латание windows своими руками

крис касперски, no-email

дыры в windows обнаруживаются регулярно, но заплатки зачастую выходят с большим опозданием, создавая угрозу атаки, для отражения которой можно (в качестве временного решения) "заштопать" дыру самостоятельно. отсутствие исходных текстов существенно усложняет эту задачу и чтобы облегчить ее решение была написана эта статья, дающая краткий обзор основных методик и приемов, используемых "кодокопателями"

введение

Сегодня, пожалуй, уже никто не сомневается в необходимости ежедневных обновлений операционной системы (нашумевшие вирусные атаки еще свежи в памяти), но при этом далеко не все отдают себе отчет в том, что с момента публикации информации об уязвимости до выхода заплатки проходит какое-то (зачастую, весьма продолжительное) время в течении которого компьютер остается совершенно незащищенным! Следовательно, даже при соблюдении полного комплекса мер безопасности угроза быть атакованным остается вполне реальной, тем более, что хакеры (именно хакеры, а не подростки, качающие безнадежно устаревшие exploit'ы из сети) используют самые новейшие дыры, против которых еще нет вакцины.

Естественно, из потенциальной _возможности_ атаки отнюдь не следует ее _неизбежность_ (в реальной жизни мы ежеминутно подвергаемся тысячам возможных угроз, но лишь ничтожная их часть реализуется на практике). В любом случае, официально обнародованные уязвимости лишь верхушка айсберга. Существует множество "черных" дыр, известных только хакерам, вовсе не собирающихся делиться знаниями с миром и активно использующих эту информацию себе на благо, а их благо — это наш вред.

Короче говоря, если выпуск заплатки задерживается — не стоит паниковать, но… и сидеть сложа руки тоже не вариант, особенно, когда стоимость информации на узле выражается многозначными суммами. Некоторые фирмы, специализирующиеся на безопасности, предлагают нам свои заплатки, опережая неповоротливую Microsoft и, как показывает практика, спрос на такие заплатки огромен.

В частности, стоило только Ильфаку Гильфанову выложить свою заплатку, устраняющую брешь в обработчике wfm-файлов (http://hexblog.com/2005/12/wmf_vuln.html) как его сервер тут же лег под натиском страждущих и это при том, что никакой реальной угрозы эта дыра не представляла, но… с заплаткой, пусть даже неофициальной, все-таки "суше и комфортнее", тем более, что она распространялась бесплатно.

Неофициальные заплатки выпускались и другими компаниями, например фирмой Determina (determina.com/security_center/security_advisories/securityadvisory_march272006_1.asp) или Zert (http://www.isotf.org/zert/#patches), однако, таких заплаток — единицы и их выход — большая удача.

А насколько реально выпустить заплатку самому? В зависимости от специфики уязвимости это может отнять от нескольких часов до… бесконечности. Самое обидное, что с выходом официального обновления весь проделанный труд пойдет насмарку, особенно, если мы закончим работу позже, чем Microsoft. Тем не менее существуют ситуации, в которых самостоятельный выпуск заплатки не только оправдан, но и является _единственным_ возможным решением. Это в первую очередь относится к древним (но все еще используемым) операционным системам и сопутствующему им программному обеспечению, чья поддержка уже прекращена и официальные заплатки не выдут _никогда_

alma mater

Наивно надеяться, что прочитав одну-единственную статью, можно _научиться_ латать систему, зараз выучив несколько языков программирования (включая си, машинный код и ассемблер), освоить отладчик с дизассемблером, наконец, разобраться в архитектуре Windows от прикладного уровня до самого дна, тьфу, то есть ядра. Предполагается, что читатель уже владеет этими знаниями и обладает всеми необходимыми навыками. В противном случае, статья даст ему лишь общее представление о положении вещей.

Автор отдает себе отчет, что далеко не всякий администратор умеет программировать и уж тем более отлаживать программы без исходных текстов, тем не менее, все-таки надеется, что статья найдет свою аудиторию.

