noktasına isabet edilmiştir.



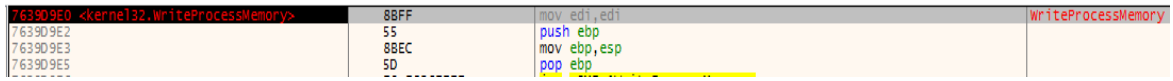
Şekil 6.65. CreateProcessInternalW çağrısında kesme noktası

Şekil 6.66.'da yığın argümanları gösterilmekte ve icedId.bin zararlı yazılımının kendi kopyasının başlatıldığı gösterilmektedir.



Şekil 6.66. Yığın argümanları

Zararlı yazılımın yürütülmesine devam edildiğinde WriteProcessMemory çağrısında kesme noktasına isabet edildiği Şekil 6.67.'de gösterilmiştir.



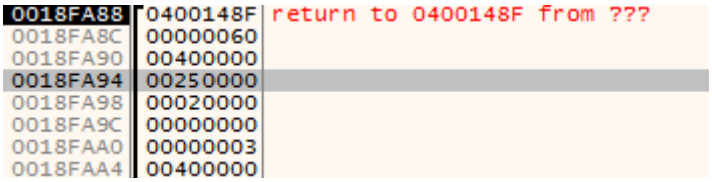
Şekil 6.67. WriteProcessMemory çağrısında kesme noktası

Buradaki amaç WriteProcessMemory ile hafızaya yazılan bir PE dosyasının varlığını tespit etmektir. Şekil 6.68.'de WriteProcessMemory sözdizimi gösterilmektedir.

```
BOOL WriteProcessMemory(
  [in]  HANDLE  hProcess,
  [in]  LPVOID  lpBaseAddress,
  [in]  LPCVOID lpBuffer,
  [in]  SIZE_T  nSize,
  [out] SIZE_T  *lpNumberOfBytesWritten
);
```

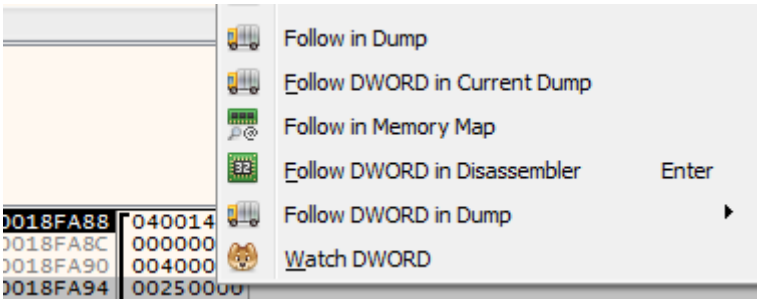
Şekil 6.67. WriteProcessMemory sözdizimi

Şekil 6.69.'da gösterilen yığın penceresine bakıldığında ilk kayıt dönüş adresi ve 4. Kayıt lpBuffer yani işleme yazılan verilerin arabelleği işaretçisidir.



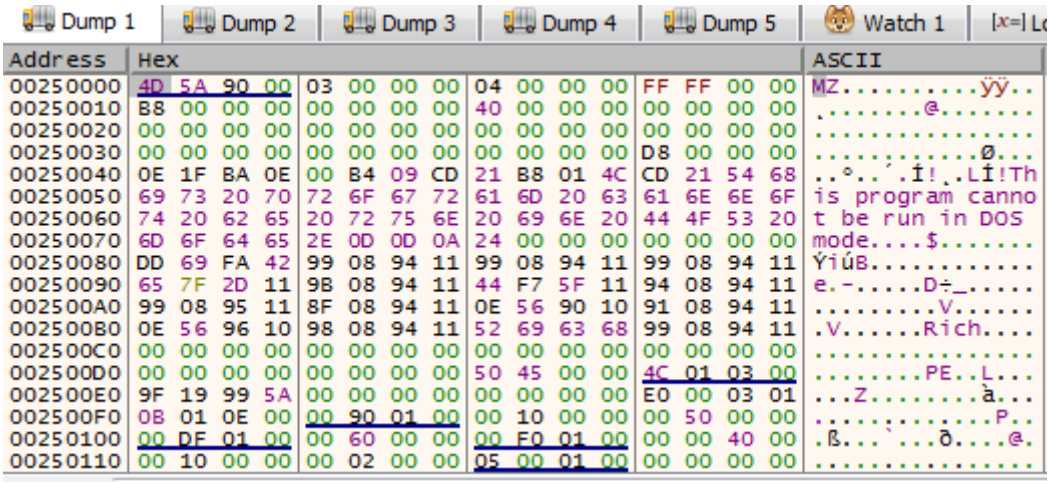
Şekil 6.69. Yığın Penceresi

Şekil 6.70.'te 4. kaydın adresine sağ tıklanıp "Follow DWORD in Current Dump" seçeneği seçilmiştir.



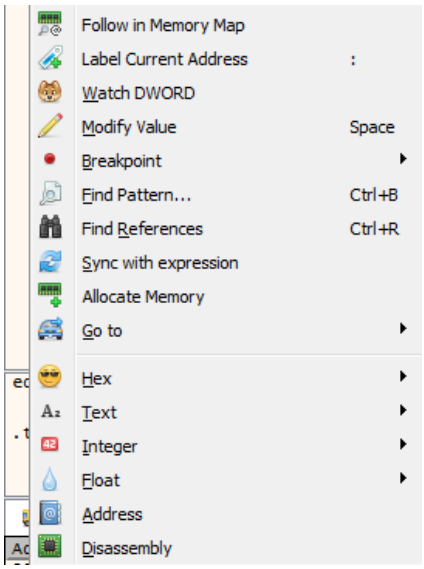
Şekil 6.70. Follow DWORD in current dump seçeneği

Şekil 6.71.'de Dump sekmesi gösterilmektedir. 4D 5A (ASCII'de MZ) sihirli sayısı gözlemlenmiş ve hafızaya yazılan bir PE dosyasının varlığı ortaya çıkarılmıştır.



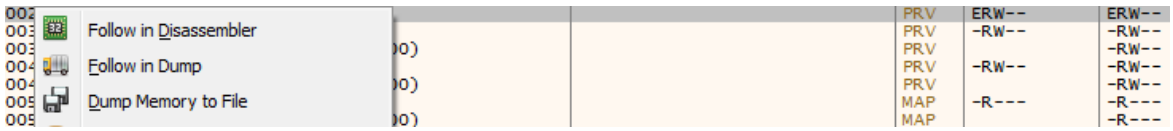
Şekil 6.71. Dump sekmesi

Pe dosyasını dump etmek için, dump sekmesindeki 00250000 adresine Şekil 6.72.'de gösterildiği gibi sağ tıklanır ve "Follow in Memory Map" seçeneği seçilir.



