exploits review
14h выпуск

крис касперски ака мыщъх, a.k.a. nezumi, a.k.a elraton, a.k.a. souriz, no-email

новогоднюю ночь мыщъх провел наедине с самым близким ему существом — с монитором. ковырял разные оси. и наковырял! в одном только ядре висты шесть дыр, болтающихся там со времен NT. половина из которых — критические. и это не считая мелких брызг в прочих программных продуктах! под бой курантов мыщъх реализован принципиально новый тип атак на ring-3 стек (условно названный им stack-crossover attack). в общем, рождественские праздники оказались необычайно продуктивными и два последующих обзора exploit'ов решено посвятить описанию багов, собственноручно обнаруженных мыщъхем. демонстрационные exploit'ы прилагаются, а вот заплаток пока еще нет и неизвестно когда они вообще будут…

Windows MessageBeep API – отказ в обслуживании

brief: применительно к незатейливой системной функции MessageBeep, выражение "кричи хоть до посинения" приобретает отнюдь не фигуральный, а вполне конкретный смысл, сопровождаемый голубым экраном смерти. Но все по порядку. Функция MessageBeep (издающая простой набор звуков в стиле SystemAsterisk, SystemExclamation, SystemHand, SystemQuestion и SystemDefault) экспортируется динамической библиотекой USER32.DLL, при дизассемблировании которой мы наталкиваемся на тонкую обертку, ведущую с прикладного уровня вглубь ядра, через реализацется через прерывание INT 2Eh (W2K) или же машинную команду SYSENTER (XP и все последующие системы). Ядро, в свою очередь, перекладывает обработку вызова MessageBeep драйверу WIN32K.SYS, в котором и сосредоточенная львиная доля подсистем USER32 и GDI32. Однако, сам по себе драйвер WIN32K.SYS не может издавать никаких звуков (ну разве что бибикнуть встроенным спикером) и потому поручает это дело драйверу звуковой карты, ставя соответствующий музон в _очередь_ и возвращая управление _до_ того как он будет проигран. Ну и какая проблема?! А вот какая: за короткий отрезок времени прикладной код может поставить в очередь на воспроизведение сотни тысяч звуков, в результате чего мы в лучшем случае получим ~90% загрузку ядра до тех пор, пока вся эта симфония не отыграет, а играть она будет долго, вплоть до морковкинового загнивания, так что семь бед — дави ресет. Причем, _никаким_ путем очистить очередь невозможно и звуковая карта будет пиликать даже после завершения зловредного процесса. Но это еще что! Подумаешь, компьютер тормозит как асфальтовый каток. Достаточно многие драйвера звуковых карт содержат ошибки, приводящие к выпадению в BSOD, со всеми отсюда вытекающими последствиями.

targets: NT, W2K, XP, Server 2003, Server 2008, Висла;

exploit: исходный код exploit'а прост до безобразия и состоит фактически из одной строки: ***for (int a = 0; a < 966666666; a++) MessageBeep(0);*** (естественно, если вызывать MessageBeep из разных потоков, то дело пойдет быстрее и вероятность выпадения в BSOD многократно возрастет, причем, данная атака может быть реализована не только локально, но и через скриптовые языки, поддерживаемые браузерами);

solution: мыщъх не извещал об этой проблеме Microsoft, так что официальная позиция последней по данному вопросу отсутствует. Лично мыщъх просто пропатчил код функции MessageBeep, воткнув перед выходом вызов Sleep(69) выдерживающий паузу в 69 мс и только потом возвращающей управления. На нормальной работе системы это обстоятельство никак не отражается, а вот "забить" очередь зловредному коду уже не получится (поскольку, патч сделан на скорую руку и системно-зависим то в паблик доступ он не выкладывается);

