Техника снятия дампа с защищенных приложений

Крис Касперски

В прошлой статье этого цикла мы прошли сквозь распаковщик, добравшись до оригинальной точки входа, и теперь, чтобы окончательно победить защиту, нам необходимо снять дамп. Существует множество утилит, предназначенных для этой цели, но далеко не всегда полученный дамп оказывается работоспособным. Почему? Попробуем разобраться!

Протекторы типа Themida (в «девичестве» eXtreme Protector) и Star-Force, которыми защищены многие популярные программы, очень глубоко «вгрызаются» в операционную систему, что снижает производительность и порождает частные BSOD. Их «коллеги» ведут себя не так агрессивно, но проблем с совместимостью все равно хватает, особенно при переходе на 64-разрядные операционные системы или многоядерные процессоры, значительно отличающиеся от тех, под которые проектировалась защита, активно использующая недокументированные возможности. Уж сколько раз твердили миру – не используйте ничего недокументированного в коммерческих приложениях, да только все не впрок! Вот и приходится браться за хакерский инструментарий и освобождаться от протекторов, даже когда программа куплена легальным путем и «ломать» ее незачем. А ведь приходится! Как странно устроен мир.

Простые случаи дампинга

Представим себе, что распаковщик уже отработал, программа остановлена в оригинальной точке входа (OEP) и мы готовы сохранить образ файла (file image) на диск, то есть сбросить дамп. Отладчик soft-ice не предоставляет такой возможности, поэтому приходится действовать обходным путем..

Но для начала необходимо выяснить, какой именно регион памяти необходимо сохранять. При условии, что защита не предпринимает никаких враждебных действий, нужная информация может быть добыта командами MOD и MAP32 (см. **рис. 1** и **листинг 1**).

Листинг 1. Определение дислокации модуля в памяти. Здесь «test_dump» – имя процесса, с которого мы собираемся снять дамп (soft-ice отображает его в правом нижнем углу)

```
# определяем базовый адрес загрузки модуля в память
:MOD  test_dump

hMod Base     PEHeader    Module Name     File Name
00400000      004000D0    test_dum        \TEMP\test_dump.exe

# смотрим на карту модуля в памяти
:MAP32 test_dump

Owner      Obj Name    Obj#    Address          Size       Type
test_dump  text        0001    001B:00401000    00003B46   CODE  RO
test_dump  rdata       0002    0023:00405000    0000080E   IDATA RO
test_dump  .data       0003    0023:00406000    00001DE8   IDATA RW
```

Базовый адрес загрузки (hMod base) располагается по адресу 400000h. Последняя секция (.data) начинается с адреса 406000h и продолжается вплоть до адреса (406000h + 1DE8h) == 407DE8h. Таким образом, нам необходимо сохранить 7DE8h байт памяти, начиная с 400000h. Но в soft-ice такой команды нет! Зато есть «история команд». Даем команду «DB 400000 L 7DE8», выходим из отладчика, запускаем symbol loader и «говорим»: «file → save soft-ice history as» (при этом размер самой истории должен быть предварительно увеличен хотя бы до 30 МБ: «edit → soft-ice initialization setting → history buffer size»). В результате образуется текстовый файл (см. **листинг 2**), который необходимо преобразовать в exe, для чего потребуется написать специальную утилиту или поискать уже готовую.

Листинг 2. Дамп памяти, снятый через history

```
:db 400000 L 7DE8

010:00400000 4D 5A 90 00 03 00 00 00-04 00 00 00 FF FF 00 00  MZP.............
010:00400010 B8 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00  ........@.......
010:00400040 0E 1F BA 0E 00 B4 09 CD-21 B8 01 4C CD 21 54 68  ........!..L.!Th
010:00400050 69 73 20 70 72 6F 67 72-61 6D 20 63 61 6E 6E 6F  is program canno
010:00400060 74 20 62 65 20 72 75 6E-20 69 6E 20 44 4F 53 20  t be run in DOS
010:00400070 6D 6F 64 65 2E 0D 0D 0A-24 00 00 00 00 00 00 00  mode....$.......
```

Как вариант, можно воспользоваться бесплатным плагином IceExt (http://stenri.pisem.net), значительно расширяющим функциональные возможности soft-ice. Если IceExt откажется запускаться, увеличьте размер кучи и стека до 8000h байт, отредактировав следующую ветвь реестра HKLM\SYSTEM\CurrentControlSet\Services\NTice.