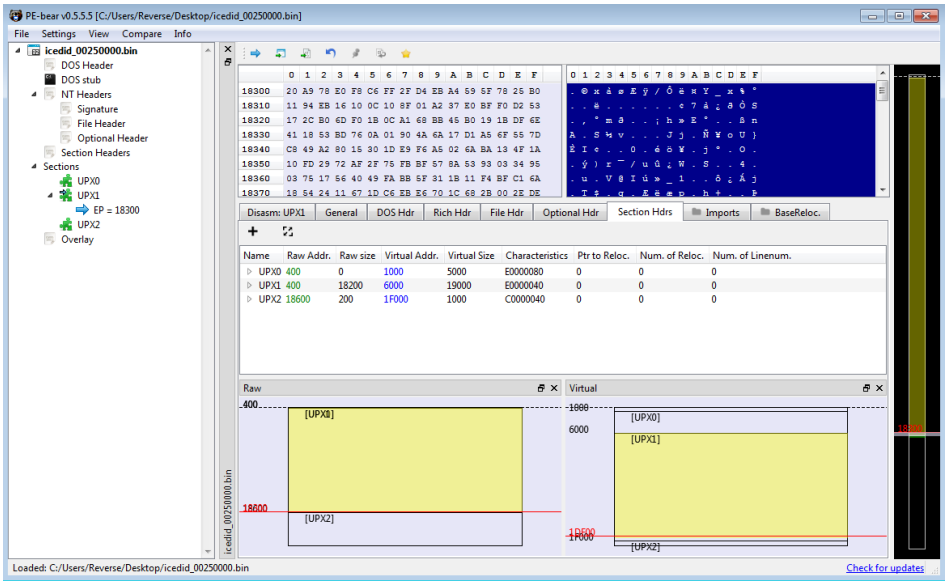
Şekil 6.72. Follow in Memory Map seçeneği

Gelen ekranda Şekil 6.73.'te gösterilen "Dump Memory to File" seçeneği seçilmiş ve yeni dosya icedid_00250000.bin ismi ile kaydedilmiştir.



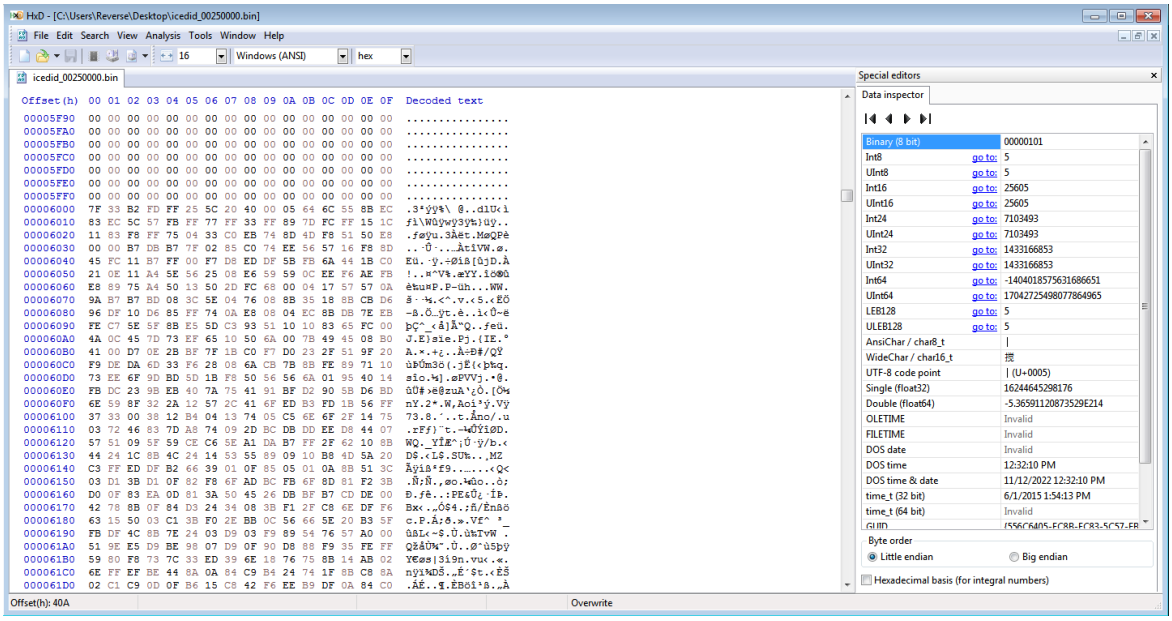
Şekil 6.73. Dump memory to file seçeneği

icedid_00250000.bin dosyasının PE-bear programı ile açılmış hali Şekil 6.74.'te gösterilmektedir.



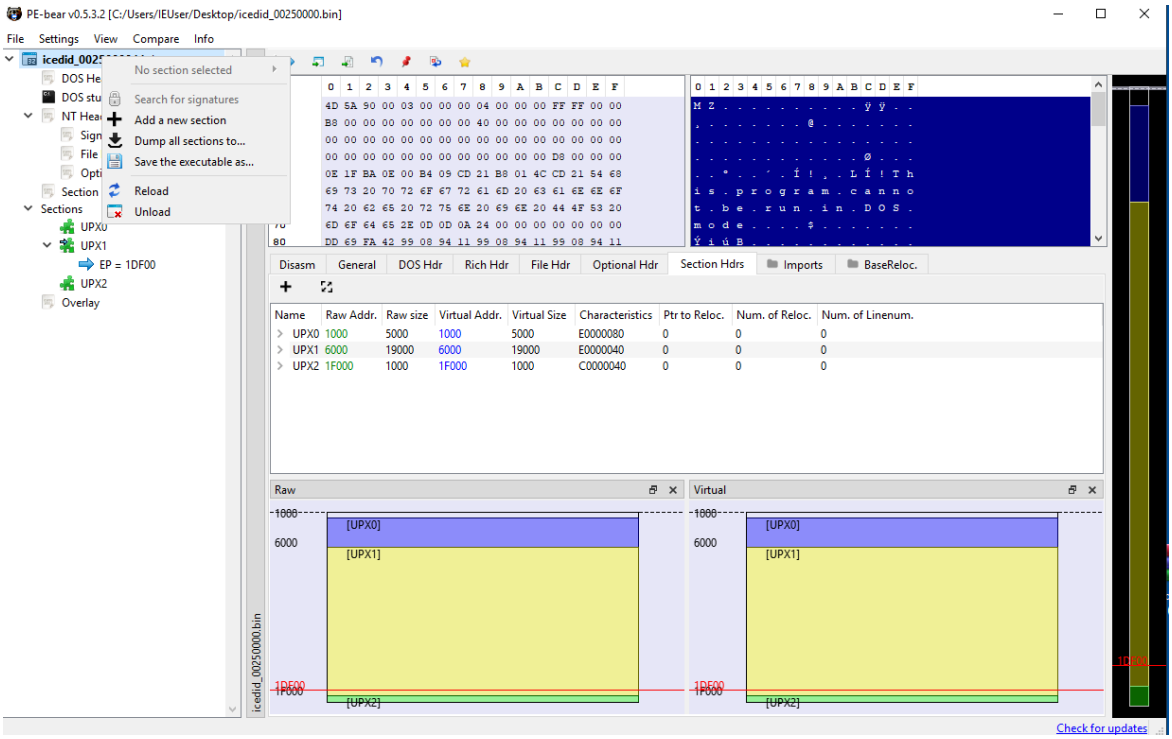
Şekil 6.74. icedid_00250000.bin dosyasının PE-bear programı ile açılmış hali

Şekil 6.74.'te görüldüğü üzere Sections yani Bölüm tabloları ismi UPX olarak adlandırılmıştır. Bu, dosyanın UPX paketleyicisi ile paketlendiğini göstermektedir. UPX paketleyicisi bölümü geçersiz kılar, yeni bir bölüm oluşturur, bu bölümün içine kod yazar ve ardından yürütmeyi buraya aktarır. Dump edilen dosyanın eşlenmiş (mapped) veya eşlenmemiş (un-mapped) olduğunu anlamak gereklidir. Bunun için dosya HxD editör ile açılmış ve Virtual Address'in işaret ettiği bellek adresi olan 00006000 adresinden itibaren veri olup olmadığı kontrol edilmiştir. UPX paketleyicisinin doğası gereği ilk bölümde veri olmayacağından 00001000 adresi kontrol edilmemiştir. Şekil 6.75.'te HxD editörü ile açılmış dosya ve 00006000 adresinden itibaren içerdiği veri gösterilmektedir.



