

Merhabalar,

NOT: GDB kullanımı ile ilgili komutların her detayına inenemiş olabilirim. Bu yüzden GDB Cheat sheet'e bakarak takip edebilirsiniz, takıldığınız yerleri. [GDB Cheat Sheet](#) bu adrestem bakabilirsiniz.

Reverse2 sorusunun çözümü için GDB aracını kullanarak ilerlemeye başlayalım.

Programı çalıştırıyoruz:

```
root@kali:~/pwn# ./reverse2
Enter the my hash :1234
Try harder !
root@kali:~/pwn#
```

Bir önceki writeup'dan yola çıkarak tekrar Gdb aracı ile programın içine girerek incelemeye başlayalım:

```
root@kali:~/pwn# gdb -q ./reverse2
Reading symbols from ./reverse2... (no debugging symbols found)...done.
gdb-peda$ b main
Breakpoint 1 at 0x80485c0
gdb-peda$ run
```

Main fonksiyonundaki assembly kodlarını görmek için “disas-main” kodunu çalıştırdıktan sonra kodlara dikkatlice bakarak yapılan işi anlamaya başlayalım:

```
0x080485e1 <+47>: lea    eax,[ebp-0x2c]
0x080485e4 <+50>: push   eax
0x080485e5 <+51>: push   0x804871e
0x080485ea <+56>: call   0x8048400 <__isoc99_scanf@plt>
0x080485ef <+61>: add    esp,0x10
0x080485f2 <+64>: mov    DWORD PTR [ebp-0x21],0x6b6e6573
0x080485f9 <+71>: mov    DWORD PTR [ebp-0x1d],0x79616661
0x08048600 <+78>: mov    DWORD PTR [ebp-0x19],0x79697369
0x08048607 <+85>: mov    DWORD PTR [ebp-0x15],0x696d7269
0x0804860e <+92>: mov    DWORD PTR [ebp-0x11],0x6e697373
0x08048615 <+99>: mov    BYTE PTR [ebp-0xd],0x0
0x08048619 <+103>: sub    esp,0xc
0x0804861c <+106>: lea    eax,[ebp-0x21]
0x0804861f <+109>: push   eax
0x08048620 <+110>: call   0x80483e0 <strlen@plt>
0x08048625 <+115>: add    esp,0x10
0x08048628 <+118>: sub    esp,0x8
0x0804862b <+121>: push   eax
0x0804862c <+122>: lea    eax,[ebp-0x21]
0x0804862f <+125>: push   eax
0x08048630 <+126>: call   0x804851b <hash_hesapla>
0x08048635 <+131>: add    esp,0x10
0x08048638 <+134>: mov    DWORD PTR [ebp-0x28],eax
0x0804863b <+137>: mov    eax,DWORD PTR [ebp-0x2c]
0x0804863e <+140>: sub    esp,0x8
0x08048641 <+143>: push   DWORD PTR [ebp-0x28]
0x08048644 <+146>: push   eax
0x08048645 <+147>: call   0x804857f <coqhoj>
```

Scanf ile bizden girdi aldıktan sonra Hard-coded şekilde ascii karakterlerinin hex halinin stack'e yazıldığını görebilirsiniz. İsterseniz hex değerleri dönüştürüp de bulabilirsiniz. Ama artık debug aracını biraz kullanabildiğimize göre bu String'in son halini kendimiz keşfedelim.

