Fuzzing

Tarihçe

Günümüzde kullanılan **Fuzz** / **Fuzzing** kavramının ilk olarak 80'li yılların sonlarında kullanıldığı söylenmektedir. IT sektöründe kabul görülen isim Barton Miller'dır.

Barton Miller, **Fuzz** terimini şiddetli bir fırtınanın başlamasıyla keşfetmiş. Fırtınalı bir günde dial-up modem ile shell üzerinden Unix bir Sisteme bağlıyken Miller, yağan yağmurun telefon hattına olan etkisinden ötürü çalıştırılan Shell Komutlarının bozduğunu farketmiş. Bundan yola çıkarak rastgele data, unstructured data (yapılandırılmamış data) kavramıyla oluşturacak bir isim vermek istemiş ve **fuzz** terimi hayat bulmuş.

Ayrıca, Miller'ın fuzz terimi için yapmış olduğu bir açıklamayı orjinal haliyle koymak istiyorum.

The original work was inspired by being logged on to a modem during a storm with lots of line noise. And the line noise was generating junk characters that seemingly was causing programs to crash. The noise suggested the term "fuzz".

--bart miller

Fuzzing

Fuzzing, güvenlik araştırmacılarının uygulamalara yönelik; rastgele veya odaklı olarak oluşturdukları bir test tekniğidir. Bu kapsamda yapılmış olunan teste **Fuzzing** veya **Fuzz Testing** denir.

Test ortamının amacı, sistem veya uygulamaların beklenin dışında; otomatik, yarı-otomatik veya manuel bir şekilde **fuzz** edilerek; Uygulama veya sistemin tepkilerini görme ve olası ihtimallerde manipüle etmek suretiyle, istisnai /çökme durumlarını görüntülemektir.

Fuzzing günümüzde kabul görülen IEEE[Institute of Electrical and Electronics Engineers: 24765:2010] tarafından standart haline getirmiş olan **Robustness Testing** içerisinde uygulamaların güvenilirlik / sağlamlık testleri için kriter olarak kabul görmektedir.Ayrıca **Fuzzing** başlığı ile ilgilenen kişilerin **Software Security Testing** başlığı altında bulunan **White Box Testing**, **Black Box Testing** ve **Gray Box Testing** kavramlarını incelemelerini tavsiye ederim.

Fuzzers

Sistem veya Uygulama **Fuzz** edilirken kullanılan herhangi bir uygulama veya script için **Fuzzer** tanımı kullanılmaktadır.

Fuzzing çoğu zaman beklenenden daha farklı (uzun veya kısa, eksi yada arti) değer, belki bozuk bir dosya formatı veya hiç ulaşamayacağı bir sinyal adresi gönderilerek yapılır, bu işlemi yapan uygulama/script/kişi için **fuzz tester** veya **fuzzer** denir. **Fuzzer**'lardan genel olarak beklenen nitelikler şu şekilde sıralanabilinir.

- **Data Oluşturma**
- **Data Aktarma**
- **Monitoring/Loglama**
- **Otomasyon**

Fuzzer'lar yapılarına görede farklı kategorilerde sınıflandırılmaktadır.

- ### Manuel
  İsmindende anlaşılacağ üzere **Manuel** olarak yapılan test biçimidir. **Maneul fuzzing** / **Manuel Fuzz testing** hedef sistemin girdi noktaları manuel olarak kontrol edilerek uygulanır. Datalar **Manuel** olarak değiştirilir ve gönderilir, sistem veya uygulamanın tepkileri **manuel** olarak incelenir.

- ### Yarı Otomatik Fuzzer
  Aynı şekilde isminden anlaşılacağı üzere yarı otomatik **fuzz testing**, **manuel**'e göre bir aşama daha basit ve başarılıdır. bir uygulama veya scriptten yararlanılarak hedef sistem veya uygulamaya otomatik bir şekilde data oluşturur ve gönderir ve tepkileri manuel olarak incelenir.

- ### Full Otomatik Fuzzer
  En kullanışlı olan **fuzzer**'lar otomatik olanlardır. Hedef sistem veya uygulama üzerinde çalıştırılır ve crash olana dek beklenir. girdi noktalarını ve sonuçlarını kendi kendine otomatik bir şekilde oluşturur, gönderir ve loglar...