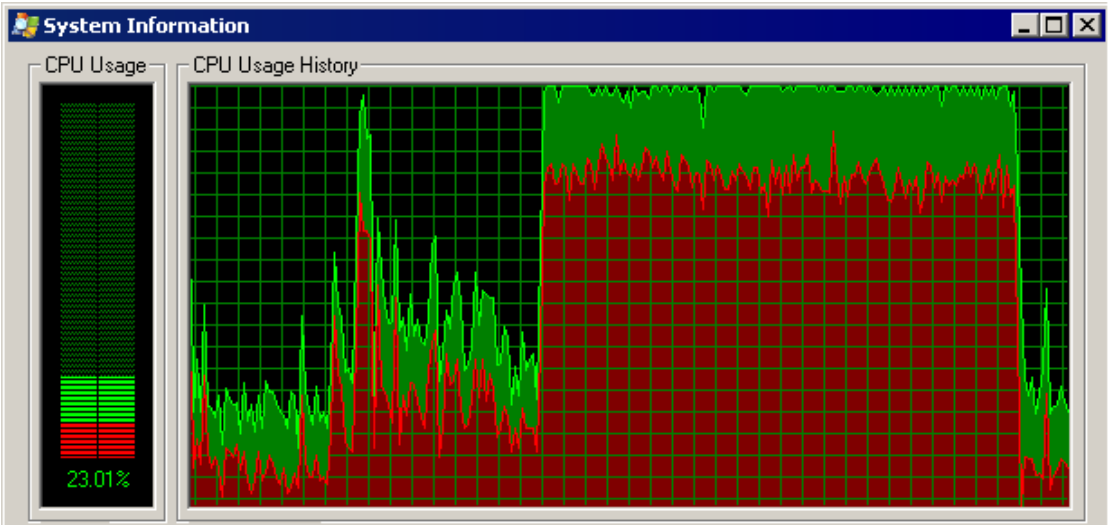

Рисунок 1 загрузка ядра при проигрывании звуковой очереди, созданной многократными вызовами MessageBeep (в данном случае использовалась достаточно коротая очередь, не приводящая к BSOD, поскольку мыщъх не враг своей машине)

SetUnhandledExceptionFilter — design bug

brief: API-функция SetUnhandledExceptionFilter, экспортируемая динамической библиотекой KERNEL32.DLL, позволяет процессу устанавливать фильтр необрабатываемых структурных исключений, заменяющий собой системный фильтр, завершающий приложение в аварийном режиме с посмертной надписью "программа совершила недопустимую операцию…". Архитектурно SetUnhandledExceptionFilter относится ко всему процессу в целом (т. е. его достаточно вызвать всего лишь один раз из любого потока), но конструктивно он вызывается в контексте потока, возбудившего исключение, что требует определенное количество стековой памяти. Если же свободного стекового пространства ни хвоста нет (или регистр ESP указывает на невыделенную или недоступную для записи память), то… вместо ожидаемой генерации EXCEPTION_STACK_OVERFLOW процессор возбуждает исключение EXCEPTION_ACCESS_VIOLATION, что совсем неудивительно, поскольку сегмент стека занимает все адресное пространство и _реальное_ переполнение стека происходит _только_ когда ESP вплотную приближается к нулевому адресу, но поскольку, первые 64 Кбайта адресного пространства в NT зарезервированы для "отлова" нулевых указателей, такая ситуация никогда не случается и операционная система лишь _эмулирует_ EXCEPTION_STACK_OVERFLOW (подробнее об этом рассказывается в разделе "full disclose"). Всякий раз, когда процессор генерирует ошибку доступа к памяти (EXCEPTION_ACCESS_VIOLATION), ядро смотрит: выходит ли стек потока за отведенный ему регион памяти (а по умолчанию, потоку выделяется 1 Мбайт) и если да, то обработчику исключений передается код EXCEPTION_STACK_OVERFLOW, который вместе с прочими параметрами кладется в… стек?! Ну конечно же в стек, а куда же еще! Но ведь у стека у нас уже нет, так?! И как же мы можем туда что-то покласть?! Анализ показывает, что EXCEPTION_STACK_OVERFLOW генерируется, когда в резерве останется чуть менее трех страниц (12 Кбайт) стекового пространства (из которых реально можно использовать только две), что вполне достаточно для большинства целей, но вот если стека _действительно_ нет, то _никакой_ прикладной обработчик _не_ вызывается (включая системный) и ядру ничего не остается, кроме как завершить процесс (именно процесс, а не поток!) без каких бы то ни было сообщений и уведомлений. Как это можно использовать для атаки?! Очень просто — внедряемся в процесс (а внедриться можно даже в более привилегированные процессы, например, через AppInit_DLLs), сбрасываем ESP в нуль и… все. Процесс клеит ласты. Тоже самое происходит при создании в нем удаленного потока API-функцией CreateRemoteThread. Вот тут некоторые могут спросить — а зачем так извращаться?! Если мы можем внедриться в процесс-жертву, достаточно вызывать API-функцию TerminateProcess и все! Ан нет. У процесса легко отобрать право завершать себя, прикладные функции