Şekil 6.75. HxD editörü ile açılmış dosya

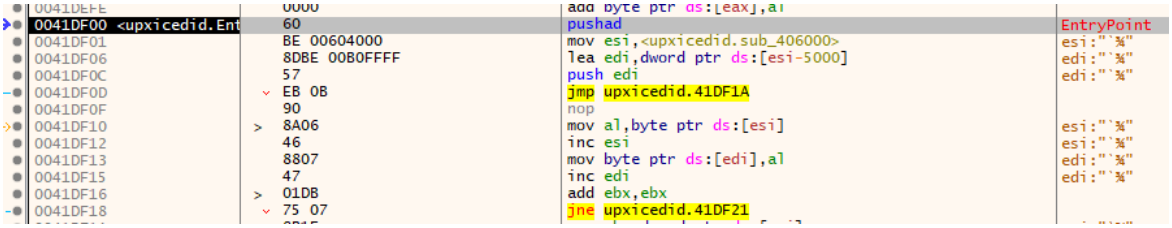
00006000 adresinden itibaren verinin olması ve öncesinde verinin olmaması bu dosyanın eşlenmiş (mapped) olduğunu göstermektedir. Bu nedenle tüm bölümlerin Raw Address'i Virtual Address'e, Raw Size'ı Virtual Size'a eşit olacak şekilde değiştirilmesi gerekmektedir. Şekil 6.76.'da bu adreslerin değiştirilmiş halleri gösterilmektedir. Yeni dosyayı kaydetmek için "Save the executable as…" seçeneği seçilmiş ve yeni dosya upxicedid.exe olarak kaydedilmiştir.



Şekil 6.76. Save the executable as… seçeneği

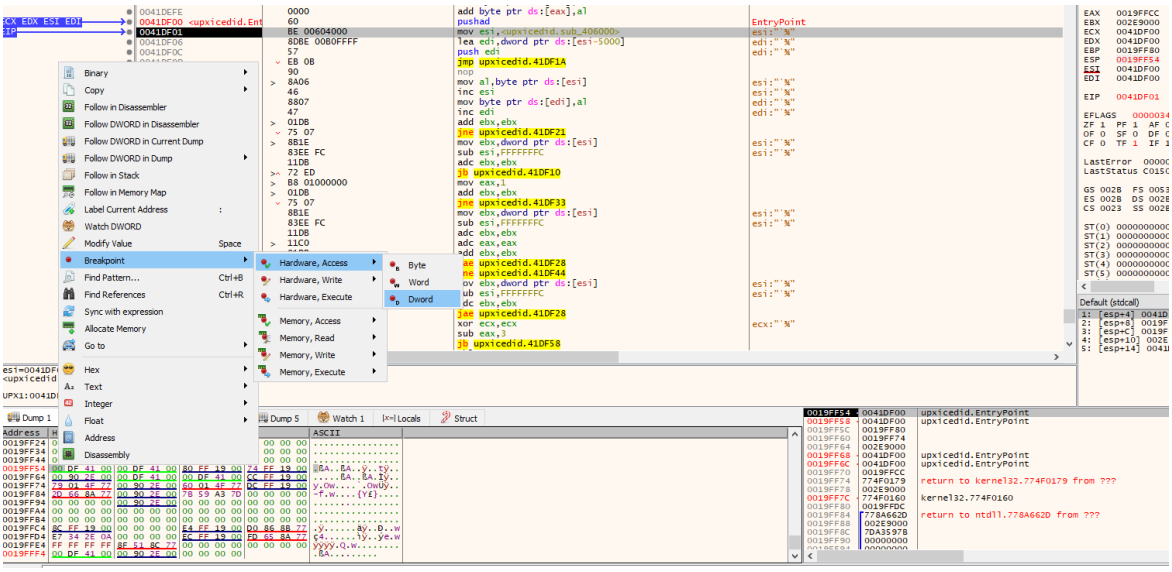
6.3.2 Paketten çıkarma ve içe aktarımların düzeltilmesi

X64dbg ile açılan upxicedid.exe dosyası yürütülmeye başlandığında giriş noktasındaki kesme noktasına isabet etmiştir. Şekil 6.77.'de giriş noktasının kod bloğu gösterilmektedir.



Şekil 6.77. Giriş noktasının kod bloğu

Giriş noktası pushad talimatı ile başlamaktadır. UPX ile paketlenmiş dosyanın orijinal giriş noktasını (OEP) bulmak için "ESP Trick" yöntemi kullanılmıştır. Bunun için tek adımda yürütme ile bir sonraki adresteki talimata geçilmiştir. Şekil 6.78.'de ESP yazmacının gösterdiği 0019FF54 adresi dump sekmesinde gösterilmiş ve bir donanım kesme noktası yerleştirilmiştir.