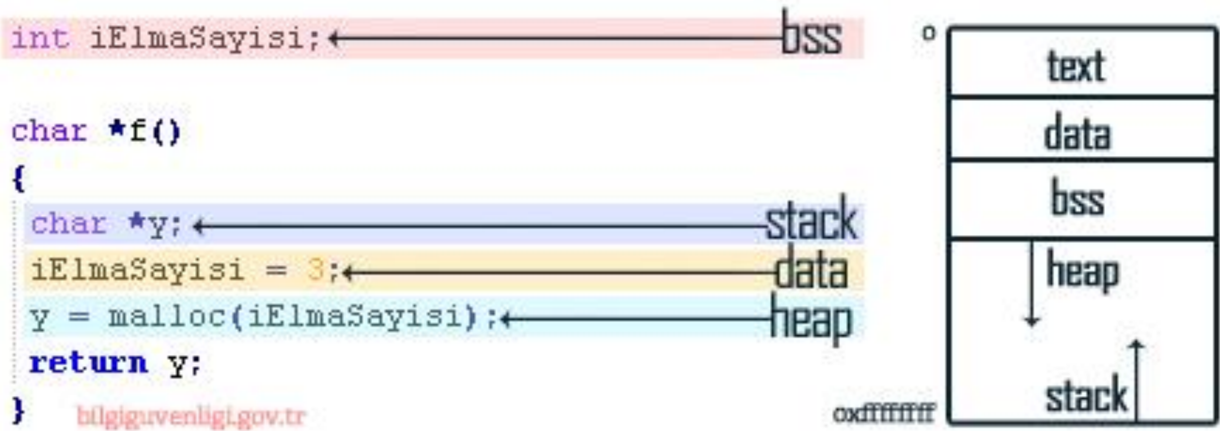
Şimdiye kadar bulunmuş bilinen, bilinmmeyen(0-days) birçok güvenlik zafiyeti **fuzzer** kullanılarak keşfedilmiştir. Çünkü insan gücünün ötesinde daha fazla varyasyonda veriyi, çok daha kısa sürede üretebilmekte ve sonuçlarını monitörize edebilmeyi kolaylaştırmaktadırlar.

Stack & Heap

Stack, hafıza alanında bulunan bir segment adıdır. Bu segmentte; İlgili **process** çalıştığı anda kullanılacak verileri depolar, **process** sonlandırıldığında ise boşaltır. Stack çalışan **process**'ler içerisinde bulunan **local değişken**leri, **function**ların alacağı **parametre**leri ve **function**ların sonlandığında **geri dönüş** yapacağı adresleri tutar.

Heap'te aynı şekilde hafıza alanında bulunan bir segmenttir fakat **stack**'tan farklı olarak **process**lerde **dinamik** olarak oluşturulan değişkenlerinin hafıza alandır. Okuduğum bir makalede bir yazar, **heap** için "yazılımcının memory alanındaki tarlası" tabirini kullanmıştı. Bencede oldukça uygun bir tanım olabilir.

Şimdi bir kaç satır kod yazarak konumuzu örnekleyelim.



Yukarıdaki kodları okumaya kalktığımızda, **stack** ve **heap** arasındaki fark umarım anlaşılmıştır. **f** function çağrıldığında(**f** functionın çağrılmasıda stack'ın içerisindeki alanda dönen bir olaydır), **f** function içerisinde, **y**(**char**) değişkeni oluşturulacaktır, bu değişken, işleyiş aşamasında oluşturulacağı için **stack**'a yerleşecektir fakat **malloc** function ile değişken için atanan, (*iElmaSayisi*(**int**) değişkeninin boyutu kadar) bellek, **heap**'ten tahsis edilecektir. **f** function sonlandırıldığında ise **heap'**ten tahsis edilen bu alan yok edilecektir. Son olarak bakacak olursak;

Global ve Statik Değişkenler = BSS Segment'te tutulur
Local Değişkenler = Stack Segment'te tutulur
Dinamik Değişkenler = Heap Segment'te tutulur

Stack ve **Heap** kavramlarının Türkçe olarak anlatıldığı/içinde geçtiği bazı kaynaklara buradan ulaşabilirsiniz:
http://www.bilgiguvenligi.gov.tr/yazilim-guvenligi/bellek-tasmasi-stack-overflow-ve-korunma-yontemleri.html
http://www.bilgiguvenligi.gov.tr/yazilim-guvenligi/linux-sistemlerde-bellek-tasmasi-koruma-mekanizmalari.html
http://www.bilgiguvenligi.gov.tr/siniflandirilmamis/arabellek-tasmasi-zafiyeti-buffer-overflow.html
http://www.bilgiguvenligi.gov.tr/yazilim-guvenligi/arm-exploitinge-giris.html