ExitProcess/TerminateProcess защитному механизму легко перехватить, наконец, "легальная" смерть процесса элементарно "документируется", путем рассылки широковещательных сообщений, "подхватываемые" теневым процессом для перезапуска текущего. Именно так антивирусы и брандмауэры сражаются с малварью. Но вот сброс ESP в нуль при пером же обращении к стеку порождает исключение, после которого не может быть выполнена _ни_ _одна_ API-функция прикладного уровня. Процесс просто необъяснимо исчезает, словно проваливаясь в черную дыру.

targets: NT, W2K, XP, Server 2003, Server 2008, Висла;

exploit: ядро exploit'а "срубающего" любой процесс при внедрении в него выглядит так: asm{xor esp, esp};

solution: решения данной проблемы, по-видимому, не существует, но есть некий воркараунд (от английского workaround — "обходное" решение проблемы или, как мы говорим по-русски — "временное решение", но… следуя же русской философии нельзя ни признать, что ничто так не постоянно, как временное) – все критические процессы запускать из под отладочного процесса и мониторить его состояние. Процесс-отладчик получает исключение EXCEPTION_ACCESS_VIOLATION _до_ завершения отлаживаемого процесса, что позволяет разрулить ситуацию и продолжить нормальное выполнение, впрочем, отладчики типа OllyDebugger на это не способны и в настоящий момент мыщъх пишет свою собственную утилиту для отражения возможных атак.

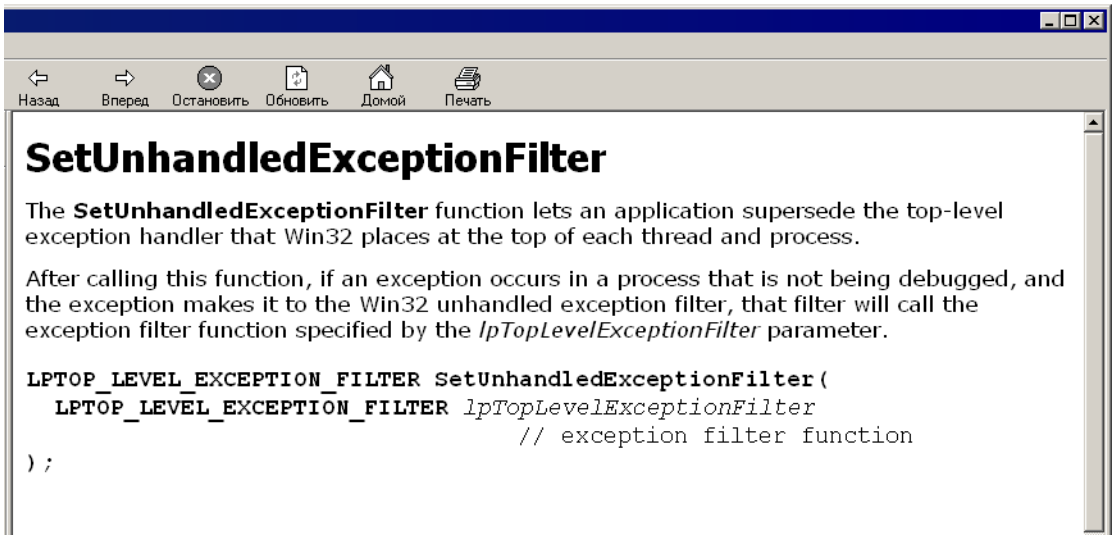

Рисунок 2 описание функции SetUnhandledExceptionFilter на MSDN

OllyDebugger – неверное определение адреса падения

brief: OllyDebugger, ставший де-факто стандартным ring-3 отладчиком, широко используется не только для взлома программ, но и… для их отладки. Ну да, ведь это же отладчик, а не лом ;) А отлаживать приходится в том числе и программы, находящиеся в состоянии клинической смерти (то есть после критического сбоя). Естественно, при этом мы неявно постулируем, что сам отладчик работает правильно и выдает достоверную информацию. К сожалению, OllyDebugger 1.10 содержит ряд ошибок и вот одна из них: когда стековое пространство _реально_ заканчивается (на самом деле, там остается еще одна страница, с атрибутами выставленными по умолчанию в PAGE_NOACCESS), система генерирует EXCEPTION_ACCESS_VIOLATION по адресу 00031000h (в однопоточной программе скомпилированной MS VC 6.0 с настройками по умолчанию и без рандомизации стекового пространства, впервые появившееся в Висле), однако, OllyDebugger, перепутав trap с fault'ом, выносит неверное суждение и сообщает об ошибке доступа по адресу 00030FFCh (при условии, что запись в стек производится командой PUSHD). Последствия — весь анализ летит к черту и хакер ни хвоста не понимает откуда тут взялось 00030FFCh, когда по всему ведь должно быть 00031000h?!

