**HAZIRLAYANLAR**

CANER BERKTAŞ

AHMETCAN AKÇAY

**REGİSTERS (Kaydediciler)**

**DATA REGISTERS**

Genel amaçlı 4 regiser;

**AX (Akümülatör Registeri):** Dört işlem operasyonlarında kullanılır.

**BX (Base Registeri):** Bellek lokasyonlarında baz adres göstericisi olarak kullanılır, yani birtür index registeri olarak kullanılabilir. Bu register data segment içerisinde bir alanıgöstermek için kullanılır.

**CX (Counter Registeri):** Döngü işlemlerinde sayaç olarak kullanılır. Döngünün kaç defadaha döneceğinin sayısını tutar.

**DX (Data Registeri):** Donanım ile yapılan giriş çıkış işlemlerinde kullanılır.

**SI :** Kaynak(source) indeks registeri

**DI:** Gidilecek(destination) indeks registeri

**BP:** Temel(base) işaretçisi

**SP:** Yığın(stack) işaretçisi

**INDEX REGISTERS**

**ESI (Extended Source Index):** Karakter ya da döngü işlemleri sırasında okuma işlemiyapılacak olan yerin adresini gösterir.

**EDI (Extended Destination Indicator):** Karakter ya da döngü işlemleri sırasında yazmaişlemi yapılacak olan yerin adresini gösterir.

**SEGMENT REGISTERS**

**CS(Code Segment):** Program kodlarının makine dilindeki halleri kod segmentde saklanır.Çalıştırılacak tüm komutlar buradadır. CS register kod segmentin başlangıç adresini saklar.

**DS(Data Segment):** İlk değer atılmış (initilazed) global ve statik değişkenler data segmenttesaklanır. DS kaydedicisi data segmentin başlangıç adresini tutar.

**SS(Stack Segment):** Dönüş adresleri yerel fonksiyon değişkenleri ve eski ebp değerleristack içerisinde saklanırlar. SS register stack başlangıç adresini saklar.

**ES:** Ekstra bölüt registeri, kullanım amacını programcı belirler.

**POINTER REGISTERS**

**EIP:** Bir sonraki çalışacak olan komutun adresini saklar

**EBP:**  Stackte referans noktası oluşturur.

**ESP:** Stack bölgesinin en üst noktasını gösterir.

**ASSEMBLY NEDİR?**

Assembly alt düzey bir programlama dilidir. İşlemciye aracısız erişim sağlar.

0C85:0100 B402 MOV AH,02

0C85:0102 B203 MOV DL,03

0C85:0104 CD21 INT 21

0C85:0106 CD20 INT 20

Ayrıntılı çıktısı alınmış assembly kodunun;

\*Gri kodlar, assembly kodunun hangi bellek adresinde bulunduğunu gösteriyor.

\*Kırmızı renkte olan assembly kodunun kendisidir.

\*Yeşil renkte olan ise assembly kodunun makine diline çevrilmiş halidir.

**X86 Assembly/Temel komutlar**

**mov**

Bir yazmaçtan bir yazmaca kopyalama yapılabileceği gibi bir yazmaca sabit bir değer deyazılabilir veya yazmaca sabit bir değer yazılabilir.

mov ah,5 > ah yazmacına 5 yazılır.

mov al, bh > ile verilen yazmacın değeri ikinci verilen yazmaca yazılır.

Bu komut ile yalnızca aynı kapasitedeki yazmaçlar arasında kopyalama yapılabilir. Sonu **x** ile bitenler arasında veya sonu **h** veya **l** ile bitenler arasında kopyalama yapılabilir.

**İnc**

Yazmaçta olan değeri 1 arttırır.

inc al > komutu yazınca al değerini 1 arttırıyor.

**mul**

Belirtilen yazmaçtaki değerle al yazmacındaki değeri çarpıp sonucu ax yazmacına koyar.

mul cl > al yazmacında ki değerle cl yazmacında ki değer çarpılıp sonuç ax yazmacına

konur.

**add**

Herhangi bir yazmaçtaki değerle herhangi sabit bir değer veya yazmaçtaki değeri toplayıp sonuç ilk belirtilen yazmaca koyulur. Add komutunda argüman olarak yalnızca eşit kapasitedeki yazmaçları alabilir. Hatalı örnek > add ax, cl

**sub**

Herhangi bir yazmaçtan sabit bir değeri ya da başka bir yazmaçtaki değeri çıkartır. Sub komutu da eşit kapasiteli yazmaçlarla işlem yapar.

sub al, 10 > al yazmacında ki değer 10 eksilir.

sbu ax, cx > ax yazmacında ki değer cx yazmacında ki değer kadar eksilir.

**loop**

Programı belirlediğimiz adrese belirlediğimiz sayıda sıçratır.

Komutun argümanında ki ofset adresi ile loop komutunun bulunduğu satır arasında cxdeğeri kadar gidip gelinir. Böylelikle bir döngü kurulmuş olur.

Örnek komut;

**14F7:100 mov cx,5**

**14F7:103 inc al**

**14F7:105 add dl,al**

**14F7:107 loop 103**

**14F7:109 int 20**

Burada 5 kere 103.ofset ile 107. ofset arasında gidip gelinir. Bütün yazmaçların ilk değerlerini 0 olarak düsünürken sonuçta al ‘in değeri 5, dl’nin değeri de f olur.

