https://www.fuzzysecurity.com/tutorials/mr\_me/6.html

Переполнение кучи. Часть 5

Ранее мы рассмотрели различные техники обхода механизмов защиты Windows при переполнении кучи. Сегодня же у меня для вас сюрприз. Сегодня мы познакомимся не с очередной техникой атаки переполнением, а узнаем, как использовать уязвимость "двойное освобождение" при переполнении.

Многие скажут вам, что двойное освобождение трудно эксплуатировать, но я вас уверяю: чаще всего его эксплуатация сильно легче, чем обычное переполнение кучи. Мне всегда было любопытно, как можно эксплуатировать эти уязвимости, ведь в некоторых [рекомендациях, отчётах?] они помечены как критические.

На вашей машине нужно установить:

- Windows XP с установленным SP2/SP3

- Immunity Debugger

- pyparser

- graphviz

- heaper.py - плагин к immunity debugger

- 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

- Некоторые знания о внутренних механизмах работы кучи

- Время.

К сожалению, когда я проводил свой анализ, Immunity Debugger не всегда работал корректно, а несколько раз даже падал. Но это нас не остановит ;)

Что такое двойное освобождение?

Определение из не очень популярной OWASP wiki (https://owasp.org/www-community/vulnerabilities/Doubly\_freeing\_memory):

"Ошибка двойного освобождения возникает, когда одно и то же значение указателя передаётся free() более одного раза"

При обсуждении этого определения я пришёл к выводу, что эта уязвимость на самом деле не является переполнением буфера (хотя OWASP говорит именно это). Если у атакующего появляется возможность освободить один и тот же блок памяти дважды, то он может использовать это, изменив висящий указатель на чанк в куче и модифицировать метаданные, чтобы произвести определённую атаку.

Допустим у вас есть освобождённый чанк размером 0x24. И пусть последним элементом lookaside[4] является один из указателей, который будет освобождаться дважды. Получается, что [второе освобождение закончится (поместится?) во FreeList[4], [интересно, как??] ]. Таким образом вы достаточно легко получите два идентичных указателя, хранящихся в разных аллокаторах кучи.

Если атакующий владеет \*некоторым\* (на самом деле минимальными) [детерминизмом, настойчивостью?], это может открыть широкий спектр атак, которые мы рассматривали ранее:

* Перезапись чанка в lookaside при освобождении памяти дважды в lookaside[n] и повторном аллоцировании одного из элементов. Затем перезапись головы списка с висящим указателем и возвращение зловредного flink с целью перезаписи указателя на функцию.
* Освобождение одного чанка в lookaside, а другого во FreeList. Таким образом, при аллокации чанка из lookaside происходит перезапись flink/blink в элементе freelist. С помощью этой техники можно:
  + Заставить flink указавать на зловредный поддельный чанк и вставить его во FreeList[0] после его разделения и возврата после relink. Оставшийся чанк будет вставлен перед поддельным и перезапишет blink поддельного. После этого blink будет указывать на зловредный flink ([Брет Мур - FreeList[0] relink атака]).
  + Заставить flink указывать на поддельный чанк. В этом случае при запросе чанка возвращается поддельный ([Брет Мур - FreeList[0] атака поиском] )
  + Заставить flink указавать на элемент Lookaside или таблицу функций. Blink при этом должен указывать на другую функцию, тем самым контролируя адрес, куда указывает вставленный чанк. Это может дать возможность создавать чанки в Lookaside[n] и доставать их оттуда, пока не будет получен зловредный flink ([Брет Мур - FreeList[0] атака вставкой] )
  + Изменить размер и аллоцировать чанк из элемента списка (после очистки lookaside). Затем в битмапе нужно изменить бит, соответствующий перезаписанному размеру. (Николас Вайзман 2008).
* Напрямую вернуть чанк, [адресом которого является сдвиг от базы кучи[directly return a chunk who's address is an offset from the Heap base itself](странная формулировка)]. С его помощью можно перезаписать управляющие структуры и получить контроль над кучей (Николас Вайзман 2008)
* Освободить два чанка во FreeList[n] и аллоцировать один из них. Перезаписать его размер и указатели flink/blink (предполагается, что этот чанк не единственный в элементе FreeList[n]) на другие элементы. Аллоцировать соответствующий размер, что изменит значение бита FreeListInUse соответствующего размера. Это даст возможность атакующему вернуть чанк, адрес которого является сдвигом от базы кучи (Николас Вайзман 2008)