Но человек — это ладно. Наступит пару раз на грабли и образумится. С "реанимационными" скриптами все намного сложнее. Еще несколько лет назад мыщъх опубликовал в "системном администраторе" статью "практические советы по восстановлению системы в боевых условиях", рассказывают о том, как написать собственный обработчик критических ошибок, восстанавливающий работоспособность программы возвращающий ее в более или менее стабильное состояние, как минимум позволяющее сохранить все не сохраненные данные (текст самой статьи можно бесплатно скачать с мыщъх'иного сервера: http://nezumi.org.ru/zq-degluck.zip). В качестве основного движка сначала использовался отладчик (сперва MS WinDbg, затем — Olly) и вот оказалось, что в некоторых ситуациях Olly "спотыкается" и программа падает окончательно, поэтому, пришлось возвращаться к MS WinDbg, который, кстати говоря, за последние несколько рез резко поумнел и превратился в достойный инструмент с хорошо документированным интерфейсом расширений;

target: OllyDebugger 1.10/2.00;

exploit: __asm{rool: push eax/jmp rool};

solution: мыщъх написал автору OllyDebugger'a письмо с описанием ошибки, но ответа так и не получил, что ж, будем ждать;

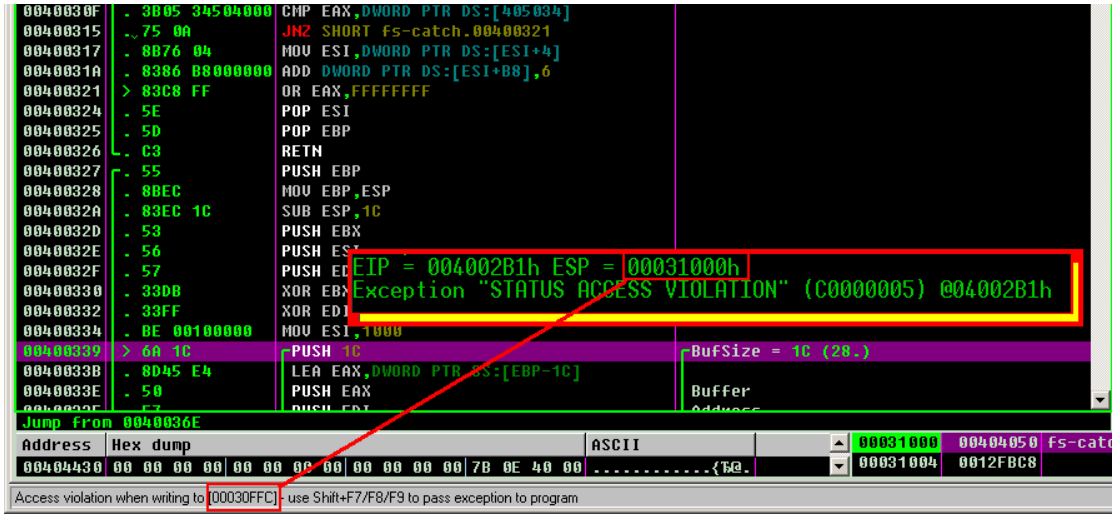

Рисунок 3 OllyDebugger считает, что исключение произошло по адресу 00030FFCh, то время как простейший отладчик, написанный за пять минут на базе MS Debugging API (результат работы которого приведен на врезке) говорит, что подлинный адрес исключения 00031000h

full disclose: новый тип атак на стек — stack-crossover

До сих пор хакеры переполняли стек в одном направлении — вперед и вниз, т. е. в область старших адресов (чтобы не возникло путаницы, условимся, что стек растет вверх). Индексное переполнение (с перезаписью произвольной ячейки памяти) встречалось намного реже. Но никто (или практически никто) еще не переполнял стек назад и вверх!

Беглый анализ, выполненный мыщхем в новогоднюю ночь, выявил большое количество приложений, подверженных угрозе данного типа, от которой никто из программистов даже и не пытался защищаться (действительно, сложно защищаться от того, чего не знаешь).

Начнем с азов, то есть с документированных, но малоизвестных особенностей организации стековой памяти. При создании нового потока система создает и новый стек, резервируя (MEM_RESERVE) необходимое количество страниц памяти (по умолчанию 1 Мбайт, но эта величина может быть изменена параметром dwStackSize API-функции CreateThread, а размер первичного стека, создаваемого при старте процесса, берется из заголовка PE-файла и может меняться линкером).