классификация и способы устранения дыр

Уязвимости встречаются на всех архитектурных уровнях: в прикладных приложениях, поставляемых вместе с Windows или приобретаемых отдельно (таких, например, как IE и MS Office); в отключаемых службах (скажем, службе печати), не отключаемых системных компонентах (например, процессе WINLOGON.EXE), динамических библиотеках прикладного уровня (KERNEL32.DLL, GDI32.DLL, USER32.DLL и др.), в компонентах ядра (драйвере TCPIO.SYS) и непосредственно в самом ядре.

Природа дыр так же варьируется в широких пределах: это может быть и классическое переполнение, вызванное отсутствием контроля длины копируемых данных, и ошибки синхронизации потоков, и недокументированные (малоизвестные) возможности, позволяющие вместе с данными передать исполняемый код (как это случилось с процедурой SETABORTPROC, поддерживающей функцию обратного вызова, о которой явно упоминалась в документации, но возможность эксплуатации которой в хакерских целях долгое время оставалась совершенно неочевидной).

Проще всего затыкаются дыры, связанные с отсутствием контроля длины обрабатываемых данных. Ошибки синхронизации в отсутствии исходных текстов исправить практически нереально, но… если уязвимая функция некритична для работы системы, на нее можно поставить "заглушку", лишившись при этом части функциональности, но зато надежно защитив систему.

с чего начать или анализ уязвимости

Прежде, чем приступать к латанию дыры, нужно локализовать ее местоположение, то есть проанализировать уязвимость, собрав всю доступную информацию (а недоступную получить самостоятельно). Сведения, приводимые на сайте Microsoft, в подавляющем большинстве случаев носят слишком расплывчатый характер, не позволяющий ни воспроизвести атаку, ни даже понять в чем суть проблемы. Сайты типа Security Focus намного более "откровенны" и помимо exploit'ов очень часто приводят ссылки на оригинальные хакерские публикации со всеми техническими подробностями.

Имея на руках код exploit'а мы можем легко разобраться каким именно образом осуществляется атака и на какие системные компоненты направлен хакерский удар. В идеале, это будет прямой вызов API/RPC-функции с передачей необычных параметров (например, чрезмерно длинной строки), тогда нам останется только модифицировать машинный код, дополнив его одной или несколькими проверками.

```
memory_size = MB * atoi(argv[2])
interface = ('spoolss', '12345678-1234-abcd-ef00-0123456789ab', '1.0')

class GetPrinterData(Structure):
        alignment = 4
        opnum = 26
        structure = (
        ('handle', '%s'),
        ('value', ':', B2),
        ('offered', '<L'),
)

query = OpenPrinterEx()
printer = "\\\\%s\x00" % (host)
query['printer'] = B1()
query['printer']['id'] = 0x41414141
query['printer']['max'] = len(printer)
query['printer']['actual'] = len(printer)
query['printer']['str'] = printer.encode('utf_16_le')
…
query = GetPrinterData()
value = "blah_blah\x00"
query['handle'] = handle
query['value'] = B2()
query['value']['max'] = len(value)
query['value']['actual'] = len(value)
```

```
        query['value']['str'] = value.encode('utf_16_le')
        query['offered'] = memory_size
```

Листинг 1 фрагмент exploit'a, вызывающего функцию GetPriterData, принимающую в качестве одного из аргументов размер блока памяти (memory_size), который необходимо выделить для записи данных о конфигурации принтера, отсутствие проверки корректности этого параметра позволяет атакующему запросить блок памяти, который не может быть выделен в принципе, в результате чего происходит "падения" суплера печати (полный исходный текст exploit'a лежит на http://downloads.securityfocus.com/vulnerabilities/exploits/21404.py)

Хуже, если exploit представляет собой "готовый" документ, графический файл или программу отправляющую на такой-то порт непонятную структуру данных, вызывающую отказ в обслуживании или переполнение буфера с передачей управления на shell-код.