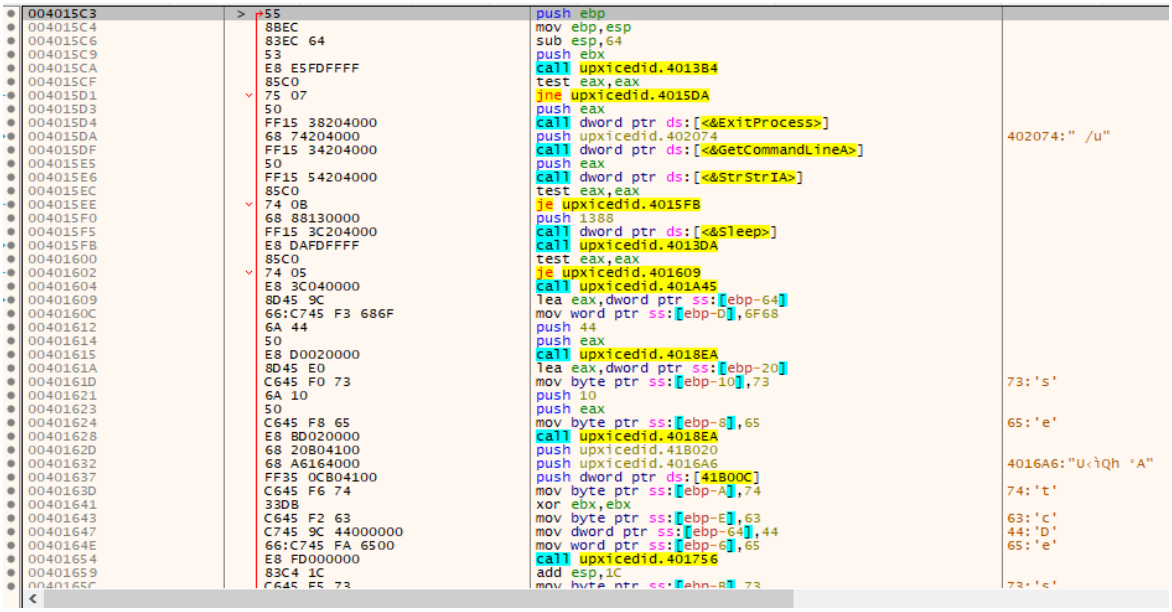


Şekil 6.78. Donanım kesme noktası eklenmesi

Dosya yürütülmeye devam edildiğinde popad talimatının işletilmesiyle, yerleştirilen donanım kesme noktası devreye girmiş ve programın yürütülmesi Şekil 6.79.'da gösterilen 0041E076 adresinde kesmeye uğramıştır.

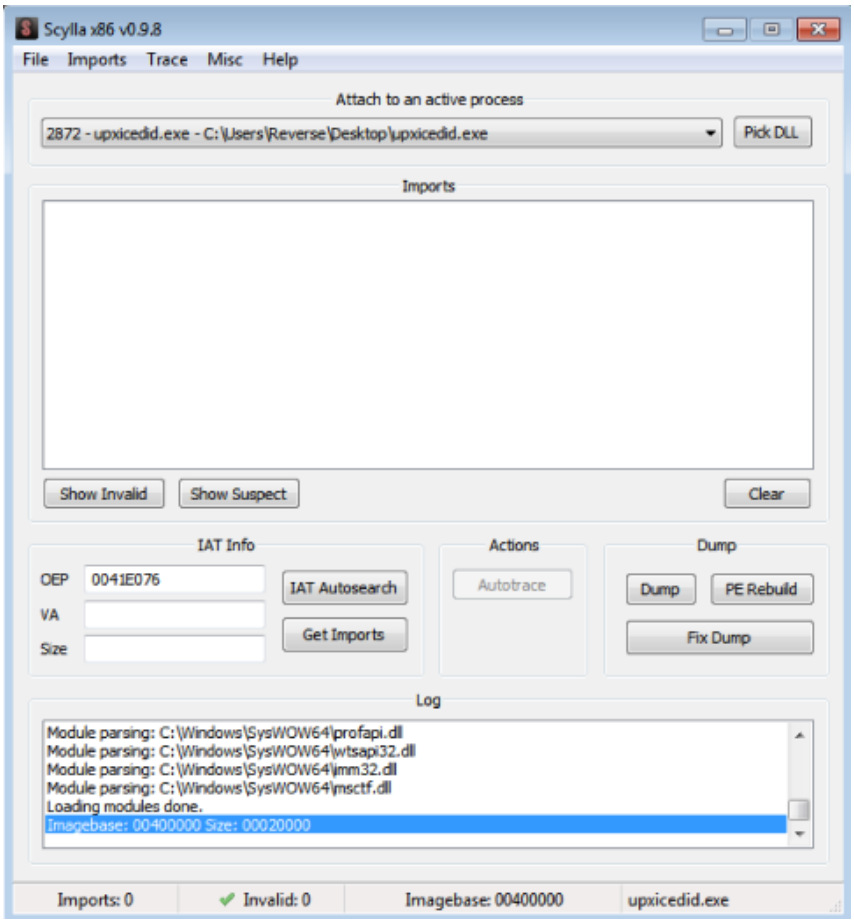
Şekil 6.79. 0041E076 adresinde kesmeye uğrama

0041E083 adresindeki koşulsuz dallanma (jmp) ile programın Şekil 6.80'de gösterilen orijinal giriş noktasına ulaşılmaktadır.



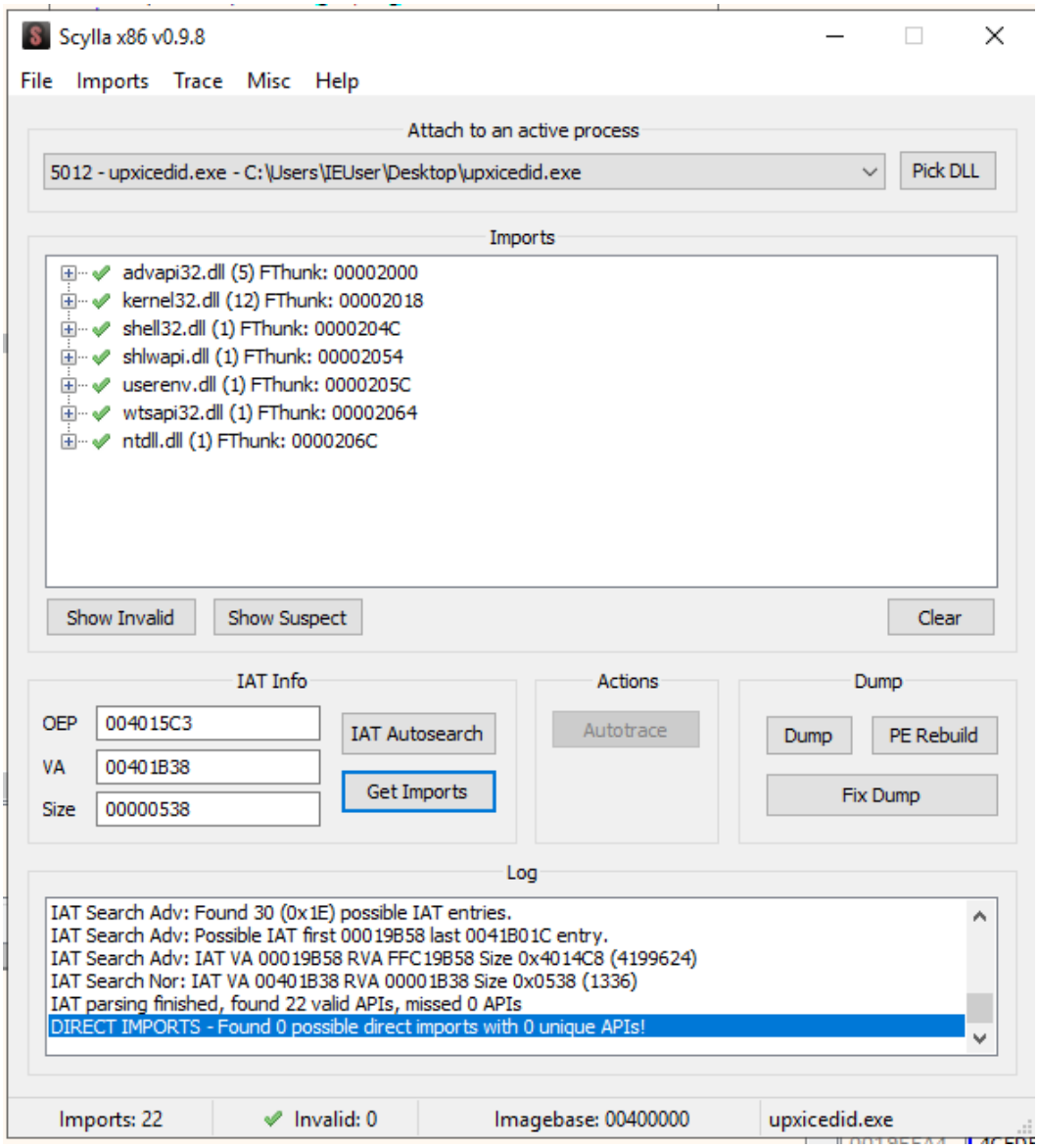
Şekil 6.80. Orijinal giriş noktası

Orijinal giriş noktası bulunan zararlı yazılımı dump etmek için Scylla isimli eklenti kullanılmıştır. Şekil 6.81.'de Scylla eklentisinin arayüzü gösterilmektedir.