- **Stack**, **Heap** ve **x86 Register**ları daha ayrıntılı olarak incelemelisiniz. **Fuzzing** kavramının amacının ilgili uygulamaların işleyişlerinde beklenmedik bir durum oluşturmak olduğunu söylemiştik, Bu işleyişe müdahale edebilmek için **Stack**&**Heap** ve **CPU Register**ları hakkında daha fazla bilgi edinmeniz gerekli.
- Ayrıca **Heap-Spraying, ASLR Bypass, DEP Bypass, SafeSeh Bypass, SEHOP Bypass, Stack Cookies Bypass** teknikleri ni araştırmalısınız.
- **Exploitation, Shell Code, PEB Space,TEB Space,TIB Space, x86 General register, x86 Segment Registers, x86 Index and Pointers** keywordleri ile yola cikarak ayrıca araştırmanızı genişletmelisiniz.

Yazılım Zâfiyetleri - Software Vulnerabilities

IT sektöründe Software Vulnerability - Yazılım Zâfiyetleri(güvenlik başlığı altında) kapsamı için farklı tanım ve kategoriler bulunmaktadır. Bu Zâfiyetler, türlerine göre farklı alt-kategorilerde ele alınırlar.

Fuzzing ile ilgilenen kişilerin ilgili Zâfiyetleri, Kapsam ve Sınıflarını iyi öğrenmeleri gereklidir. Oluşturulacak **Fuzzer(lar)'ın** amaçları, gerek tespit edilen **Crash**'lerin incelenmesi açısından veya **Exploit** edilme aşamasında; ilgili Zâfiyetlerin Sınıf ve Kapsamlarına göre ele alınması gerekecektir.

Software Vulnerabilities içinde, Konumuzun **Fuzzing** olmasından dolayı en çok ilgili olacağımız üst başlığımızı **Memory Corruption Vulnerability** - **Bellek Bozulması Zâfiyetleri**'dir. Bellek Bozulması, Genellikle **Prose**sler çalıştığında, Ilgili **Process** için bellek üzerinde ayrılan bellek konumunun izinsin olarak değiştirimesi/hata sonucu değiştirilmesinden ötürü oluşmaktadır.

Windows İşletim Sistemi için uygulamaların yaklaşık olarak 10%'u (ref @ wikipedia.com) **Heap Corruption** - Heap Bozulması/taşması'ndan ötürü gerçekleşmektedir.

Stack-Based veya **Heap-Based Buffer Overflow** dolayı oluşan güvenlik zâfiyetlerinin, güvenlik seviyeleri oldukça yüksektir.. Bellek bozulmasından/taşmasından ötürü oluşan güvenlik zâfiyetlerinden yararlanılarak **code execution** yapılabilmektedir..

Memory Corruption odaklı oluşan bazı güvenlik zâfiyetlerinin sınıfları şu şekildedir.

- Format String

  Kullanıcı taraflı alınan girdilerin(inputs) eksik denetiminden ötürü uygulama içerisinde kullanılacak alanın içerisine yerleşen özel tanımlandar(%x gibi) dolayı oluşurlar. c dilindeki **printf** bunun için sık kullanın örnek bir function'dır.
  Bir ftpserver 'da bulunan Format String Zâfiyeti:
  http://www.exploit-db.com/exploits/20957/
- Stack Buffer Overflows

  Local değişkenler, return adresleri kısacası process'ın akışı Stack segmentte saklanır demiştik. Stack için tahsis edilen bu alanın mevcut kapasitesi aşılırsa/taşarsa, Stack Buffer Overflow hataları oluşmaktadır.
  Microsoft IIS Ftp Server'da bulunan Stack Overflow Zâfiyeti:
  http://www.exploit-db.com/exploits/9541/
- Heap Buffer Overflows

  Değişkenlere Heap segmentte yer tahsis edilir demiştik(malloc) tahsis edilen bu alan kapsite olarak aşılırsa/taşarsa, Heap Buffer Overflow hataları oluşmaktadır.
  Microsoft Internet Explorer'da bulunan Heap Overflow Zâfiyeti:
  http://www.exploit-db.com/exploits/20174/
- Use After Free