Рисунок 4 легендарная DEC PDP, обогнавшая время и определившая архитектуру операционных систем на весь последующий век. Microsoft позаимствовала отсюда немало ярких идей, никак и нигде это не обозначив

На дно стека ложится — выражаясь в терминах DEC – так называемая желтая сторожевая страница (yellow guard page) или просто PAGE_GUARD в терминах Microsoft. Это выделенная (MEM_COMMIT) страница памяти с атрибутами (PAGE_READWRITE | PAGE_GUARD). При первом обращении к ней, генерируется исключение STATUS_GUARD_PAGE_VIOLATION, перехватываемое системой, которая снимает атрибут PAGE_GUARD с текущей страницы, выделяет (то есть коммитит от eng – to commit) следующую страницу памяти, и назначает ее сторожевой, путем присвоения атрибута PAGE_GUARD. Таким образом, по мере роста стека, сторожевая страница перемещается наверх, а стеку выделяется все больше и больше памяти. Это достаточно известный факт, описанный в MSDN.

А вот, что MSDN не описано, так это то, что сразу же при создании стека, в непосредственной близости от его вершины размещается (опять-таки выражаясь в терминах DEC) красная сторожевая страница (red guard page), за которой идет выделенная страница памяти с атрибутами PAGE_READWRITE и уже на самой вершине стека располагается страница PAGE_NOACCESS, в результате чего фактический размер стека на 3 страницы (12 Кбайт) меньше обозначенного.

При достижении красной сторожевой страницы генерируется исключение EXCEPTION_STACK_OVERFLOW, которое обрабатывается либо SEH-обработчиком, назначенным программистом, либо управление получает фильтр исключений верхнего уровня, принудительно завершающий работу приложения с выдачей ругательного сообщения известного типа. В распоряжении SEH-обработчика имеется две страницы свободного стекового пространства, что позволяет ему корректно обработать ситуацию.

Проблема в том, что ни MSDN, ни популярные книги по программированию не говорят, что именно нужно делать и 99,9% программистов допускают одну и ту же фатальную ошибку. Предположим, что в нашей программе имеется рекурсивная функция, которая при определенных обстоятельствах может съесть весь стек целиком (на Си++ приложениях такое часто случается).

Когда желтая сторожевая страница докатывается до красной, генерируется исключение EXCEPTION_STACK_OVERFLOW, но при этом красная сторожевая страница становится "зеленой" (термин мой — КК), т. е. лишенной каких бы то ни было защитных атрибутов. Если программист установит фильтр, отлавливающий EXCEPTION_STACK_OVERFLOW и, например, завершающий рекурсивную функцию с тем или иным кодом ошибки, то… все "как

бы" будет работать, но… при повторном возникновении аналогичной ситуации, красная сторожевая страница уже _отсутствует_ и при достижении предыдущего барьера исключение EXCEPTION_STACK_OVERFLOW уже _не_ генерируется. Рекурсивная функция продолжает исполняться и дальше, отъедая одну страницу за другой. А вот когда она со всего маху врезается в последнюю стековую страницу (ту ,что с атрибутами PAGE_NOACCESS), процессор генерирует исключение EXCEPTION_ACCESS_VIOLATION и передает его ядру. Ядро видит, что стек исчерпан и передавать управление SEH-обработчику нет никакой возможности, т. к. он сам нуждается в стеке, а стека-то нет. В результате — происходит тихая смерть процесса на ядерном уровне без передачи управления на ring-3. Естественно, это несколько упрощенная схема, но…

Если мы можем вызывать переполнение стека тем или иным образом (например, рекурсивным запросом), то в первый раз произойдет исключение более или менее корректно обрабатываемое программой, а вот во второй раз — программа склеит ласты. Хороший способ для реализации атаки на отказ в обслуживании!!! (Вообще-то, если программист не пионер, и не только курил мануалы от Microsoft, но еще и нюхал DEC, то в обработчике исключений он вернет красной сторожевой страницы ее статус вызовом функции VirtualProtect с атрибутом PAGE_READWRITE | PAGE_GUARD, но таких программистов среди современников не встречается).