Şekil 6.81. Scylla eklentisinin arayüzü

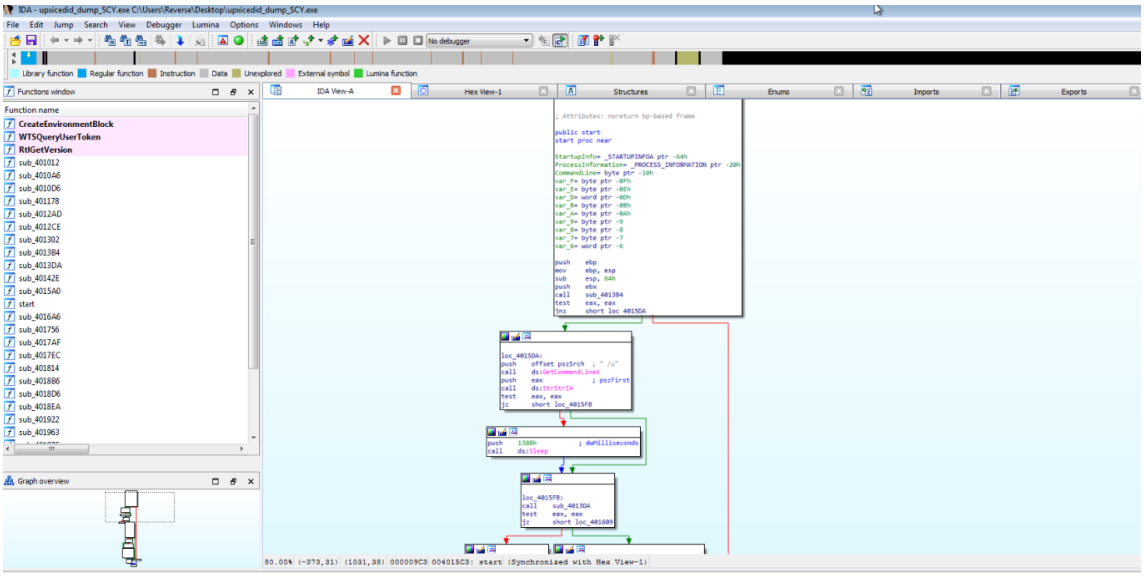
OEP adresi 004015C3 olarak değiştirilmiş ve dump butonuna tıklanarak upxicedid_dump.exe ismiyle kaydedilmiştir. Yeni dosyanın Import Address Table'ının düzeltilmesi gerekmektedir. IAT Autosearch butonuna tıklanmış ve ardından Get Imports seçeneğine tıklanmıştır. Şekil 6.82.'de içe aktarımlar ve içe aktarım sayısının 22 olduğu gösterilmektedir. Fix Dump Seçeneği seçilip kaydedilen dosyanın Import Address Table'ı ve içe aktarılanları düzeltilmiş ve upxicedid_dump_SCY.exe olarak kaydedilmiştir.



Şekil 6.82. İçe aktarımlar ve içe aktarım sayısı

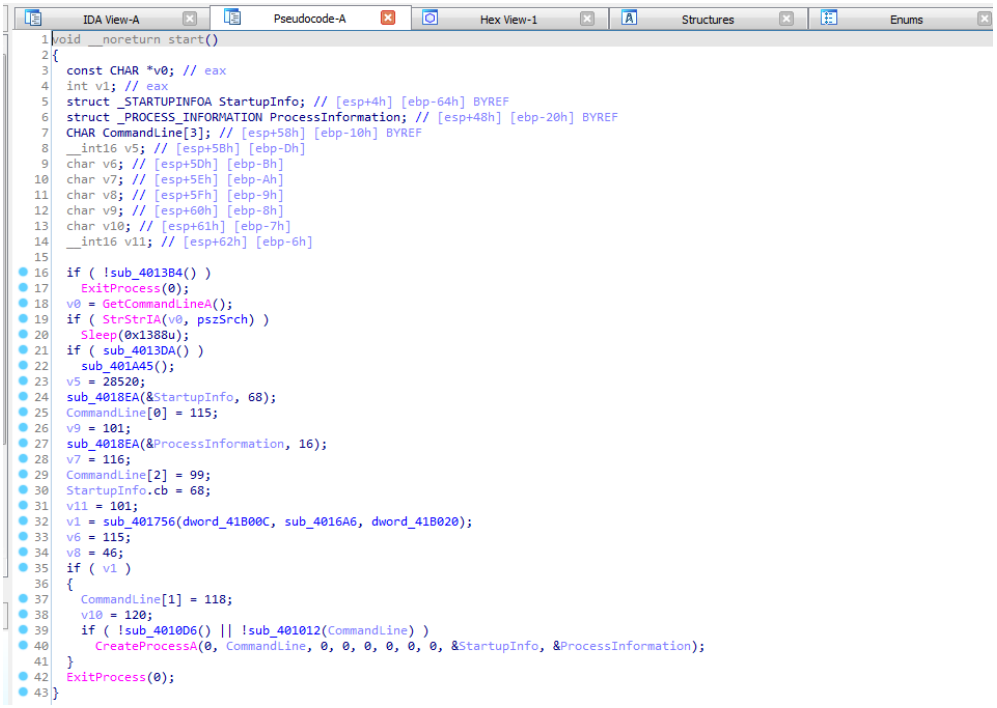
6.3.3 IDA Pro ile statik analiz ve içe aktarımların tekrar düzeltilmesi

Statik analiz için IDA Pro ile açtığımız upxicedid_dump_SCY.exe isimli zararlı yazılımın kod akışı ve işlevleri assembly dilinde grafiksel olarak Şekil 6.83.'te gösterilmektedir.