  Değişkenlere ayrılan alan işleri bittiğinde yok edilir demiştik(free), bir alan yok edildikten sonra tekrar kullanılırsa Use After Free hataları oluşmaktadır.
  Microsoft Internet Explorer'da bulunan Use After Free Zâfiyeti:
  http://www.exploit-db.com/exploits/28682/
- Double Free

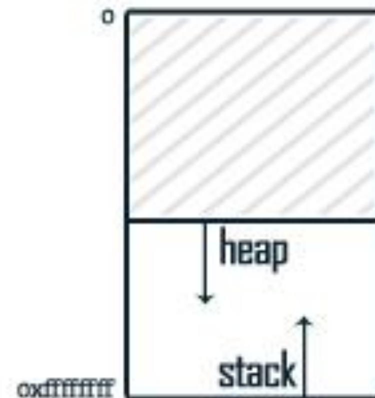
  Değişkenlere ayrılan alan işleri bittiğinde yok edilir demiştik(free), bir alan yoketme tekrarlanırsa Dobule Free hataları oluşmaktadır.
  Microsoft Internet Explorer'da bulunan Double Free Zâfiyeti:
  http://www.exploit-db.com/exploits/3577/

Yazılım zâfiyetleri bunlarla sınırlı değildir, farklı tür ve başlıklar altındada incelenmektedirler. Bu konuda daha ayrıltılı bilgi edinmek için araştırmanızı Software Vulnerabilities genişletebilirsiniz. Şimdi kısaca örnek verdiğimiz zafiyet türleri için bir kaç satır kod yazıp, test ortamında neler olup bittiğini görelim ve yani kısaca manuel olarak kendimiz test edelim;

Format String

```
int main(int argc, char **argv)
{
char buf[100];
sprintf(buf, "Merhaba,%s", argv[1]);
printf(buf);
return 0;
}
```

bilgiguvenligi.gov.tr



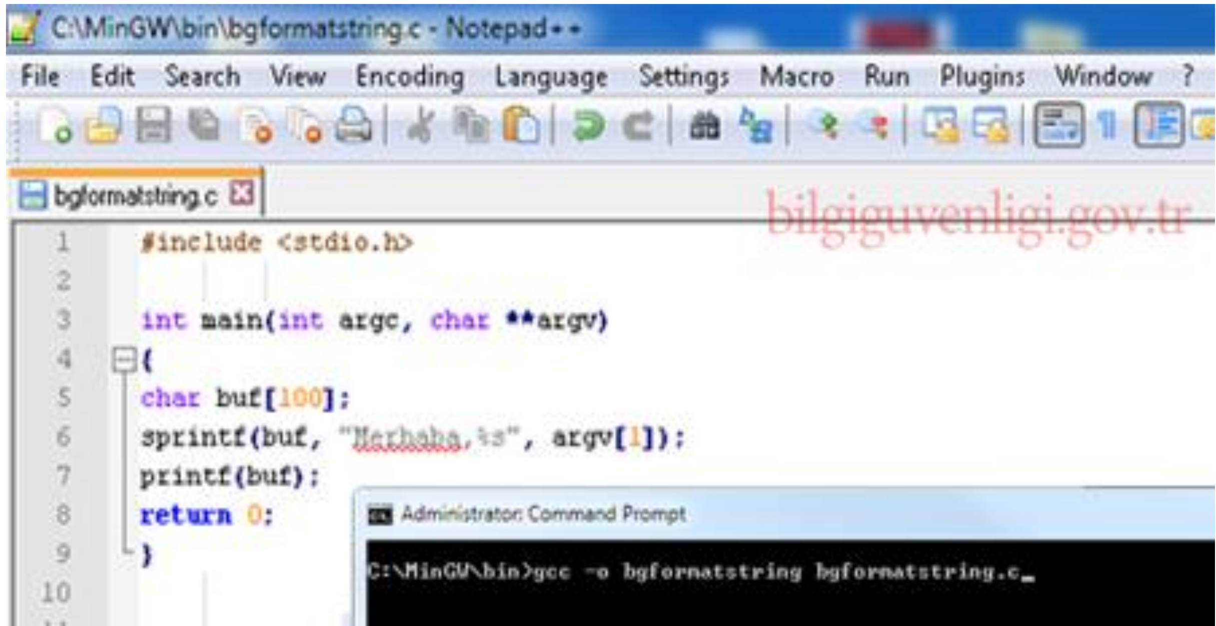
Yukarıdaki kodları okumaya kalktığımızda, **main** function parametre olarak argv almakta. Buradaki **argv** kullanıcımızdan gelen girdiyi temsil etmekte. hemen alt satırda yer alan char **buf[100]** hafızada **buf** değişkeni için **100 byte**'lık bir alanın tahsis edileceğini göstermektedir. bir alt satıra indiğimizde ise **sprintf** function kullanılarak, **buf** değişkenine kullanıcıdan gelen **argv[1]** değerine artı olarak "Merhaba, " atması yapılıyor ve onun alt satırında ise **printf** ile **buf** değişkenin yeni halini ekrana yansıtmakta.