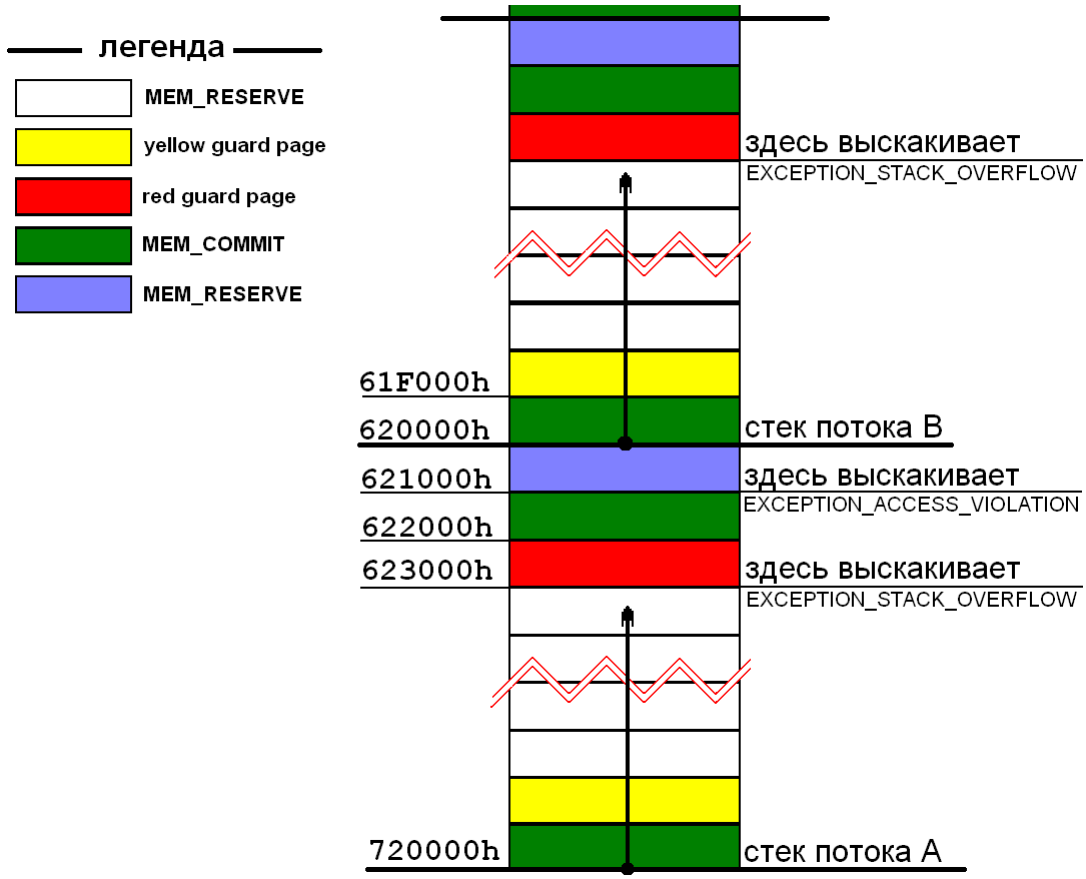

Рисунок 5 устройство стека в операционных системах семейства Windows (примечание: при переполнении потока A и пересечении границ отведенного ему пространства все будет работать до тех пор, пока поток A не "споткнется" о сторожевую страницу потока B — вот тут-тот операционная система и раскусит обман, выплюнув исключение вместо того, чтобы послушно переместить сторожевую страницу потока B на одну позицию вверх)

А теперь задумаемся: зачем Microsoft "застолбила" последнюю стековую страницу, выставив ее в PAGE_NOACCESS?! Ответ — вот для таких пионеров и "застолбила", в противном случае, при повторном переполнении стека (когда красной сторожевой страницы уже нет), программа вылетела бы за пределы стека и пошла "чесать" совершенно посторонние данные, никоим образом ей не принадлежащие. Вот такой, значит, механизм защиты стека от переполнения мы имеем в Windows-системах. Чисто программный и совсем не аппаратный.

Почему не аппаратный? Так ведь тогда на стек каждого потока ядру пришлось бы заводить свой собственный селектор, количество которых в x86 не безгранично и намного меньше, чем потоков в средненагруженной системе. К тому же, адресовать локальные переменные пришлось бы через префикс SS – прощай плоская модель памяти и здравствуйте тормоза!!!

К нашему хакерскому счастью (и большому программистскому несчастью) программную защиту легко одолеть. Для этого достаточно из shell-кода вызывать API-функцию VirtualAlloc, присвоив последней странице стека статус MEM_COMMIT. А если еще сбросить атрибут PAGE_GUARD у красной сторожевой страницы (что можно сделать вызовом VirtualProtect). Аналогичного результата можно добиться просто перезаписав адрес возврата из функции с переполняющимся буфером указателем на API-функции VirtualAlloc/VirtualProtect, передав им атрибуты через стек, что работает на системах с неисполняемым стеком (XP SP2 с аппаратной поддержкой DEP), но, увы, на Висле из-за рандомизации адресного пространства приходится искать более изощренные пути.