Bunun için main+99 adresinde, yani DWORD Pointerler'a taşıma işlemlerinin bittiği adıma geçiyoruz.(

```
End of assembler dump.
gdb-peda$ b *0x08048615
Breakpoint 2 at 0x8048615
gdb-peda$ c
Continuing.
Enter the my hash :12345
```

Bu adımda bizi tam istediğimiz yerde durdurdu ve bizde stackte nasıl bir String girilmiş hemen “examine” stack’e sorgulama işlemine başladık.

```
gdb-peda$ x/s $ebp-0x11 bfElf.py
0xffffd297: "ssin"
gdb-peda$ x/s $ebp-0x15
0xffffd293: "irmissin"
gdb-peda$ x/s $ebp-0x19
0xffffd28f: "isiyirmissin"
gdb-peda$ x/s $ebp-0x1d
0xffffd28b: "afayisiyirmissin"
gdb-peda$ x/s $ebp-0x21
0xffffd287: "senkafayisiyirmissin"
```

Bu adresleri yukardaki resimleri incerseniz neye göre “examine” edildiğini anlamanız rahat olacaktır.

“x/s” x = examine , s=String değer okuma belirteçleridir.

Hemen ardından gelen call fonksiyonu strlen yani eax’ın tuttuğu adresteki verinin uzunluğunu hesaplayacak. Bunuda hemen strlen altına bp atarak kontrol edelim:

```
EAX: 0x14
EBX: 0x0
ECX: 0x7
EDX: 0xb ('\x0b')
ESI: 0xf7fa9000 --> 0x1d9d6c
EDI: 0xf7fa9000 --> 0x1d9d6c
EBP: 0xffffd2a8 --> 0x0
ESP: 0xffffd260 --> 0xffffd287 ("senkafayisiyirmissin")
EIP: 0x8048625 (<main+115>: add esp,0x10)
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
-----code-----
0x804861c <main+106>: lea eax,[ebp-0x21]
0x804861f <main+109>: push eax
0x8048620 <main+110>: call 0x80483e0 <strlen@plt>
=> 0x8048625 <main+115>: add esp,0x10
```

0x14= 20 karakter uzunluğunda ! doğru. Pedanın Güzelliğinden faydalanarak bir sonraki fonks. olan hash_hesapla ‘ya bakalım.

```
0x804862b <main+121>: push eax
0x804862c <main+122>: lea eax,[ebp-0x21]
0x804862f <main+125>: push eax
=> 0x8048630 <main+126>: call 0x804851b <hash_hesapla>
0x8048635 <main+131>: add esp,0x10
0x8048638 <main+134>: mov DWORD PTR [ebp-0x28],eax
0x804863b <main+137>: mov eax,DWORD PTR [ebp-0x2c]
0x804863e <main+140>: sub esp,0x8
Guessed arguments:
arg[0]: 0xffffd287 ("senkafayisiyirmissin")
arg[1]: 0x14
-----stack-----
0000| 0xffffd260 --> 0xffffd287 ("senkafayisiyirmissin")
0004| 0xffffd264 --> 0x14
0008| 0xffffd268 --> 0xf7e00a89 (add ebx,0x1a8577)
```

Burada tahmini olarakda olsa çok güzel yardımı dokunuyor PEDA’nın ☺ stringimizi ve uzunluğunu parametre olarak alıyor.

Şimdi asıl olay olan hash_hesapla’da neler dönüyor? ☺ :