Пример эксплуатации уязвимости двойного освобождения

Начнём с кода. Скомпилируйте его (замечание: я использую ассемблерные вставки(AT&T) для поиска по vtable и вызовы функций):

/\*

Эксплуатация двойного освобождения с помощью перезаписи чанка в lookaside

POC пример, Стивен Сиили

Благодарю Бретта Мура за код vtable

Замечание: мы перезапишем указатели на функцию приложения, и обратимся по нему, передавая управление в контролируемый атакующим буффер.

\*/

#include <stdio.h>

#include <windows.h>

char \*b1,\*b2,\*b3,\*b4,\*c1,\*c2,\*c3,\*c4,\*d1,\*d2,\*d3;

char \*trigger;

long \*pHeap;

int i;

DWORD function[20];

DWORD ptr1, ptr2;

void callback1(){

printf("(+) Вызван коллбэк 1...\n");

}

void callback2(){

printf("(+) Вызван коллбэк 2...\n");

}

void callback3(){

printf("(+) Вызван коллбэк 3...\n");

}

void callback4(){

printf("(+) Вызван коллбэк 4...\n");

}

void create\_vtable(){

int x;

printf("(+) Создаётся vtable...\n");

(DWORD) function[0] = (DWORD) &callback1;

(DWORD) function[1] = (DWORD) &callback2;

(DWORD) function[3] = (DWORD) &callback3;

(DWORD) function[4] = (DWORD) &callback4;

for(x=5;x<20;x++){

(DWORD) function[x] = (DWORD) &callback1;

}

}

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

{

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

printf("[ -------- двойное освобождение -------- ]\n");

create\_vtable();

printf("\n(+) Создана vtable @ 0x%08x\n", &function);

printf("(+) Производится поиск по vtable @ 0x%08x и вызов по адресу 0x%08x\n", &function, &function[4]);

ptr1 = (int)function;

// поиск по vtable и вызов callback4

\_\_asm\_\_("movl %0, %%ecx":"=m"(ptr1) );

\_\_asm\_\_("mov 0x10(%ecx),%eax\n");

\_\_asm\_\_("call \*%eax");

b1=HeapAlloc(pHeap, 0, 24);

b2=HeapAlloc(pHeap, 0, 24);

b3=HeapAlloc(pHeap, 0, 24);

b4=HeapAlloc(pHeap, 0, 24);

HeapFree(pHeap, 0, b1);

HeapFree(pHeap, 0, b2);

HeapFree(pHeap, 0, b3);

HeapFree(pHeap, 0, b4); // 1ое освобождение

HeapFree(pHeap, 0, b4); // 2ое освобождение (двойное)

c1=HeapAlloc(pHeap, 0, 24);

c2=HeapAlloc(pHeap, 0, 24);

c3=HeapAlloc(pHeap, 0, 24);

c4=HeapAlloc(pHeap, 0, 24);

printf("(+) Чанк b4 (0x%08x) освобождён дважды и висит..\n",&b4);

printf("(+) Перезапись flink висящего чанка указателем на функцию..\n");

gets(c4);

d1=HeapAlloc(pHeap, 0, 24);

d2=HeapAlloc(pHeap, 0, 24);

d3=HeapAlloc(pHeap, 0, 24); // возврат контролируемого чанка из lookaside

printf("(+) Перезапись возвращаемого указателя на функцию...\n");

gets(d3); // перезапись указателя приложения и захват управления

printf("(+) Поиск по vtable @ 0x%08x и вызов по адресу 0x%08x\n", &function, &function[3]);

ptr2 = (int)function;