Şekil 6.83. IDA pro ile açılmış zararlı yazılım

F5 tuşuna tıklanarak assembly kodlarının geri derlenmiş hali Şekil 6.84'te gösterilmektedir.



```
1  void __noreturn start()
2  {
3    const CHAR *v0; // eax
4    int v1; // eax
5    struct _STARTUPINFOA StartupInfo; // [esp+4h] [ebp-64h] BYREF
6    struct _PROCESS_INFORMATION ProcessInformation; // [esp+48h] [ebp-20h] BYREF
7    CHAR CommandLine[3]; // [esp+58h] [ebp-10h] BYREF
8    __int16 v5; // [esp+5Bh] [ebp-Dh]
9    char v6; // [esp+5Dh] [ebp-Bh]
10   char v7; // [esp+5Eh] [ebp-Ah]
11   char v8; // [esp+5Fh] [ebp-9h]
12   char v9; // [esp+60h] [ebp-8h]
13   char v10; // [esp+61h] [ebp-7h]
14   __int16 v11; // [esp+62h] [ebp-6h]
15
16   if ( !sub_4013B4() )
17     ExitProcess(0);
18   v0 = GetCommandLineA();
19   if ( StrStrIA(v0, pszSrch) )
20     Sleep(0x1388u);
21   if ( sub_4013DA() )
22     sub_401A45();
23   v5 = 28520;
24   sub_4018EA(&StartupInfo, 68);
25   CommandLine[0] = 115;
26   v9 = 101;
27   sub_4018EA(&ProcessInformation, 16);
28   v7 = 116;
29   CommandLine[2] = 99;
30   StartupInfo.cb = 68;
31   v11 = 101;
32   v1 = sub_401756(dword_41B00C, sub_4016A6, dword_41B020);
33   v6 = 115;
34   v8 = 46;
35   if ( v1 )
36   {
37     CommandLine[1] = 118;
38     v10 = 120;
39     if ( !sub_4010D6() || !sub_401012(CommandLine) )
40       CreateProcessA(0, CommandLine, 0, 0, 0, 0, 0, 0, &StartupInfo, &ProcessInformation);
41   }
42   ExitProcess(0);
43 }
```

Şekil 6.84. Zararlı yazılımın geri derlenmiş hali

sub_4013B4 isimli işleve çift tıklandığında Şekil 6.85.'te gösterilen geri derlenmiş kod bloğu ile karşılaşılmıştır.

```
1 int sub_4013B4()
2 {
3   int result; // eax
4
5   result = sub_4018B6();
6   if ( result )
7     result = sub_401178(result, dword_402078, 8, &dword_41B000) != 0;
8   return result;
9 }
```

Şekil 6.85. sub_4013B4 kod bloğu

sub_4018B6 işlevine çift tıklandığında karşılaşılan geri derlenmiş kod bloğu Şekil 6.86.'da gösterilmiştir.

```
1  struct _LIST_ENTRY *sub_4018B6()
2  {
3    struct _PEB *v0; // eax
4    struct _LIST_ENTRY *result; // eax
5    struct _LIST_ENTRY *v2; // eax
6    struct _LIST_ENTRY *v3; // eax
7
8    v0 = NtCurrentPeb();
9    if ( v0 && (v2 = v0->Ldr->InInitializationOrderModuleList.Flink) != 0 && (v3 = v2 - 2) != 0 )
10     result = v3[3].Flink;
11   else
12     result = 0;
13   return result;
14 }
```

Şekil 6.86. sub_4018B6 kod bloğu

sub_4018B6 işlevi Process Environment Block'a erişim yapmaktadır. Şekil 6.85.'de gösterilen sub_401178 işlevine çift tıklandığında Şekil 6.87.'de gösterilen kod bloğu gelmektedir.

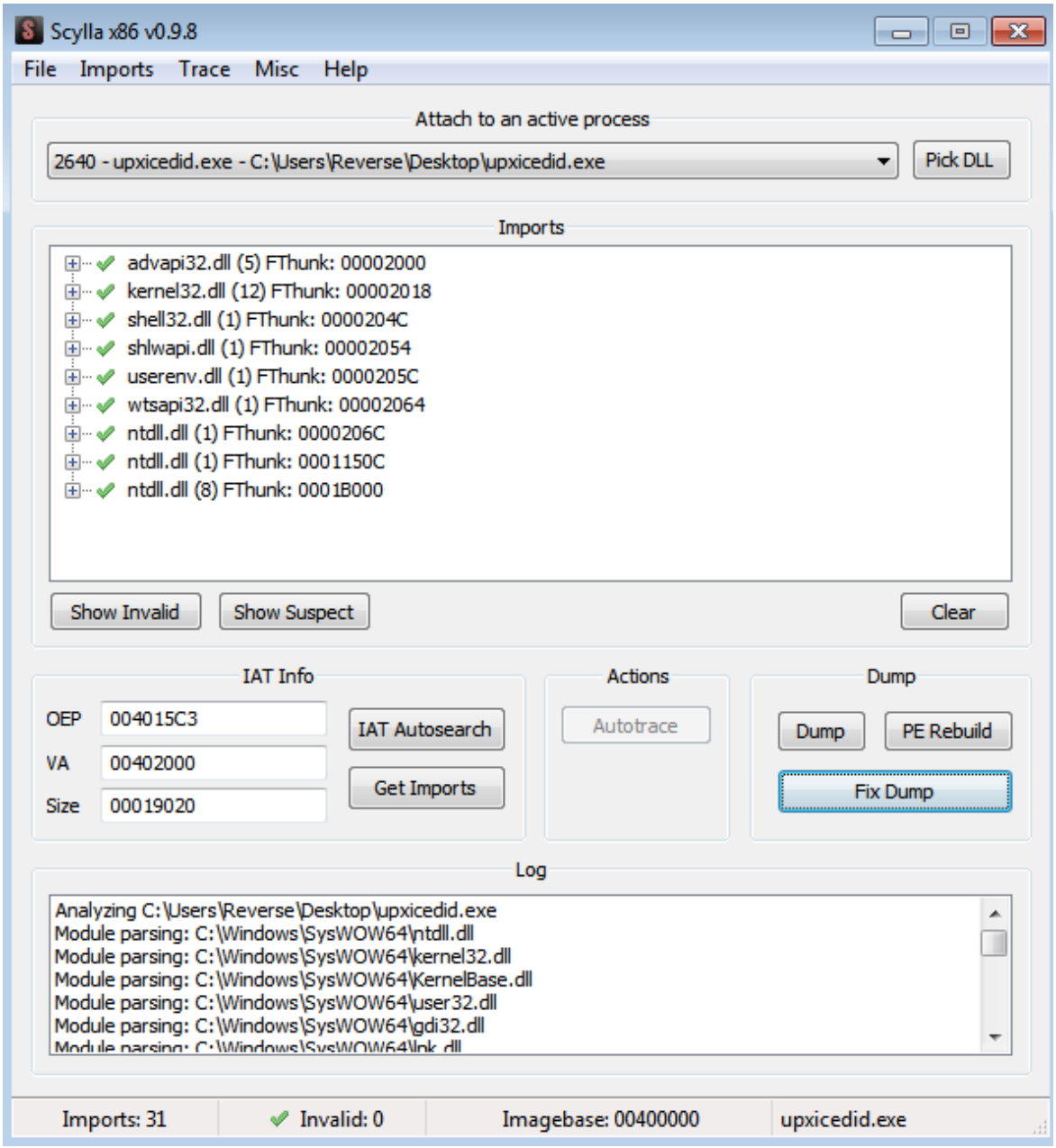
```
1  int __cdecl sub_401178(unsigned int a1, int a2, unsigned int a3, int a4)
2  {
3    unsigned int v4; // ecx
4    _DWORD *v5; // edx
5    int v6; // eax
6    _DWORD *v7; // esi
7    unsigned int v8; // eax
8    unsigned int v9; // edx
9    unsigned int v10; // ebx
10   unsigned int v11; // edi
11   int v12; // ebp
12   int v13; // eax
13   unsigned __int8 *v14; // edx
14   int v15; // ecx
15   unsigned __int8 v16; // al
16   int v17; // edx
17   bool v18; // zf
18   int v20; // [esp+14h] [ebp-Ch]
19   unsigned int v21; // [esp+18h] [ebp-8h]
20   unsigned int v22; // [esp+1Ch] [ebp-4h]
21
22   v4 = a1;
23   v21 = a3;
24   if ( *(_WORD *)a1 != 23117 )
25     return 0;
26   v5 = (_DWORD *)(a1 + *(_DWORD *)(a1 + 60));
27   if ( (unsigned int)v5 < a1 )
28     return 0;
29   if ( (unsigned int)v5 >= a1 + 4096 )
30     return 0;
31   if ( *v5 != 17744 )
32     return 0;
33   v6 = v5[30];
34   if ( !v6 )
35     return 0;
36   v7 = (_DWORD *)(v6 + a1);
37   if ( v6 + a1 < a1 )
38     return 0;
39   v8 = a1 + v5[20];
40   if ( (unsigned int)v7 >= v8 )
41     return 0;
42   v9 = a1 + v7[7];
43   v10 = a1 + v7[8];
44   v11 = a1 + v7[9];
```