Команда !DUMP (см. **рис. 2** и **листинг 3**) позволяет сохранять блоки памяти на диск в двоичном виде, что очень удобно:

Листинг 3. Дамп памяти, снятый командой !DUMP плагина IceExt

```
:!DUMP

Dump memory to disk
!dump FileName Addr Len
Ex:
 !dump c:\dump.dat 400000 1000
 !dump \??\c:\dump.dat 400000 1000
 !dump \??\c:\dump.dat edx+ebx ecx

:!DUMP C:\dumped 400000 7DE8

DUMP: \??\C:\dumped 400000 7de8
```

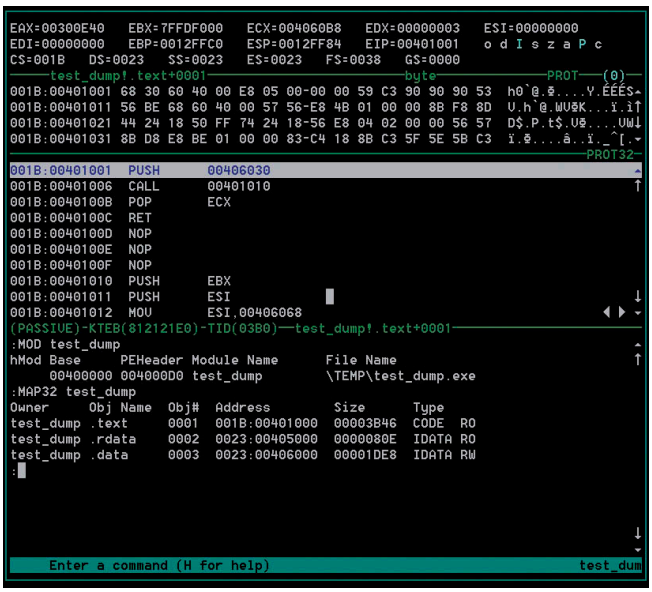

Рисунок 1. Определение региона памяти для снятия дампа

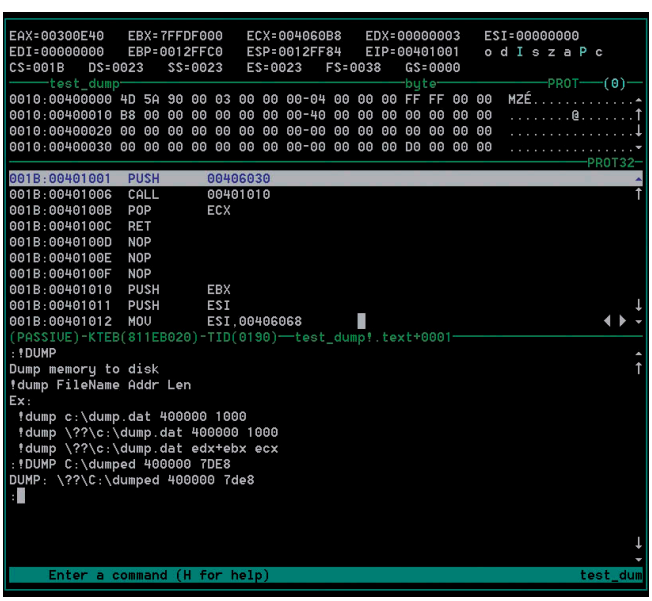

Рисунок 2. Снятие дампа в soft-ice с помощью плагина IceExt

Другой плагин — icedump (programmerstools.org/system/files?file=icedump6.026.zip) тоже умеет дампить память, и так же, как и IceExt, он распространяется на бесплатной основе вместе с исходными текстами.