// поиск по vtable и вызов callback3

\_\_asm\_\_("movl %0, %%ecx":"=m"(ptr1) );

\_\_asm\_\_("mov 0xc(%ecx),%eax\n");

\_\_asm\_\_("call \*%eax");

HeapDestroy( pHeap );

exit(1);

}

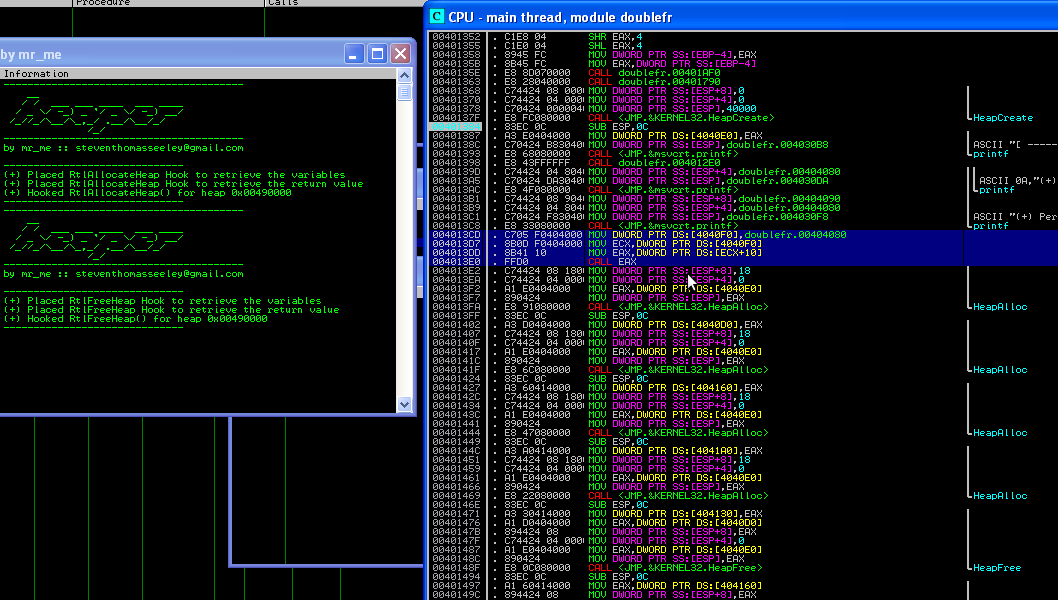
При внимательном рассмотрении вы заметите, что мы используем указатели на функции приложения и что мы перезаписываем функцию в vtable. Откройте скомпилированную программу в отладчике и установить брекпойту сразу после вызова HeapCreate() (0x00401364). Также вызовите следующую команду:

!hidedebug ZwQueryInformationProcess

Теперь запустите приложение. После остановки на брекпойту с помощью heaper повесьте в куче несколько хуков на аллокацию и освобождение.

!heaper hook <heap> -h alloc !heaper hook <heap> -h free

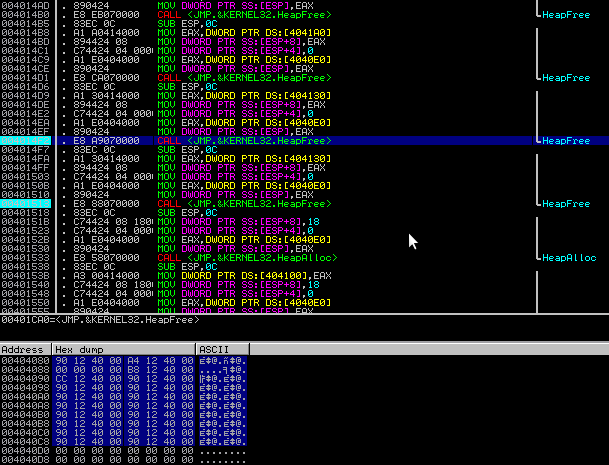
Мы собираемся отследить аллокации и освобождения и попытаться понять, как они работают, в процессе.