Şekil 6.87. sub_401178 kod bloğu

24. kod satırına bakıldığında “if (*(_WORD *)a1 != 23117) “ koşulu görülmektedir. 23117 decimal değerdir. Bu değer hexedecimal değere çevrildiğinde 0x5A4D yani ZM karakterlerini temsil etmektedir. Bu işlev, içerdiği veriye göre PE dosyasında bazı içe aktarımların dinamik olarak oluşturulmasını sağlamaktadır.

X64dbg ile dump işlemi ve Import Address Table düzeltilmesi adımında, dosyayı dump etmeden önce sub_4013B4 işlevini işletmek ve geri dönüş yaptıktan sonra düzeltme işlemini tamamlamak içe aktarımları etiketlendirme sürecinde kolaylık sağlayacaktır. Çünkü dinamik olarak oluşturulan içe aktarımlar da düzeltilecek ve etiketlenmiş bir şekilde statik analizde görüntülenecektir.

Şekil 6.88.’de sub_4013B4 işlevi işletildikten sonra değişen içe aktarım sayısının 31 olduğu gösterilmektedir.



Şekil 6.88. Değişen içe aktarım sayısı

Dump edilen zararlı yazılımın Import Address Table'ı ve içe aktarılanları düzeltilmiştir. IDA Pro ile analiz edilen zararlı yazılımın geri derlenmiş kod bloğu Şekil 6.89.'da gösterildiği gibidir.

```
void __noreturn start()
{
  const CHAR *v0; // eax
  int v1; // eax
  struct _STARTUPINFOA StartupInfo; // [esp+4h] [ebp-64h] BYREF
  struct _PROCESS_INFORMATION ProcessInformation; // [esp+48h] [ebp-20h] BYREF
  CHAR CommandLine[3]; // [esp+58h] [ebp-10h] BYREF
  __int16 v5; // [esp+5Bh] [ebp-Dh]
  char v6; // [esp+5Dh] [ebp-Bh]
  char v7; // [esp+5Eh] [ebp-Ah]
  char v8; // [esp+5Fh] [ebp-9h]
  char v9; // [esp+60h] [ebp-8h]
  char v10; // [esp+61h] [ebp-7h]
  __int16 v11; // [esp+62h] [ebp-6h]

  if ( !sub_4013B4() )
    ExitProcess(0);
  v0 = GetCommandLineA();
  if ( StrStrIA(v0, pszSrch) )
    Sleep(0x1388u);
  if ( sub_4013DA() )
    sub_401A45();
  v5 = 28520;
  sub_4018EA(&StartupInfo, 68);
  CommandLine[0] = 115;
  v9 = 101;
  sub_4018EA(&ProcessInformation, 16);
  v7 = 116;
  CommandLine[2] = 99;
  StartupInfo.cb = 68;
  v11 = 101;
  v1 = sub_401756(NtCreateUserProcess, (int)sub_4016A6, (int)dword_41B020);
  v6 = 115;
  v8 = 46;
  if ( v1 )
  {
    CommandLine[1] = 118;
    v10 = 120;
    if ( !sub_4010D6() || !sub_401012(CommandLine) )
      CreateProcessA(0, CommandLine, 0, 0, 0, 0, 0, 0, &StartupInfo, &ProcessInformation);
  }
  ExitProcess(0);
}
```

Şekil 6.89. Zararlı yazılımın geri derlenmiş kod bloğu

Kod bloğunda sayılarla değişken atamaları görülmektedir. Bu sayılar ASCII aralığındadır ve karşılıklarını karakter olarak görmek için o sayının üzerine gelip R tuşuna basmak yeterlidir. Anlamlandırılmış değişken atamaları Şekil 6.90.'da gösterildiği gibidir.

```
16  if ( !sub_4013B4() )
17    ExitProcess(0);
18  v0 = GetCommandLineA();
19  if ( StrStrIA(v0, pszSrch) )
20    Sleep(0x1388u);
21  if ( sub_4013DA() )
22    sub_401A45();
23  v5 = 'oh';
24  sub_4018EA(&StartupInfo, 68);
25  CommandLine[0] = 's';
26  v9 = 'e';
27  sub_4018EA(&ProcessInformation, 16);
28  v7 = 't';
29  CommandLine[2] = 'c';
30  StartupInfo.cb = 68;
31  v11 = 'e';
32  v1 = sub_401756(NtCreateUserProcess, (int)sub_4016A6, (int)dword_41B020);
33  v6 = 's';
34  v8 = '.';
35  if ( v1 )
36  {
37    CommandLine[1] = 'v';
38    v10 = 'x';
39    if ( !sub_4010D6() || !sub_401012(CommandLine) )
40      CreateProcessA(0, CommandLine, 0, 0, 0, 0, 0, 0, &StartupInfo, &ProcessInformation);
41  }
42  ExitProcess(0);
43}
```

Şekil 6.90. Anlamlandırılmış değişken atamaları

Yığına karakter atama ve o karakterlerden bir dize oluşturma üzerine kurulu obfuscation tekniği ile karşılaşılmıştır. Temsil ettikleri harfler ve oluşan dize aşağıda gösterilmiştir.

CommandLine[0] = s
CommandLine[1] = v
CommandLine[2] = c
v5 = oh
v6 = s
v7 = t
v8 = .
v9 = e
v10 = x
v11 = e

40. satırdaki CreateProcessA ile yeni bir işlem yaratılmaktadır. IcedID özel bir işlem enjeksiyonu yöntemi kullanmakta ve yeni bir svchost.exe kopyası oluşturmaktadır.