Полученный дамп можно загрузить в дизассемблер типа IDA Pro, но от его запуска лучше воздержаться (особенно на соседних компьютерах), поскольку для корректной работы необходимо восстановить таблицу импорта и ресурсы, но это настолько обширный вопрос, что здесь мы не будем его касаться, тем более что существуют готовые утилиты: Import Reconstructor (http://www.wasm.ru/baixado.php?mode=tool&id=64) восстановит импорт, а Resource Rebuilder (http://www.wasm.ru/baixado.php?mode=tool&id=156) – ресурсы.

Для отладчика OllyDbg существует плагин OllyDump (http://dd.x-eye.net/file/ollydump300110.zip) со встроенным реконструктором таблицы импорта (см. **рис. 3**).

Наконец, можно воспользоваться и автономным дампером. Самым первым (и самым неумелым) был ProcDump,

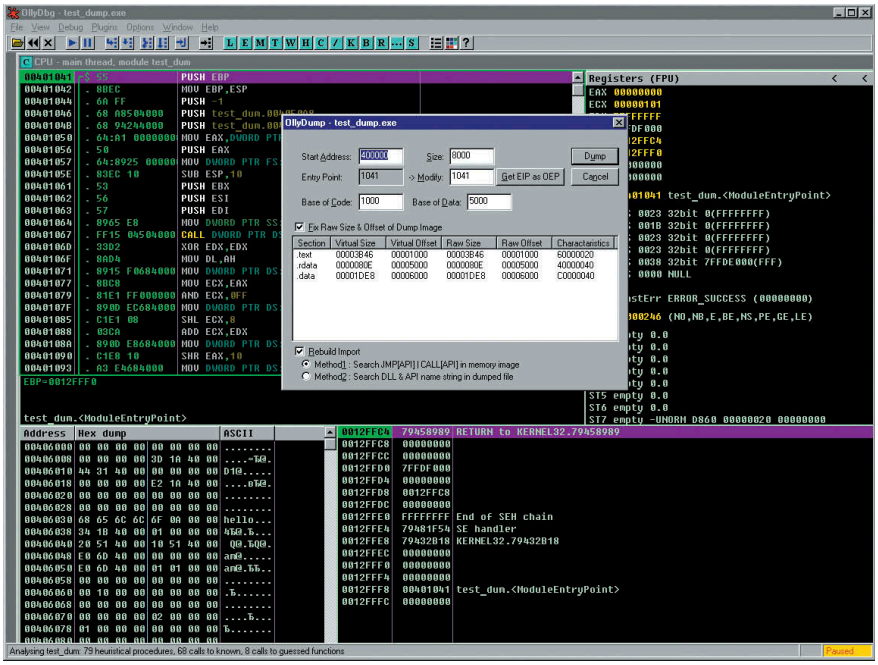

Рисунок 3. Снятие дампа в OllyDbg с помощью плагина OllyDump

затем появился Lord PE, учитывающий горький опыт своего предшественника и способный сохранять дамп даже в тех случаях, когда PE-заголовок умышленно искажен защитной, а доступ к некоторым страницам памяти отсутствует (атрибут PAGE_NOACCESS). Венцом эволюции стал PE-TOOLS (см. **рис. 4**), с которым мы и будем работать. Базовый комплект поставки можно найти практически на любом хакерском сервере, например, на WASM (http://www.wasm.ru/baixado.php?mode=tool&id=124) или на CrackLab (http://www.cracklab.ru/download.php?action=get&n=MTU1), а свежие обновления лежат на «родном» сайте проекта http://neox.iatp.by, кстати говоря, уже несколько раз поменявшим свой адрес (по непонятым причинам базовый пакет на нем отсутствует).

Если мы работаем с отладчиком прикладного уровня (типа OllyDbg) и стоим в OEP, то снять дамп с программы очень просто. Достаточно переключиться на PE-TOOLS, выбрать нужный процесс в списке и сказать «dump full», однако, с отладчиками уровня ядра (soft-ice, Microsoft Kernel Debugger) этот номер не проходит и тут приходится хитрить. Запоминаем (в голове или на бумажке) первые два байта от начала OEP и записываем сюда EBFEh, что соответствует машинной инструкции JMP short $-2, зацикливающей процесс. Теперь можно смело выходить из отладчика, идти в PE-TOOLS, снимать дамп и восстанавливать оригинальные байты в любом hex-редакторе.