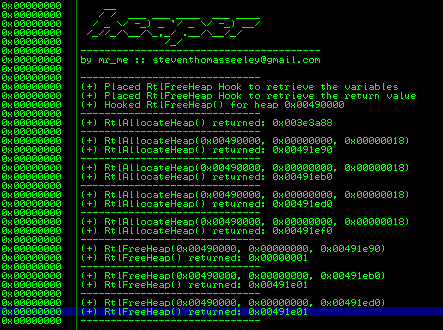
Вы заметите, что часть кода примера подсвечена. Это таблица виртуальных функций. По сути приложения на c++ имеют vtable для всех объектов (она хранится как первый DWORD по указателю на объект) и содержит функции-члены(методы) этого объекта. В приведённом примере нет объекта, только vtable, содержащая указатели, которые мы позже будем использовать для захвата управления процессом.

Заметьте, что vtable расположена по адресу 0x00404080. [Её можно найти в дизассемблере.] Это значение записывается в ECX, происходит вызов функцию по смещению +0x10 от него и управление передаётся этой функции. Это будет важно далее.

Следующее, что нужно сделать - установить брекпойты на последних двух вызовах HeapFree() и вывести дамп по адресу 0x00404080 (vtable). Вы увидите несколько DWORD адресов, которые используют для вызова определённых функций по смещению.



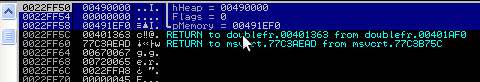
Давайте запустим приложение и проследим за вызовами HeapAlloc() и HeapFree():

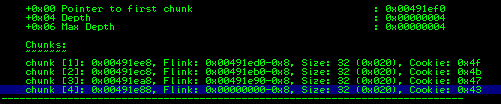


Заметьте, если на этом этапе просмотреть содержимое lookaside[4], мы увидим 3 чанка, а во FreeList[4] не будет ни одного (нам ещё предстоит вызвать HeapFree дважды).

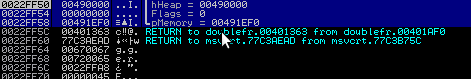


Перепрыгнув HeapFree(), мы обнаружим, что lookaside[4] теперь содержит чанк 0x00491e88, хотя этот адрес является некорректным. В lookaside должен был быть помещён 0x00491ef0. В настоящий момент я не знаю точно, почему это происходит в Immunity Debugger. Так или иначе у нас есть другие пути отслеживания действий приложения, поэтому эта проблема нам не помешает.