**cmp**

İki yazmaçtaki değeri ya da bir yazmaçtaki değeri kıyaslar. Tek başına bir işe yaramaz.

**14F7:100 cmp al,bl >** çalışır

**14F7:102 cmp bl,5 >** çalışır

**14F7:104 cmp 5,bl >** çalışmaz, çünkü; ilk argüman sabit bir değer.

**14F7:106 cmp ax,bl >** çalışmaz, farklı boyutlardaki yazmaçlar kıyaslanmış

**jg, jl, je, jge, jle**

Kendisinden hemen önce gelen cmp komutuna göre programın akışını belirli bir ofsete yönlendirirler.

14F7:100 cmp al,bl

14F7:102 jl 10e

Yukarıda ki komutta al değeri bl den küçükse akış 10e ofsetine yönlendirilecektir. Değilse program normal akışına devam edecek.

**jg** → büyüktür.

**jl** → küçüktür.

**je** → eşittir.

**jge** → büyük eşittir.

**jle** → küçük eşittir.

**jmp**

Belirli bir ofsete atlama yapmamıza yarar.

14F7:144 jmp 110 > burada program akışı 110. ofsete yönlendirilir.

**PUSH**

Yığına(Stack) değer göndermek için kullanılır.Atılan(Kaydedilen) veriyi geri getirmek için POP kullanılır.

MOV eax,1234 #eax e 1234 değerini atadık

PUSH eax #eax deki değeri yığına koyduk(yerleştirdik nasıl denirse)

**CALL**

Alt fromgram çağırmak için kullanılır(Değer saklamak için de kullanılabiliyor). Call direk stack ile ilişkilidir call kullanıldığında kendinden sonraki adres stack alanına yazılır bu sayede program çağırılan alanda işi bittikten sonra nereye döneceğini bilebilir.

CALL CS:[BX+DI] #Çağırılan alan gider ve bir alt satırın adresine geri döner (Tabi o kısmı programın gidişatına bağlaı birazda)

**POP**

Stack den veri çekmeye yarar(daha evvelden push ile konulmuş değeri geri çağırımak için kullanılır).Stack e en son yazılmış veriyi çağırırı.

MOV eax,1234 #eax in içine 1234 yazdık

PUSH eax #eax değerini stack e attık

XOR eax,eax #eax in içini boşalttık

POP eax #1234 değerini Stack den çağırdık ve eax in içine attık

**LEA**

MOV komutuyla benzerdir tek farkı değeri değil değerin yolunu yazmaça yazmış oluruz. Böylelikle istediğimiz yerden değeri çağırabiliriz.

LEA [register],[adres]

**XOR**

Şu şekilde çalışır:

1 ^ 1 = 0

1 ^ 0 = 1

0 ^ 1 = 1

0 ^ 0 = 0

**Buffer Overflow(BOF)**

Ara bellek taşması. Programlarda herhangi bir türde veri alan yerler aldıkları bu verileri bir değişkene kaydederler. Kaydedilen bu değişken için hafızada belirli bir alan ayrılır. Gelen veri bu alana sığmıycak kadar büyükse gönderilen değerin fazlalık kısmı diğer değişkenlere geçer(Sıradaki değişkene).

Ör:

Zafiyetli C kodu:

#include <stdio.h>

int main(){

char name[20]="";

char lastname[20]="";

printf("Name: ");

scanf("%s",name);

printf("Name = %s\n",name);

printf("Last Name = %s\n",lastname);

}

**Açıklama:**

Mantıken bu program bizden aldığı değeri name değerinin içine kaydeder.name değerinin max uzunluğu 20 karakter olmak zorunda 20 karakterden fazla bir değer gelirse gelen değer lastname değerine taşar ve eğer lastname yi de aşar ise program hata vericek(Fuzze etmiş olcaz).

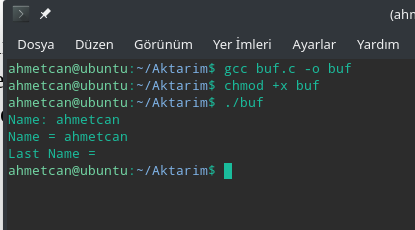
Uygulama:

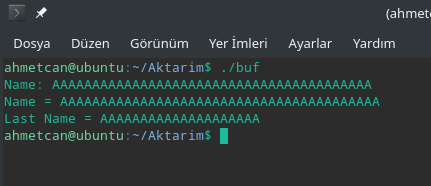
gcc bof.c -o bof

chmod +x bof

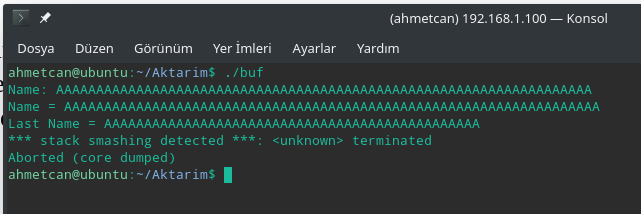
./bof

**Normalde:**



**lastname veri taşırma:**

**Programı Patlatma İşlemi(Fuzzing):**



Buradan bu programda BOF açığı olduğunu çıkarabiliriz.

Burada BOF’un 1.adımı olan fuzzing işlemini gerçekleştirdik.