Buraya kadar gelindiğinde, kodların işlem akışı normal olarak gözükmektedir. Fakat, kodlar daha dikkatli incelendiğinde, kullanıcıdan gelen değerin direkt olarak(yani denetimsiz bir şekilde) işlenip tekrar kullanıcıya yansıtıldığı görülecektir. **Format String** türündeki hatalarına eksik denetimden kaynaklı sebepler olduğundan bahsetmiştik. Burada açıkca görülmektedir ki, Kullanıcıdan gelecek olan **format string** işlevine sahip bir bildirgeç (%x, %n vs gibi) ekran çıktısına, yürütülürken hafızaya müdahale ederek işlevsel olarak yansıyacaktır.

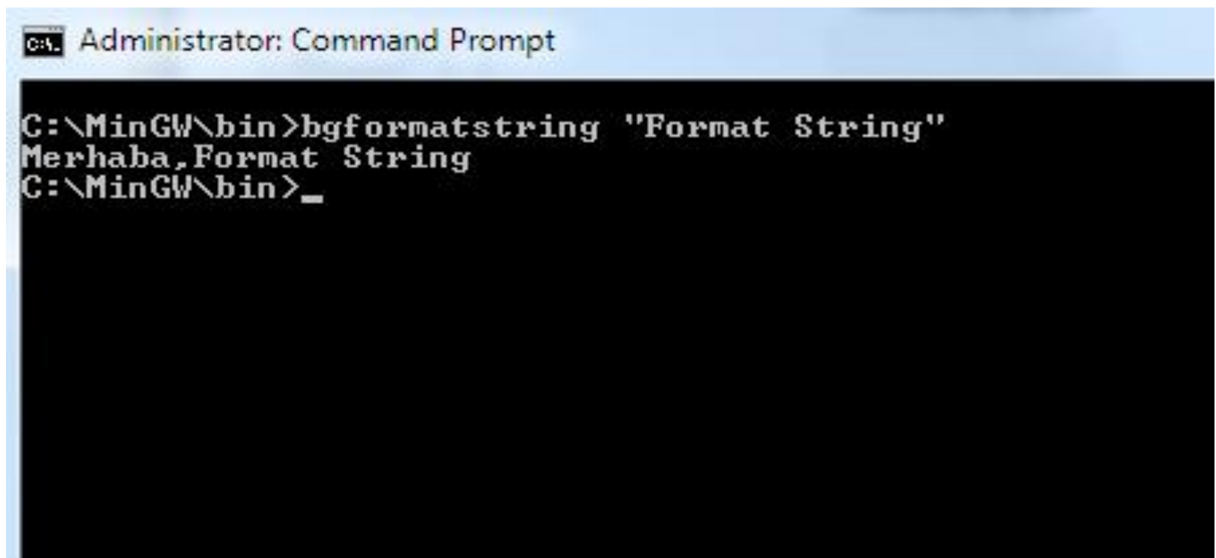
printf function'ı format string parametrelerinde güvensiz function olarak işaret edilmektedir. Burada oluşan zafiyetin sebebi olarak gösterilebilinir.

Şimdi kodlarımızı derleyip bir uygulama haline getirelim, metin editörünü kullanarak yeni bir belge acalim(ben notepad++ kullanıyorum), kodlarımızı yazıp, derleyicimizin bulunduğu klasör içerisine (benim kendi c derleyicim MinGW c:\MinGW\bin dizininde bulunmakta) bgformatstring.c olarak kayit edelim daha sonra command penceresini acip, derleyici dizinine gecip gcc ie kodlarimizi derleyelim.

gcc -o bgformatstring bgformatstring.c



herhangi bir hata mesaji almadiysaniz, uygulamamız derlenmiştir. aynı dizin içerisindeyken command penceresinden **bgformatstring** "**Format String**" yazip enter tuşuna bastığınızda şu şekilde bir çıktı verilecektir.