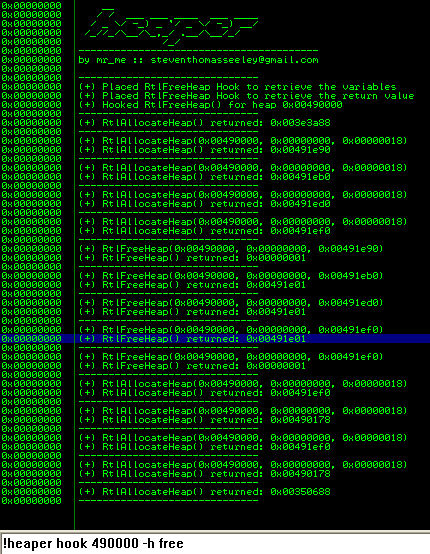




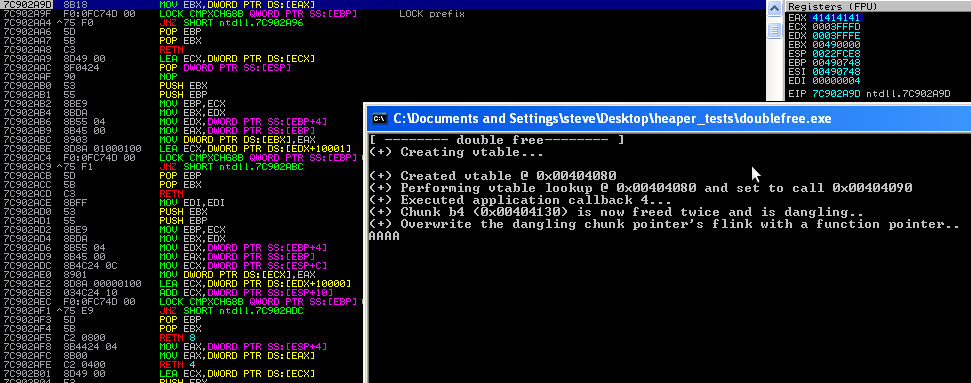
Теперь, остановившись на втором вызове HeapFree() (вызывающим двойное освобождение) мы видим следующие аргументы:



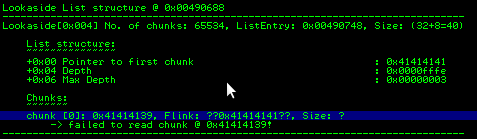
Давайте разберёмся, что происходит, рассмотрев логи хуков:



Всё, что осталось сделать - доставать чанки из lookaside[4] до тех пор, пока мы не получим тот, что вызывает двойное освобождение. Несмотря на четыре последовательных аллокации из lookaside[4], чанк всё ещё находится в списке lookaside[4]. Нам нужно переписать всего 0x4 байта в чанке, чтобы изменить указатель flink, тем самым косвенно создавая поддельный чанк в lookaside.



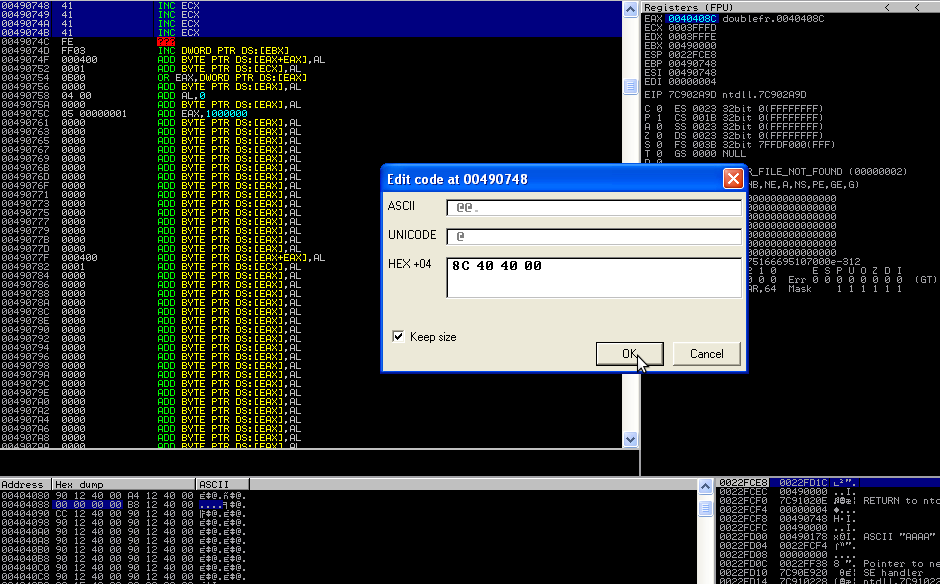
Посмотрим на изменённый lookaside[4]:



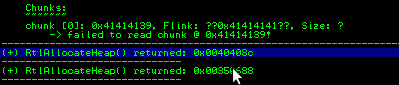
Помните vtable? Если прокрутить ниже в окне дизассемблера, вы заметите вызов [vtable+0xc] после двойного освобождения. Мы знаю, что указатели на функции занимают DWORD (4 байта), значит всё, что нам нужно - перезаписать 3й элемент в vtable.