Обыкновенные упаковщики (типа UPX) не сопротивляются снятию дампа, поскольку борьба с хакерами в их задачу не входит. Иное дело – протекторы. Фактически это те же самые упаковщики, но снабженные целым арсеналом систем, противодействующих взлому. Защитные методики можно разделить на активные и пассивные. К пассивным относятся все те, что работают только на стадии распаковки и не вмешиваются ни в саму программу, ни в операционную систему. Активные – перехватывают API-функции внутри адресного пространства защищаемого процесса или даже устанавливают специальный драйвер, модифицирующий ядро операционной системы таким образом, что прямое снятие дампа становится невозможным. Очевидным побочным эффектом активных защит ставится их неуживчивость с новыми версиями Windows, зачастую приводящая к краху операционной системы. Печальнее всего то, что, запуская setup.exe, мы даже и не подозреваем, что там может оказаться, да и далеко не все протекторы поддерживают корректурную деинсталляцию...

В поисках самого себя

Снятие дампа начинается с определения региона памяти, принадлежащего исполняемому файлу (или динамической библиотеке). Приемы, описанные выше, всецело опираются на PE-заголовок и таблицу секций, используемую операционной системой практически только на стадии загрузки файла. В частности, нас интересуют поля ImageBase (адрес базовой загрузки), SizeOfImage (размер образа) и содержимое таблицы секций. Протекторы любят затирать эти поля после завершения распаковки или подсовывать заведомо некорректные значения. Дамперы первого поколения от этого «сходили с ума», но PE-TOOLS в большинстве случаев восстанавливает недостающую информацию самостоятельно, но иногда он все-таки не срабатывает. И что тогда?

Самое простое – исследовать карту памяти подопытного процесса, возвращаемую API-функциями VirtualQuery/VirtualQueryEx. Регионы, помеченные как MEM_IMAGE, принадлежат исполняемому файлу или одной из используемых им DLL (в PE-TOOLS за построение карты отвечает команда «dump region» (см. **рис. 5**)).

Активные защиты могут перехватывать эти функции, подсовывая подложные данные, и тогда приходится спускаться на один уровень вглубь, обращаясь к функции NtQueryVirtualMemory, которая, как и следует из ее названия, существует только в NT-подобных операционных системах и экспортируется библиотекой NTDLL.DLL, но в некоторых случаях перехватывается и она, вынуждая нас обращаться к функции NtQuerySystemInformation. Долгое время она оставалась совершенно недокументированной и многие протекторы о существовании подобной лазейки даже и не подозревали. Теперь же ее описание доступно на MSDN: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sysinfo/base/ntquerysysteminformation.asp, с грозным предупреждением, что поведение функции может измениться в любой новой версии, а потому в долгосрочных продуктах на нее лучше не закладываться.

В самом крайнем случае (когда перехвачена и NtQuerySystemInformation) приходится прибегать к последовательному разбору структур данных, относящихся к памяти процесса (soft-ice именно так и поступает), однако гораздо проще и надежнее просто скопировать кусок адресного пространства. Если образ не был переме-

щен, базовый адрес загрузки тот же самый, что и в PE-заголовке exe-файла. При работе с перемещенным образом, базовый адрес приходится определять вручную путем поиска сигнатур PE и MZ, двигаясь от OEP вверх (т.е. в сторону младших адресов). К нашему счастью, полностью затереть PE-заголовок защита не может, поскольку тогда перестанут работать некоторые API-функции, взаимодействующие с ресурсами и т. д.

Если ни одним из способов определить границы образа не удается, приходится дампить фрагменты адресного пространства, загружая их в IDA Pro как двоичный файл, естественно, с сохранением начального адреса фрагмента. Для анализа работы защитного механизма этого в большинстве случаев оказывается вполне достаточно, тем более что IDA Pro позволяет подгружать недостающие части «налету».