Существуют по меньшей мере два пути решения проблемы: тщательно проанализировав каждое поле структуры/документа, мы рано или поздно найдем в нем что-то "необычное", если, конечно, данная структура документирована и у нас на руках имеется спецификация отчетливо описывающая назначение каждого поля. В противном случае, анализ превращается в гадание на кофейной гуще, а, учитывая, что Microsoft не очень-то стремиться документировать "внутренности" своего программного обеспечения, гадать придется помогу и очень часто.

```
<html>
        <a href="mhtml://mid:AAA...AAAA">example</a>
</html>
```

Листинг 2 фрагмент exploit'a, поражающего IE посредством слишком длинной сроки href="mhtml://mid:…", которая сразу же бросается в глаза при его просмотре (полный текст находится на www.securityfocus.com/data/vulnerabilities/exploits/18198.htm)

Как вариант, можно воспользоваться отладчиком, установив в начало shell-кода программную точку останова, представляющую собой код CCh, тогда при передаче управления на shell-код отладчик "всплывет" и можно будет узнать в какой функции мы находимся просто посмотрев на стек и изучив лежащие там адреса возврата. Однако, если стек разрушен при переполнении, то адреса материнской функции там уже не окажется и в лучшем случае будут присутствовать адреса пра-пра-материнских функций, лежащие в более старших адресах за концом shell-кода. В худшем же случае, регистр указатель вершины стека (ESP) окажется искажен и перенаправлен на совершенно постороннюю область памяти. Как же тогда определить в какой именно функции произошло переполнение?!

Рисунок 1 просмотр стека в отладчике OllyDbg

Тут есть разные способы. Например, запомнив адрес начала shell-кода в памяти, можно установить на него аппаратную точку останова на чтение/запись памяти и запустить exploit повторно. Поскольку, стек используется довольно интенсивно, точка останова будет срабатывать очень часто и чтобы не отвлекать себя лишними всплытиями отладчика, стоит прибегнуть к условной точке останова (отладчики soft-ice и OllyDbg это позволяют), срабатывающую лишь при записи в память определенных данных — т.е. самого shell-кода. В качестве сигнатуры достаточно использовать первые восемь байт.

Но положение shell-кода в памяти может варьироваться от одного запуска exploit'a к другому (особенно, если он находится не в стеке, а динамической памяти, так же называемой кучей, она же — heap). Ничего не остается, как ставить точку останова на команду call (о том как это сделать рассказывается в моей статье "хакерские трюки или как поставить бряк на jmp eax", лежащей на http://nezumi.org.ru/jump-eax.7z) и отслеживать все вызовы функций один за другим, дожидаясь наступление переполнения.

Рисунок 2 отладка в Visual Soft-Ice

Если же это не поможет…. тогда…. тогда думать надо! Готовых рецептов для особо сложных ситуаций не существует!

```
byte[] shellcode = new byte[]
{
```

```
        (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90,
        (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90,
        (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90,

        // shellcode copied from metasploit to run calc.exe
        (byte)0xCC, (byte)0xc9, (byte)0x83, (byte)0xe9, (byte)0xde, (byte)0xd9,
        (byte)0xee, (byte)0xd9, (byte)0x74, (byte)0x24, (byte)0xf4, (byte)0x5b,
        (byte)0x81, (byte)0x73, (byte)0x13, (byte)0x38,
…
};
```

Листинг 3 установка программной точки останова путем замены первого байта shell-кода exploit'а на CCh, обратите внимание, что первый байт shell-кода (конечно же, имеется ввиду первый _актуальный_ байт shell-кода) не всегда совпадает с началом массива, хранящего shell-код. в частности, в данном случае, начало shell-кода "оккупировала" последовательность команд NOP, в середину которой и передается управление при переполнении (см. www.securityfocus.com/data/vulnerabilities/exploits/JvmGifVulPoc.java)

правка кода на диске

Достаточно широко распространена правка машинного кода непосредственно в исполняемом файле, динамической библиотеке или драйвере. Это простой и одновременно с тем надежный метод, реализуемый в hiew'e или другом hex-редакторе со встроенным ассемблером. "Затычки" на функции создаются путем внедрения машинной команды RET в начало функции, при этом, если функция следует stdcall соглашению (как это делают практически все API-функции), за командой RET должен стоять аргумент, указывающий сколько байт необходимо снять со стека после извлечения адреса возврата. Определить это значение можно либо умножением количества аргументов функции на их размер (4 байта для x86), либо путем "подсмотра" за настоящим RET n в дизассемблере.