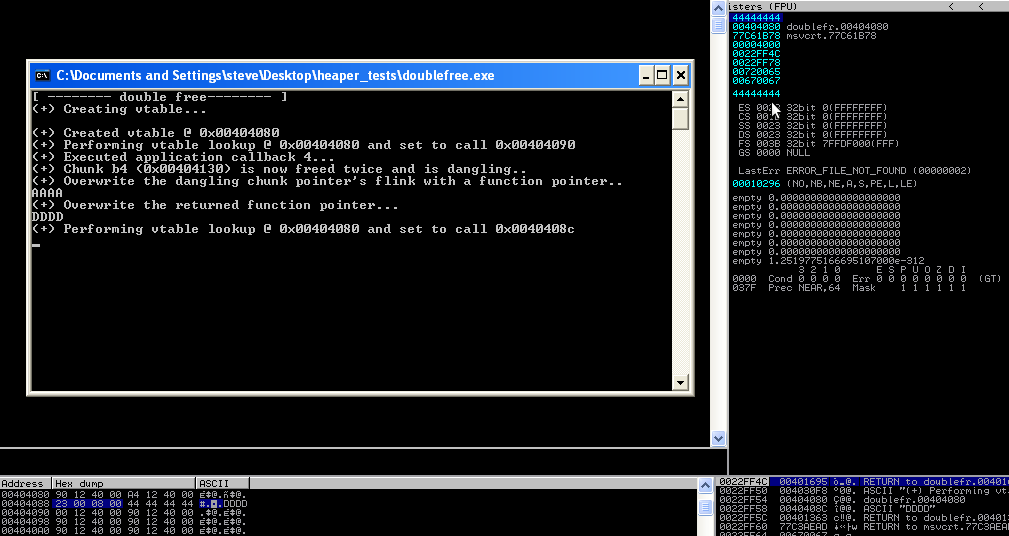
Тут мы просто изменили первые четыре байта в элементе lookaside[4] (теперь выглядит так, что он содержит указатель на первый чанк) и поменять значение EAX на указатель на функцию 0x00404080+0xc.



Если мы сейчас продолжим выполнение, нам вернётся 0040408c как валидный адрес чанка.



Теперь мы можем поместить в него шеллкод. Если мы просто перезапишем первые 4 байта, при вызове функции будет происходить вызов по контролируемому нами указателю на функцию, [передавая управление нашему зловредному буферу]. Стоит заметить, что раз наша vtable хранится в ECX, мы можем попробовать сместить стек на эту область памяти и таким образом [обойти dep ??? dependency??]. Например, так: mov esp, ecx; pop r32; pop r32; pop r32; pop r32; retn.



Заключительные мысли

Как я сказал во введении, вы можете использовать практически любой вектор атаки на кучу для получения контроля с использованием двойного освобождения (в зависимости от уровня вашего [детерминизма]). Поскольку указатель является висячим, атакующий может переписать 0x4-0x8 байтов с указателями flink/blink (больше в случае, если чанк находится в lookaside) через аллокацию висячего чанка. Это открывает дорогу множеству видов атак на приложение, которые способны обойти механизмы защиты Windows XP SP3. Здесь я просто представил самый простой способ добиться исполнения кода. Однако этого без сомнений можно добиться с помощью [двойного освобождения в lookaside и двойной аллокации (However this could no doubt be achieved with a simple double free straight into the lookaside and allocating just twice.)].

К сожалению, я запомнил, как недавно в IRC один человек пытался меня троллить. В итоге ему удалось показать, что я не знаю, как использовать двойное освобождение. Этот пост посвящается ~~тому пидору и~~ всем тем, кто постоянно спрашивает, что такое двойное освобождение и как его можно эксплуатировать.

Ссылки:

- Уязвимости двойного освобождения // Часть 1 [Мэтью Коновер] – тут (https://www.symantec.com/connect/blogs/double-free-vulnerabilities-part-1)

- Уязвимости двойного освобождения // Часть 2 [Мэтью Коновер] – тут (https://www.symantec.com/connect/blogs/double-free-vulnerabilities-part-2)

- Эксплуатация Freelist[0] на XP SP2 [InsomniaSec] – тут (https://www.insomniasec.com/downloads/publications/Exploiting\_Freelist[0]\_On\_XPSP2.zip)