Дамп извне

Прежде чем читать адресное пространство чужого процесса, до него еще предстоит добраться. Windows изолирует процессы друг от друга на случай непреднамеренного удара по памяти (который сильно досаждал пользователям Windows 3.x), но предоставляет специальный набор API-функций для межпроцессорного взаимодействия. Классический путь: получаем обработчик процесса, который мы собрались дампить вызовом OpenProcess и передаем его функции ReadProcessMemory вместе с остальными параметрами (откуда и сколько байт читать). При этом необходимо учитывать, что некоторые страницы могут быть помечены защитой как недоступные и перед обращением к ним необходимо вызвать функцию VirtualProtectEx, разрешив полный доступ (PAGE_EXECUTE_READWRITE) или, по крайней мере, открыв страницы только на чтение (PAGE_READONLY).

Естественно, функции OpenProcess/ReadProcessMemory/VirtualProtectEx могут быть перехвачены защитой, и тогда вместо дампа мы получим error, а то и reboot. Низкоуровневые функции NtOpenProcess/NtReadVirtualMemory/NtProtectVirtualMemory перехватываются с той же легкостью, к тому же некоторые защиты изменяют маркер безопасности процесса, запрещая открытие его памяти на чтение даже администратору!

Считается, что снятие дампа на уровне ядра открывает большие возможности для реверс инжиниринга и противостоять этому никак невозможно, поскольку драйвер работает с наивысшим уровнем привилегий, который позволяет все. Но ведь и драйверу защиты, работающему на уровне ядра, тоже доступно все, в том числе и модификация ядра операционной системы, в которых нуждается драйвер дампера. Причем никаких документированных функций для чтения памяти чужого процесса (за исключением вышеупомянутых) в системе нет!

Чтобы читать память процесса напрямую, драйвер должен к нему подключиться, вызвав функцию KeAttachProcess или ее современный аналог KeStackAttachProcess, появившийся и впервые документированный в Windows 2000. Пользоваться обеими функциями следует с величайшей осторожностью и прежде, чем подключаться к другому процессу, необходимо отсоединиться от текущего, вызвав KeDetachProcess/KeStackDeattachProcess. Однако эти функции могут быть перехвачены защитой со всеми вытекающими отсюда последствиями (протектор Themida именно так и поступает).