7. SONUÇ

Tez kapsamında tersine mühendislik ve süreçleri hakkında bilgi verilmiş, assembly dilinin kullanımı ve içeriği anlatılmış, PE dosya formatının ayrıntıları ve tersine mühendislik sürecinde önem arz eden diğer bilgilere değinilmiştir. Tersine mühendisliği önleme yöntemleri, bu yöntemlerin aşılması ve analiz safhaları 3 farklı örnek üzerinde gösterilmiştir.

İlk örnekte hata ayıklamayı önleme yöntemlerinin gerçek bir program üzerinde nasıl aşıldığı uygulamalı olarak gösterilmiştir. 4 farklı hata ayıklamayı önleme yöntemi kullanılmış ve bu yöntemleri farklı şekillerde aşmanın yolları ele alınmıştır. Yazılım geliştiricilerine, geliştirdikleri yazılımları tersine mühendislikten nasıl koruyabilecekleri ve bu hata ayıklamayı önleme yöntemlerinin yine tersine mühendislik ile nasıl aşıldıklarının gösterilmesi amaçlanmıştır.

İkinci örnekte kod mağaralarının kullanımı ile yazılım lisanslama sürecinde izlenebilecek yasal olmayan bir yaklaşım ele alınmış ve yazılım lisanslama süreci gösterilmiştir. Yeni bölüm ekleme, hata ayıklamayı önlemeden kurtulma, kod mağarası oluşturma ve seri anahtarı için kod bloğu ekleme işlemleri yapılmış ve programın girilen kullanıcı adına göre isteyeceği lisans anahtarının ekrana basılması sağlanmıştır.

Son örnekte ise çevrimiçi bankacılık oturumlarından kimlik bilgilerini çalan ve sahte işlemler başlatan IcedID (BokBot) isimli bankacılık truva atı incelenmiştir. Zararlı yazılımın işlem yaratma süreci ve bölümleri analiz edilmiş, eşleme işlemleri yapılmıştır. Paketten çıkarma yöntemi uygulanan ve ardından içe aktarılanları düzeltilen zararlı yazılım, IDA Pro ile temel seviyede analiz edilmiştir. Değişken atamaları anlamlandırıldıktan sonra hangi işleme enjekte edildiği ortaya konmuştur.

Yapılan çalışmalar, yazılım geliştiricilerinin yazılımlarını nasıl koruyabileceklerini öğrenmelerini sağlamayı ve kariyerine zararlı yazılım analizi ile devam etmek isteyenler için tersine mühendislik yöntemleri hakkında bir fikir edinme olanağı sunmayı ve yol gösterici olmayı hedeflemiştir. Literatürdeki boşluğu doldurması açısından yazılım korsanlarının yaklaşımları da gösterilmiş ve yazılım geliştiricilerinin dikkat etmesi gereken noktalar belirtilmiştir. Tersine mühendislik, tersine mühendisliği önleme ve aşma yöntemlerinin bir yazılım geliştiricisi tarafından öğrenilmesi, yazılım geliştirme sürecine büyük bir katkı sağlamanın yanı sıra yazılım korsanlığına karşı alınabilecek önlemlerin zamanla daha fazla ortaya çıkmasına vesile olacaktır.

KAYNAKLAR

[1] W. Wang, Introduction in *Reverse Engineering Technology of Reinvention,* Boca Raton, FL, USA: CRC Press, 2010 pp. *1–3*.

[2] *US Army Reverse Engineering Handbook MIL-HDBK-115A*, AL, USA, JUNE 2006

[3] R. Wong, Preparing to Reverse in *Mastering Reverse Engineering,* Brimingham, UK: Pack Publishing, 2018 pp. *9–11*.

[4] R. Blum, The IA-32 Platform in *Professional Assembly Language,* Indianapolis, USA: Wiley Publishing, 2005 pp. *25–26.*

[5] D. Kusswurm, X86-64 Core Architecture in *Modern X86 Assembly Language Programming,* Geneva,IL, USA: Apress Media, 2014 pp. *9–11.*

[6] B. Dang, A. Gazet and E. Bachaalany, X86 and X64 in *Practical Reverse Engineering,* Indianapolis, USA: John Wiley & Sons, 2014 pp. *13-17.*

[7] P. Yosifovich, A. Ionescu and D. A. Solomon, System Architecture in *Windows Internals Part 1,* Washington, USA: Microsoft Press, 2017 pp. *53-54.*

[8] E. Eilam, Antireversing Techniques in *Reversing Secrets of Reverse Engineering,* Indianapolis, USA: Wiley Publishing, 2005 pp. *332-333.*

[9] T. Shields, Anti-Debugging – A Developers View, Veracode Inc, USA, 2010.

[10] Jong-Wouk Kim, J. Bang and Mi-Jung Choi, "Defeating Anti-Debugging Techniques for Malware Analysis Using a Debugger", Advances in Science, Technology and Engineering Systems Journal Vol. 5, No. 6, pp. 1178-1189, 2020, doi: 10.25046/aj0506142

[11] C. Collberg, C. Thomborson and D. Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In Proc. ACM POPL'98, pp. 184-196, 1998.

[12] C. Collberg, C. Thomborson and D. Low. A Taxonomy of Obfuscating Transformations. Technical Report 148, The University of Auckland, New Zealand, 1997.

[13] M. Dalla Preda, M. Madou, K. De Bosschere, and R. Giacobazzi, “Opaque Predicates Detection by Abstract Interpretation”, Algebraic Methodology and Software Technology, LNCS 4019:81--95, 2006.

[14] P. Szor, Advanced Code Evolution Techniques and Computer Virus Generator Kits in *The Art of Computer Virus Research and Defense,* Phoenix, MaryLand, USA Addison-Wesley Professional, 2005 pp. *244–250*.

[15] J. Park, Y.-H. Jang, S. Hong and Y. Park, "Automatic Detection and Bypassing of Anti-Debugging Techniques for Microsoft Windows Environments", Adv. Electr. Comput. Eng, vol. 19, no. 2, pp. 23-29, 2019.

[16] X. Ugarte-Pedrero, I. Santos, and P. G. Bringas, "Structural Feature based Anomaly Detection for Packed Executable Identication," in Proceedings of the 4th International Conference on Computational Intelligence in Security for Information Systems (CISIS'11), 2011, pp. 50-57.

[17] I. Santos, X. Ugarte-Pedrero, and B. Sanz, "Collective Classification for Packed Executable Identification," in Proceedings of the 8th Annual Collaboration, Electronic messaging, AntiAbuse and Spam Conference (CEAS'11), 2011, pp. 231-238.

[18] R. Perdisci, A. Lanzi, and W. Lee, "Classification of Packed Executables for Accurate Computer Virus Detection," in Pattern Recognition Letters, 2008, pp. 1941-1946.

[19] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware," in Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06), Washington, DC, USA, 2006, pp. 289-300.

[20] R. Lyda and J. Hamrock, "Using Entropy Analysis to Find Encrypted and Packed Malware," in Proceedings of the IEEE Symposium on Security and Privacy (SSP'07), March 2007, pp. 40-45.