Görmüş olduğunuz gibi uygulamamız parametre olarak aldığı kullanıcı girdisini (user input) formatlayıp, Merhaba ekiyle ekrana yansıttı. Şimdi dilerseniz "format string" yazdığınız kısma farklı birşeyler yazmayı deneyin. Yazdığınız tüm girdileri ekrana yansıttığını göreceksiniz. Şimdi konumuza

geri dönerek bakarsak, Uygulamamızın ilgili girdiyi ekrana yansıtırken kullanmış olduğu (**printf** functionını hatırlayalım) functionın format string için bildirgeçleri vardı(%s,%i,%n vs) örnek olarak girdilerimizin arasına bu bildirgeçlerden koyarsak acaba neler olacak bakalım.

```
Administrator: Command Prompt

C:\MinGW\bin>bgformatstring %s
Merhaba,Merhaba,%s
C:\MinGW\bin>
```

bgformatstring %s, Görmüş olduğunuz üzere kullanıcı girdisi(user input) olarak %s bildirgeçini aldığında, hafızasındaki değeri bize tekrar ettirdi. isterseniz uygulamanın içerisindeki **sprintf** functionının görevini tekrar bir hatırlayalım, kullanıcıdan **argv**[1] olarak aldığı değeri, "Merhaba," ile %s bildirgeçini kullanarak **buf** değişkenine atayacaktı, bu arada hemen **stack** ve **heap** kavramını gidelim; **process**lerin local değişkenlerinin veya yeni yaratılan değişkenlerini ve bu değişkenlere bağlı olan adresslerin hafıza alanındaki segmentlerde tutulduğunu söylemiştik, şimdi tekrar geri dönersek. kullanıcıdan aldığı değer ile hafızaya müdahale etmiş olduk. Peki ne olacak dersek, hemen örneğimizi çoğaltarak görelim ve %s bildirgeçini bir kaç kez üst üste yazalım.

```
Administrator: Command Prompt

C:\MinGW\bin>bgformatstring %s
Merhaba,Merhaba,%s
C:\MinGW\bin>bgformatstring %s%s%s%s
Merhaba,Merhaba,%s%s%s%s%s(null)Merhaba,%s%s%s%s
C:\MinGW\bin>bgformatstring %s%s%s%s%s
Merhaba,Merhaba,%s%s%s%s%s%s(null)Merhaba,%s%s%s%s%s]|ÉÉÉÉÏ ÜïïÜ14╞ê:ıvâ>
C:\MinGW\bin>
```

farklı bir örnek ile hafıza adresindeki değerleri okuyalım;

bgformatstring AA%x.%x.%x.%x.%x.%x.%x.%x.%x

```
Merhaba,AA403064.701725.0.28febc.7683a442.768d02a8
C:\MinGW\bin>bgformatstring AA%x.%x.%x.%x.%x.%x.%x
Merhaba,AA403064.6b1725.0.28febc.7683a442.768d02a8.6872654d
C:\MinGW\bin>bgformatstring AA%x.%x.%x.%x.%x.%x.%x.%x
Merhaba,AA403064.671725.0.28febc.7683a442.768d02a8.6872654d.2c616261
C:\MinGW\bin>bgformatstring AA%x.%x.%x.%x.%x.%x.%x.%x.%x
Merhaba,AA403064.6f1725.0.28febc.7683a442.768d02a8.6872654d.2c616261.78254141
C:\MinGW\bin>bgformatstring AA%x.%x.%x.%x.%x.%x.%x.%x.%x
Merhaba,AA403064.341725.0.28febc.7683a442.768d02a8.6872654d.2c616261.78254141
C:\MinGW\bin>_
```

görmüş olduğunuz gibi AAdeğerinden sonra gönderdiğim her %x için bana bellekten bir değer döndürdü. bende AA değerini bulana kadar devam ettim ve 7825**4141** olrak işaretlediğim alandaki 4141, benim parametrenin başında gönderdiğim **AA** değerinin hafızadaki hexdecimal karşılığını görebilmekteyiz.