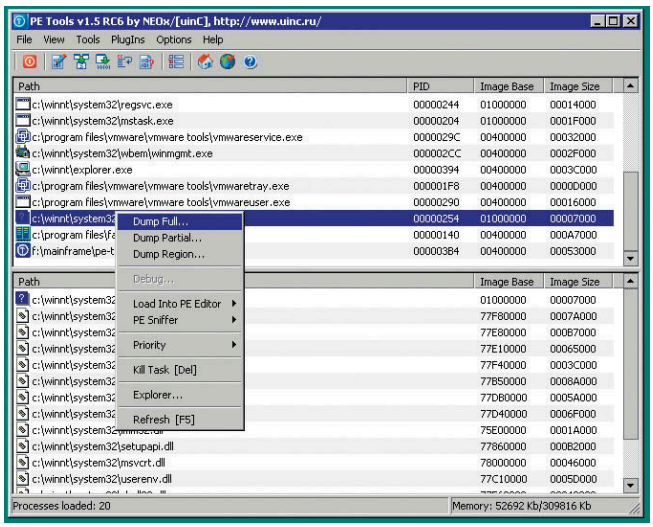

Рисунок 4. Внешний вид утилиты PE-TOOLS

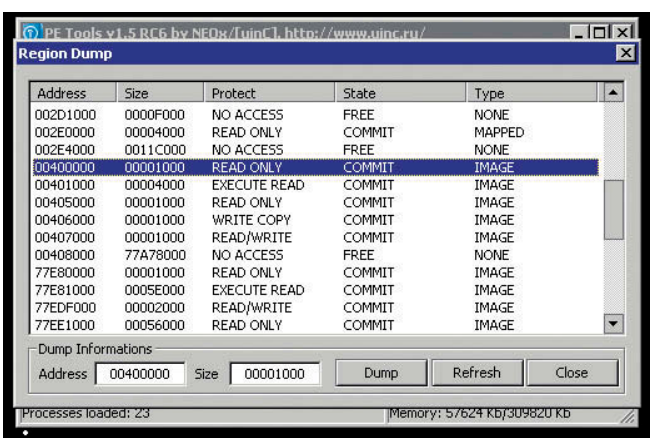

Рисунок 5. Просмотр карты памяти в PE-TOOLS

Важно отметить, что универсальных способов перехвата не существует – протектор может модифицировать таблицу экспорта, внедрять свои jmp в начало или даже середину сервисных функций ядра и т. д. А это значит, что на ядро полагаться нельзя и вариантов у нас только два: использовать те функции, которые не догадалась перехватить защита, или переключать адресные пространства вручную.

Специальный плагин к PE-TOOLS (http://neox.iatp.by/eXtremeDumper.zip, http://www.wasm.ru/pub/21/files/dumping/eXtremeDumper.rar), написанный MS-REM, пробивается к процессу через следующую цепочку сервисных вызовов PsLookupProcessByProcessId → ObOpenObjectByPointer → ObDereferenceObject, которую пока еще никто не перехватывает, что позволяет снимать дамп даже с очень сильно защищенных программ, однако, сколько этот способ еще продержится, сказать невозможно. Создатели протекторов не сидят сложа руки и на хакерских форумах тоже бывают.

В долговременной перспективе надежнее всего использовать недокументированную (и к тому же неэкспортируемую!) функцию KiSwapProcess, адрес которой меняется от системы к системе, что затрудняет перехват. В то же время дампер может легко определить его посредством таблицы отладочных символов, бесплатно распространя-

емых Microsoft. Для работы с ними понадобится библиотека dbghelp.dll из комплекта Debugging Tools (http://www.microsoft.com/whdc/devtools/debugging/default.mspx) и утилита symchk.exe, взятая оттуда же.

Функция KiSwapProcess – это одна из самых низкоуровневых функций, напрямую работающих с регистром CR3, в который заносится указатель на каталог страниц выбранного процесса, после этого его адресное пространство можно читать как свое собственное машинной командой MOVSD, в грубом приближении представляющий собой аналог memcpy. Предвидя такой исход событий, некоторые защиты пошли на отчаянный шаг: перехватив SwapContext и ряд других функций, работающих с CR3, они стали разрушать каталог страниц «своего» процесса на время переключения контекстов и вновь восстанавливать его, когда в нем возникает необходимость. Настоящее варварство! Обращение к каталогу страниц происходит из десятков недокументированных функций, которые в каждой версии ядра реализованы по своему. А это значит, что такая агрессивная защитная политика рискует «свалиться» в сплошной BSOD, не оставляющий пользователю никаких шансов для нормальной работы!

Но это еще не самое страшное. Все больше и больше протекторов переходят на динамическую распаковку, расшифровывая страницы по мере обращения к ним, а затем зашифровывая их вновь. Даже если мы пробьемся сквозь защиту и дорвемся до процесса, дампить будет нечего... вернее, полученный дамп будет практически на 99% зашифрован.

Механизмы динамической расшифровки

Алгоритм динамической расшифровки, реализованный в протекторе Armadillo и известный под именем CopyMem, в общих чертах выглядит так: защита порождает отладочный процесс, передавая функции CreateProcess в качестве имени нулевой аргумент командной строки, «благодаря» чему в диспетчере задач отображаются две копии запущенной программы. Одна из них – сервер (условно), другая – клиент. Сервер посредством функции VirtualProtectEx делает все страницы клиента недоступными (атрибут PAGE_NOACCESS) и передает ему управление, ожидая отладочных событий с помощью функции WaitForDebugEvent, а события долго ждать себя не заставляют и при первой же попытке выполнения кода в недоступной странице возбуждается исключение, передающее серверу бразды правления. Сервер расшифровывает текущую страницу, взаимодействуя с клиентом посредством API-функций ReadProcessMemory/WriteProcessMemory, устанавливает необходимые атрибуты доступа и возвращает клиенту управление.