В результате: при переполнении стека (например, все той же рекурсивной функцией) исключение _вообще_ не возникает и начинают затираться чужие данные и вот тут-то и начинается самое интересное — кому эти данные принадлежат? Стек главного потока на NT размещается в младших адресах перед страничным образом исполняемого файла и вершина стека смотрит в пустую область, где и затирать-то нечего. Но вот стеки второго и (всех последующих) потоков как правило размещаются следом за страничным образом и потому при переполнении стека мы ударяем в конец исполняемого файла, нанося ему тяжелые телесные повреждения. В зависимости от "настроения" линкера там может находится и секция данных, и таблица импорта (в у динамических библиотек — таблица экспорта) и секция ресурсов, да мало ли еще что. Наибольший интерес, естественно, представляют собой данные, доступные для записи и экспорт/импорт.

Memory map

Address	Size	Owner	Section	Contains	Type	Access	Initial
00010000	00001000				Priv	RW	RW
00020000	00001000				Priv	RW	RW
0012D000	00001000				Priv	RW Guar	RW
0012E000	00002000			stack of main thread	Priv	RW Guar	RW
00130000	00004000				Priv	RW	RW
00230000	00001000				Priv	RWE	RWE
00240000	00003000				Map	RW	RW
00250000	00016000				Map	R	R
00270000	0002F000				Map	R	R
002A0000	00041000				Map	R	R
002F0000	00004000				Map	R	R
00300000	00008000				Priv	RW	RW
00400000	00001000	stack-al		PE header	Imag	R	RWE
00401000	00005000	stack-al	.text	code	Imag	R	RWE
00406000	00001000	stack-al	.rdata	imports	Imag	R	RWE
00407000	00003000	stack-al	.data	data	Imag	R	RWE
00410000	00008000				Priv	RW	RW
00510000	00002000				Map	R	R
00A20000	00001000				Priv	RW	RW
00A21000	000FF000			stack of thread 00000804	Priv	RW	RW
00C1E000	00001000				Priv	RW Guar	RW
00C1F000	00001000			stack of thread 00000920	Priv	RW Guar	RW
00D1E000	00002000				Priv	RW Guar	RW
77F80000	00001000	ntdll		PE header	Imag	R	RWE
77F81000	00044000	ntdll	.text	code,exports	Imag	R	RWE
77FC5000	00005000	ntdll	ECODE	code	Imag	R	RWE
77FCA000	00004000	ntdll	PAGE	code	Imag	R	RWE
77FCE000	00003000	ntdll	.data	data	Imag	R	RWE
77FD1000	00001000	ntdll	EDATA		Imag	R	RWE
77FD2000	00028000	ntdll	.rsrc	resources	Imag	R	RWE
77FFA000	00003000	ntdll	.reloc	relocations	Imag	R	RWE
79430000	00001000	KERNEL32		PE header	Imag	R	RWE
79431000	00059000	KERNEL32	.text	code,imports,exports	Imag	R	RWE
7948A000	00004000	KERNEL32	.data	data	Imag	R	RWE
7948E000	00052000	KERNEL32	.rsrc	resources	Imag	R	RWE
794E0000	00004000	KERNEL32	.reloc	relocations	Imag	R	RWE

Рисунок 6 карта памяти многопоточной программы. видно, что стек главного потока расположен в самом начале адресного пространства и при его переполнении затирать ему совершенно нечего, а вот стеки последующих потоков расположены за концом образа

исполняемого файла и при их переполнении начинает затираться содержимое секции данных, доступной как на чтение, так и на запись!

Однако, если область за концом страничного образа уже занята, то пространство под стек выделяется в другом месте адресного пространства, например, за блоком памяти, принадлежащем куче. Самое интересное, что стратегия выделения памяти под стековое пространство стремится к максимально плотному заполнению адресного пространства и потому перед стеком практически всегда находится что-то "полезное" и только в редких случаях невыделенная область памяти, что происходит, например, при освобождении памяти или завершении потока, владеющего данным регионом.