Format String için incelemenizde fayda olacak linklere buradan ulaşabilirsiniz:

http://www.enderunix.org/docs/formatstr.txt - türkçe
http://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf - english
http://www.phrack.org/archives/59/p59_0x07_Advances%20in%20format%20string%20exploitation_by_riq%20&%20gera.txt english
http://www.defcon.org/images/defcon-18/dc-18-presentations/Haas/DEFCON-18-Haas-Adv-Format-String-Attacks.pdf english

Stack Overflow

```c
int main(int argc, char **argv)
{
    char buf[100];
    strcpy(buf, argv[1]);
    return 0;
}
```

bilgiguvenligi.gov.tr

0
heap
stack
0xffffffff

Yukarıdaki kodları okumaya kalktığımızda, **main** function parametre olarak argv almakta. Buradaki **argv** kullanıcımızdan gelen girdiyi temsil etmekte. hemen alt satırda yer alan char **buf[100]** hafızada **buf** değişkeni için **100 byte**'lık bir alanın tahsis edileceğini göstermektedir. bir alt satıra indiğimizde ise **strcpy** function kullanılarak, **buf** değişkenine kullanıcıdan gelen **argv[1]** değeri kopyalanıyor.

Buraya kadar gelindiğinde, kodların işlem akışı normal gibi gözükmektedir. Fakat, kodlar daha dikkatli incelendiğinde, kullanıcıdan gelen değerin direkt olarak(yani denetimsiz bir şekilde) işlenip **buf** değişkenine atandığı anlaşılacaktır.

buf değişkeni için hemen üst satırda 100 byte'lik bir yer tahsis etmiştik, yani bu demek oluyor ki kullanıcıdan 100 byte üzerine cikacak bir değer gelirse, **stack**'ta ayrılan yer taşacak/çatlayacak ve **stack overflow** dediğimiz türden hata ortaya çıkacaktır.

strcpy function'ı güvensiz function olarak işaret edilmektedir. Burada oluşan zafiyetin sebebi olarak gösterilebilinir.

Şimdi kodlarımızı derleyip bir uygulama haline getirelim, metin editörünü kullanarak yeni bir belge acalim(ben notepad++ kullanıyorum), kodlarımızı yazıp, derleyicimizin bulunduğu klasör içerisine (benim kendi c derleyicim MinGW c:\MinGW\bin dizininde bulunmakta) bgbof.c olarak kayit edelim daha sonra command penceresini acip, derleyici dizinine gecip gcc ie kodlarimizi derleyelim.

C:\MinGW\bin\bgbof.c - Notepad++

File Edit Search View Encoding Language Settings Macro Run Plugins Window ?

bgbof.c

```c
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(int argc, char **argv)
5  {
6  char buf[100];
7  strcpy(buf, argv[1]);
8  return 0;
9  }
```

gcc -o bgbof bgbof.c

herhangi bir hata mesaji almadiysaniz, uygulamamız derlenmiştir. aynı dizin içerisindeyken command penceresinden

bgbof AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

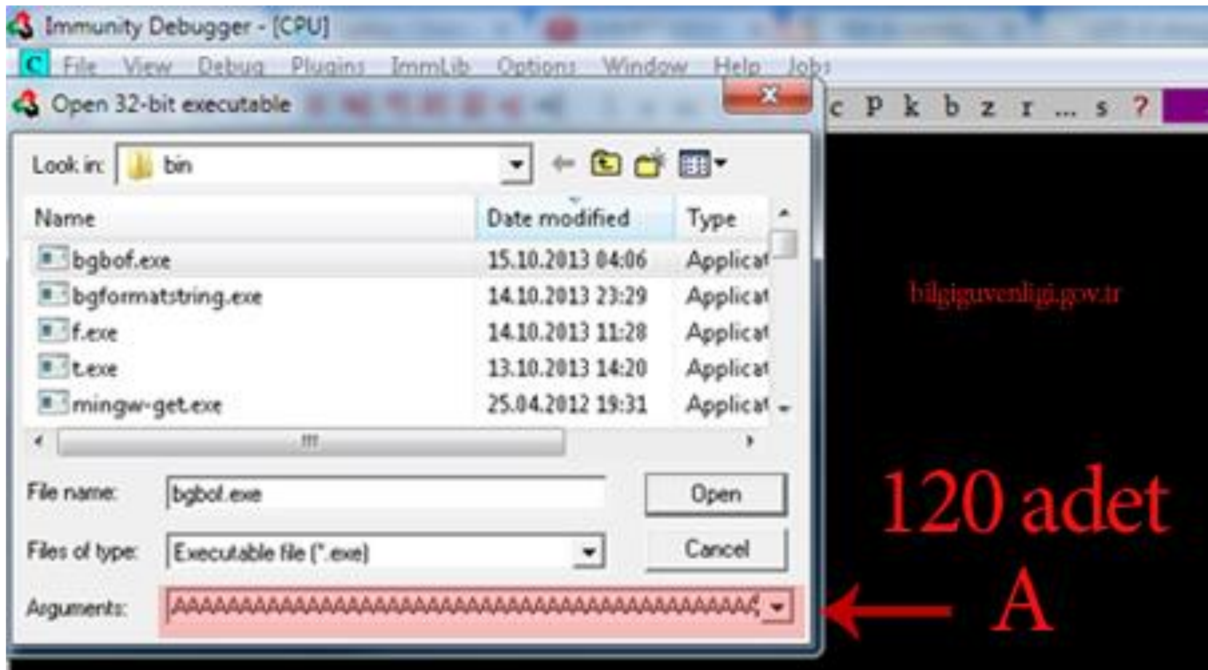
100 adet A yazip enter tuşuna basalım, uygulamamızın başarılı bir şekilde çalıştığını göreceksiniz. Çünkü kullanıcıdan gelen değer, buf değişkeni için tahsin edilen alanın boyutunu aşmadı.. Şimdi uygulamamıza daha büyük boyutta bir veri gönderelim. mesela 120 Adet A yazıp bastığınızda uygulamanın crash olduğunu göreceksiniz. Kullanıcı girdisi(user input) tahsis edilen alanın kapasitenin