Fonksiyon bizden aldığı Girdi değerini sadece karşılaştırma olarak en son kullandığı için aslında çok kafa karıştıracak bir şey yokta diyebiliriz. Sadece kendi Stringi ve onun uzunluğuyla sabit işlemler döndürüyor. Hash döngüsünün neler yaptığını inceleyebilirsiniz adım adım. Ben ise burda artık daha fazla kafa ağrıtmadan döngü bitiminde karşılaştırma için ürettiği hash değerini yakalamaya çalışayım:

```

Dump of assembler code for function hash_hesapla:
0x0804851b <+0>:    push    ebp
0x0804851c <+1>:    mov     ebp,esp
0x0804851e <+3>:    sub     esp,0x10
0x08048521 <+6>:    mov     DWORD PTR [ebp-0x4],0x0
0x08048528 <+13>:   mov     eax,DWORD PTR [ebp-0x4]
0x0804852b <+16>:   mov     DWORD PTR [ebp-0x8],eax
0x0804852e <+19>:   jmp     0x08048557 <hash_hesapla+60>
0x08048530 <+21>:   mov     edx,DWORD PTR [ebp-0x4]
0x08048533 <+24>:   mov     eax,DWORD PTR [ebp+0x8]
0x08048536 <+27>:   add     eax,edx
0x08048538 <+29>:   movzx   eax,BYTE PTR [eax]
0x0804853b <+32>:   movsx   eax,al
0x0804853e <+35>:   add     DWORD PTR [ebp-0x8],eax
0x08048541 <+38>:   mov     eax,DWORD PTR [ebp-0x8]
0x08048544 <+41>:   shl     eax,0xa
0x08048547 <+44>:   add     DWORD PTR [ebp-0x8],eax
0x0804854a <+47>:   mov     eax,DWORD PTR [ebp-0x8]
0x0804854d <+50>:   sar     eax,0x6
0x08048550 <+53>:   xor     DWORD PTR [ebp-0x8],eax
0x08048553 <+56>:   add     DWORD PTR [ebp-0x4],0x1
0x08048557 <+60>:   mov     eax,DWORD PTR [ebp-0x4]
0x0804855a <+63>:   cmp     eax,DWORD PTR [ebp+0xc]
0x0804855d <+66>:   jnl     0x08048530 <hash_hesapla+21>
0x0804855f <+68>:   mov     eax,DWORD PTR [ebp-0x8]
0x08048562 <+71>:   shl     eax,0x3
0x08048565 <+74>:   add     DWORD PTR [ebp-0x8],eax
0x08048568 <+77>:   mov     eax,DWORD PTR [ebp-0x8]
0x0804856b <+80>:   sar     eax,0xb
0x0804856e <+83>:   xor     DWORD PTR [ebp-0x8],eax
0x08048571 <+86>:   mov     eax,DWORD PTR [ebp-0x8]
0x08048574 <+89>:   shl     eax,0xf
0x08048577 <+92>:   add     DWORD PTR [ebp-0x8],eax
0x0804857a <+95>:   mov     eax,DWORD PTR [ebp-0x8]
0x0804857d <+98>:   leave
0x0804857e <+99>:   ret
End of assembler dump.

```

Kırmızı ile çizdiğim alanda hash üretme işlemleri “senkafayisiyirmissin” string için çeşitli işlemlerden geçerek üretilmekte. İsterseniz ne işlemler döndüğünü gün yüzüne çıkartabilirsiniz ama şimdilik ben bu adımı geçiyorum.

Son olarak +95 adımına bp atacağım ve eax değerine atılan son değere bakalım , belki bizim merak ettiğimiz hash bu olabilir ☺

```

gdb-peda$ b *hash_hesapla+99
Breakpoint 3 at 0x0804857e
gdb-peda$ c

```

“c” ile devam ederek “p/x \$eax” ile return edilen değere bakalım.

```

Breakpoint 3, 0x0804857e in hash_hesapla ()
gdb-peda$ p/x $eax
$23 = 0x22c92d96
gdb-peda$ p/xd $eax
$24 = 583609750
gdb-peda$ 

```

Son olarak decimal olarak çıktı aldık ve bunu unutmadan artık bitirelim bu işi ☺

Son Fonksiyon olan “coqhoj” a doğru giderken gönderilen parametrelere bakalım ve şimşekler çıksın ^^

```

=> 0x08048645 <main+147>:    call    0x0804857f <coqhoj>
0x0804864a <main+152>:    add     esp,0x10
0x0804864d <main+155>:    mov     eax,0x0
0x08048652 <main+160>:    mov     edx,DWORD PTR [ebp-0xc]
0x08048655 <main+163>:    xor     edx,DWORD PTR gs:0x14
Guessed arguments:
arg[0]: 0x4d2
arg[1]: 0x22c92d96

```

Bundan sonrasını size bırakıyorum , iyi eğlenceler ☺