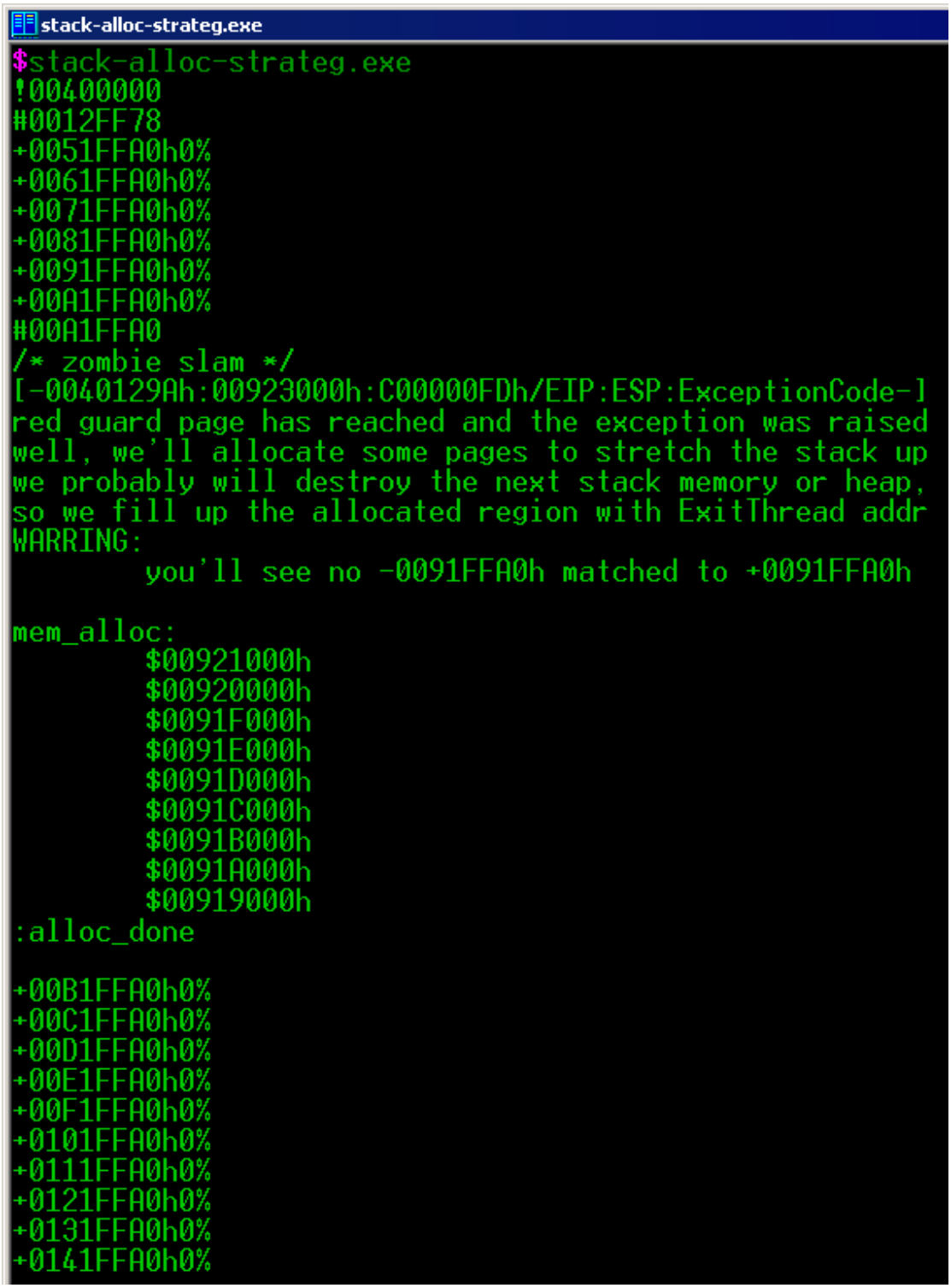

Рисунок 7 результат работы программы "Stack/Heap Allocation strategist", демонстрирующей стратегию выделения памяти для стека/кучи и обход программной защиты от переполнения (саму программу вместе с исходными текстами можно скачать с http://nezumi.org.ru/souriz/hack/ stack-alloc-strateg.zip)

Но, как бы там ни было, к атакам такого типа не готовы ни специалисты по безопасности, ни программисты и пока они опомнятся, у хакеров предостаточно времени для анализа программ, многие из которых допускают "двойное" переполнение стека с "подавлением" исключения EXCEPTION_STACK_OVERFLOW. Список таких программ мыщъх по некоторым соображением не приводит, но вот саму идею с удовольствием выкладывает на всеобщее обозрение.