üsttüne çıktı ve bundan dolayı uygulama **Stack**'ta bulunan diğer alanların üzerine yazdı. Buuna bağlı olarak uygulamanın akışını değiştirmiş olduk. Peki neden Crash oldu? Çünkü **Stack** kavramında bahsetmiştik, **Stack process'**lerin işleyişteki local değişkenlerini, function parametrelerini ve geri dönüş adreslerini tutuyor, Buradaki registerlardan bahsetmişitik hatta x86 registerların araştırmasının faydalı olacağından bahsetmiştik.

Olayımızın register'lar ile kesiştiği nokta'da tam olarak burasıdır. Kapasitenin aşılmasından ötürü **Stack** üzerinde bulunan diğer alanlara yazılan veri, bir noktadan sonra **EIP**'nin değerinide değiştirdi. **EIP** uygulama akışında çalıştırılacak olan bir sonraki kod adresinin tutulduğu alandır. Bu alanın (**EIP**) değiştirilmesi uygulama akışınıda belirtilen adrese gitmesine sebep oluyor yani CPU'ya **EIP** adresi işaret ediliyor fakat böyle bir adres olmadığı için uygulama crash oluyor.

Çünkü adres alanı biz doldurduk. Şimdi dilerseniz aynı komut satırını windows ortamında bir debugger ile inceleyip görelim. Ben Immunity Debugger kullanacağım dilerseniz bu adresten ulaşabilirsiniz veya farklı bir araçda bakılabilir.

Immunity Debugger linki : http://www.immunityinc.com/products-immdbg.shtml



Uygulamamızı debugger ile birlikte çalıştırdığımızda crash sonrasında aşağıdaki ekran çıktısındada gözüktüğü gibi; **EBP** ve **EIP** üzerine **41414141** yazıldığını fark edeceksiniz. 41 değerinden daha önce bahsetmiştik. büyük A'nın hexdecimal karşılığıdır. yani uygulamaya verdiğim parametre stack'te taşmaya sebebiyet vermiş ve EIP'ye kadar doldurulmuş..

```
Registers (FPU)
EAX 00000000
ECX 009A0F48
EDX BAADF000
EBX 7EFDE000
ESP 0028FF30 ASCII "AAA"
EBP 41414141
ESI 00000000
EDI 00000000

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 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
```

Stack Overflow için yukarıda verilen kaynaklardanda yararlanılabilinir.

Bölüm 1 için anlatacaklarımız şimdilik bu kadar, **Bölüm 2**'de **use after free** ve **double free** zâfiyet türleri hakkında örnekler vereceğiz .

Operasyon 130 başlığı altında bir kaç basit fuzzer geliştireceğiz. Bunların dışında **Bölüm 3**'e kadar yazılarımız devam edecektir. Bölüm 3'te bir kaç farklı **fuzzer** inceledikten sonra **Browser Fuzzing** ve **Exploitation** başlığı altında örnek zafiyeterden yararlanarak bir kaç exploit geliştirmeye çalışacağız.
.
Unutmadan Miller'ın o güne ait anlattıklarnı kendi ağzından duymak için takip bu linki edebilirsiniz: http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html

İbrahim BALİÇ