Рисунок 3 правка файла в hiew'e

В 64-битной версии Windows, работающей на платформе AMD x86-64, наблюдается совершенно иная картина. Во-первых, все операнды и адреса там по умолчанию 64-разярные, а, во-вторых (и это самое главное!) параметры API-функций передаются не через стек, а через регистры. Первые четыре аргумента всех API-функций помещаются в RCX, RDX, R8 и R9 (регистры перечислены в порядке следования аргументов, крайний левый аргумент помещается в RCX). Остальные же кладутся в стек. Такой способ вызова функций называется **x86-64 fast calling conversion** (соглашение о быстрой передаче параметров для x86-64), и он достаточно подробно описан на блоге "The old new thing" в заметке "The history of calling conventions, part 5 amd64" (http://blogs.msdn.com/oldnewthing/archive/2004/01/14/58579.aspx), а так же в моей статье "архитектура x86-64 под скальпелем ассемблерщика" (http://nezumi.org.ru/x86-64.text.zip).

В процессе правки возникает неизбежный вопрос: где брать место для размещения машинных команд, выполняющих дополнительные проверки? Если объем внедряемого кода невелик, то можно поискать в файле свободное место между функциями, отведенное под выравнивание и поставить на него jmp из "латаемой" функции, а потом сделать jmp обратно. Количество прыжков может быть и больше одного, но практически создавать такой "раздробленный" код очень утомительно и проще становится переписать часть латаемой функции на оптимизированном ассемблере, чтобы выиграть немного места. Поскольку, Windows в основном написана на Си/Си++ и других языках высокого уровня, то простор для оптимизации есть практически всегда, правда, вытворять подобные вещи в hiew'e равносильно самоубийству и логичнее прибегнуть к ассемблерному транслятору, умеющему создавать двоичные файлы с произвольным базированием, например, к знаменитому FASM'у.

При этом следует особое внимание уделить перемещаемым элементам, т. е. тем ячейкам, содержимое которых модифицируется системным загрузчиком в процессе запуска файла/подключения динамической библиотеки. Адреса перемещаемых элементов содержатся в специальной секции PE-файла, которую можно просмотреть утилитой DUMPBIN, запущенной с ключом /RELOCATIONS (она входит в штатную поставку практически всех Windows-компиляторов), однако, опытные кодокопатели обходятся и без этого. В 99,999% случаев перемещаемыми элементами являются указатели на абсолютные адреса внутри файла, например, встретив в динамической библиотеке команду MOV EAX, [400000h] с вероятностью близкой к единице можно утверждать, что на месте операнда 400000h находится перемещаемый

элемент. В исполняемых файлах перемещаемые элементы встречаются реже, и тут уже без обращения к relocation table не обойтись.

При переписывании кода функции, машинные команды с перемещаемыми элементами должны оставаться на своих местах!!! Это легко сказать, но очень трудно сделать, особенно учитывая, что одни и те же ассемблерные команды могут ассемблироваться в различные машинные коды и потому никогда нельзя сказать наперед где какая команда окажется. Приходится либо программировать через директиву db (т.е. писать на смеси ассемблера и машинного кода), либо править таблицу перемещаемых элементов после переписывания "латаемой" функции. Оба способа имеют свои достоинства и недостатки, поэтому выбор того или иного из них определяется конкретными обстоятельствами.

Для модулей прикладного режима существует замечательная возможность поместить весь внедряемый код в динамическую библиотеку, вызываемую из "латаемой" функции. В этом случае, внедряемый код может быть написан на любом языке высокого уровня. Для драйверов такой способ, естественно, не подходит, поскольку в Windows отсутствует стандартная функция загрузки драйверов и приходится писать много кода, что просто нецелесообразно.

После внесения любых изменений в драйвер или системные динамические библиотеки, необходимо пересчитать их контрольную сумму, в противном случае, Windows просто сообщит об ошибке загрузки и остановит систему. Контрольная сумма рассчитывается утилитой EDITBIN, запущенной с ключом /RELEASE. Как и DUMPBIN, она входит в штатный комплект поставки практически всех Windows-компиляторов.

Так же нельзя забывать и про SFC, автоматически восстанавливающую измененные файлы, эталонные копии которых хранятся в каталоге WINNT\System32\dllcache. Естественно, все эти операции необходимо осуществлять, загрузившись с другого диска или консоли восстановления. Вполне подойдут для этой цели и Linux Live CD с поддержкой NTFS (например, KNOPPIX). Утилиты DUMPBIN и EDITBIN замечательно запускаются из-под Windows-эмулятора WINE, впрочем, саму процедуру правки удобнее все-таки осуществлять на Windows, а к Linux'у прибегать только для копирования измененных файлов в системный каталог и dllcache.

Еще одна маленькая деталь. При последующей установке обновлений от Microsoft, система, обнаружив, что файл изменен, вправе "заругаться" на неизвестную версию и отказать в обновлении. Так что всегда храните оригинальные копии исправленных файлов и перед установкой официальных обновлений выполняйте "откат".

Рисунок 4 использование KNOPPIX Live CD для переноса "заштопанных" файлов на системных раздел NTFS

правка кода в памяти

Достойной альтернативной правкой кода на диске является его правка… ну, да! в оперативной памяти! К сильной стороне этого способа относиться то, что не нужно создавать резервные копии системных файлов, совершая откат при установке обновлений, усмирять разбушевавшуюся SFC и т. д. и т. п. А к слабой – ненадежность и невозможность сигнализации о факте невозможность наложения заплатки, тем не менее, этот способ используется во многих неофициальных заплатах, в том числе и в уже упомянутом hot-fix'e Ильфака.

Реализуется он очень просто, можно даже сказать, тривиально. В ветку реестра HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs добавляется имя динамической библиотеки, которая с этого момента будет проецироваться на все прикладные процессы, причем не просто проецироваться, но еще и получать управление в точку входа (DllEntryPoint) _до_ запуска процесса.

Убедившись, что мы спроецированы на "наш" исполняемый файл (что легко определяется по сигнатуре) правив его в памяти, предварительно присвоив атрибуты записи вызовом API-функции VirtualProtect, а по окончании правки, восстанавливаем статус-кво, повторным вызовом VirtualProtect.

Заплатки на динамические библиотеки накладываются по чуть-чуть более сложной схеме: сначала мы загружаем требуемую библиотеку через LoadLibrary, возвращающую базовый адрес ее загрузки, используем его для правки по описанному выше сценарию. Отслеживать загрузку динамической библиотеки процессом не нужно — будучи однажды загруженной в его адресное пространство, повторно она уже не загружается и выполненные нами изменения не теряются. Правда, слегка возрастают накладные расходы: если раньше все

процессы разделяли одну и туже динамическую библиотеку, то сейчас модифицированные страницы памяти "расщепляются", переходя в личное пользование каждого процесса.

Допустим, мы имеем сто запущенных процессов, тогда при модификации одной страницы в одной динамической библиотеке мы теряем sizeof(MEMORY_PAGE)*100 = 400 Кб памяти плюс память, занимаемую нашей библиотекой-загрузчиком, прописанной в AppInit_DLLs (поскольку, она разделяется всеми процессами, умножать ее размер на сто не нужно). Другими словами, максимально возможные потери навряд ли перешагнут за черту одного мегабайта. По современным понятиям — это ничтожное количество памяти (особенно для системы в которой одновременно работает порядка ста процессов).

Единственная проблема в том, что ветка AppInit_DLLs находится под строгим надзором множества антивирусов, персональных брандмауэров и других защитных систем, которые зачастую удаляют оттуда все вновь появившееся библиотеки без всяких запросов на подтверждение! А если наша библиотека будет удалена, заплатка окажется не наложенной, но… об этом никто не узнает! Так что правка файла на диске при всех своих недостатках все же _надежнее_

заключение

Собственноручно изготовленную заплатку рекомендуется использовать только на своих машинах и не пытаться распространять ее, поскольку для этого пришлось бы учитывать весь существующий парк Windows-систем со всеми комбинациями пактов обновления, в противном случае заплатка может не привести к аварийной остановке загрузки системы, после чего восстановить работоспособность сможет далеко не всякий пользователь.

Впрочем, здесь все зависит от вашего опыта и умения программировать.

Рисунок 5 тестирование заплаток на виртуальных машинах типа VM Ware поможет избежать краха основной системы