Остальные страницы остаются зашифрованными, и при обращении к ним вновь возбуждается исключение, которое передается серверу через WaitForDedugEvent. Сервер зашифровывает предыдущую страницу, отбирая все атрибуты доступа, какие у нее только есть, и расшифровывает текущую страницу, возбудившую исключение (в действительности для увеличения производительности защита поддерживает примитивный кэш, позволяя клиенту иметь несколько расшифрованных страниц одновременно).

«Нечестные» защитные приемы

Нормальные упаковщики (UPX, PKLITE, PECOMPACT) сжимают исполняемый файл без потерь, и после завершения распаковки он возвращается к своему первоначальному виду, что делает процесс снятия дампа тривиальной задачей. Протекторы в стремлении усилить защиту зачастую идут на довольно рискованный шаг – они слегка «корежат» обрабатываемый файл с таким расчетом, чтобы он мог работать только под протектором, а после освобождения от него становится нежизнеспособным. Наиболее популярные способы подобной «нечестной» защиты рассмотрены ниже.

Кража байт с OEP. Самая простая и широко распространенная подлянка, используемая даже в таких безобидных протекторах как, например, ASProtect. Суть ее заключается в том, что упаковщик «крадет» несколько инструкций из оригинальной точки входа, сохраняет их в потайном месте (возможно, в замаскированном или зашифрованном виде), а после завершения распаковки эмулирует выполнение краденых байт. Чаще всего для этой цели используется стек (тогда краденые байты обычно становятся операндами инструкций PUSH), реже – прямое воздействие на регистры и память (при этом краденые инструкции трансформируются в псевдокод и в явном виде нигде не сохраняются). Суть в том, что в точке входа распакованного образа оригинальных байт уже не оказывается и снятый дамп становится неработоспособным. К нашему счастью, подавляющее большинство программ начинается со стартового кода, который является частью библиотеки времени исполнения (RTL), поставляемой вместе с компиляторами. Используя оставшийся «хвост» стартового кода, мы легко отождествим компилятор и восстановим краденые байты из его библиотеки. Если же данного компилятора в нашем распоряжении не окажется, первые несколько байт стартового кода в 9 из 10 случаев вполне предсказуемы, и зачастую их удается восстановить самостоятельно (естественно, для этого необходимо иметь опыт работы с различными RTL). Кстати, IDA Pro распознает компилятор именно по первым байтам стартового кода, и, если они отсутствуют или искажены, механизм FLIRT работать не будет. Это значит, что мы останемся без имен библиотечных функций и процесс дизассемблирования займет намного больше времени.

Полиморфный мусор в OEP. Вместо того чтобы красть байты с OEP, некоторые протекторы предпочитают модифицировать стартовый код, разбавляя значимые инструкции бессмысленным полиморфным мусором. Это никак не влияет на работоспособность снятого дампа, но ослепляет «FLIRT», вынуждая нас либо вычищать полиморфный мусор, либо определять версию компилятора «на глазок», загружая сигнатуры вручную (IDA Pro это позволяет).

Переходники в таблице импорта к куче. Протектор Themida использует следующий прием, серьезно затрудняющий восстановление таблицы импорта. Непосредственные адреса API-функций заменяются переходниками на область памяти, выделенную VirtualAlloc (т.е. кучу), которая по умолчанию в дамп не попадает, поэтому восстанавливать импорт приходится вручную. Это несложно, но утомительно – ищем вызовы API-функций, ведущие к куче (то, что это именно куча, а не что-то другое, можно определить по карте), дампим соответствующий регион памяти на диск, удаляем переходники, заменяя их действительными адресами, после чего запускаем Import Reconstructor или другую утилиту аналогичного назначения и... нет, это еще не все! Это только начало!

Потребность в отладочном процессе-сервере объясняется тем, что по другому ловить исключения на прикладном уровне просто не получается. А как же механизм структурных исключений или, сокращенно, SEH? Регистрируем свой собственный обработчик и ловим исключения, что называется по месту возникновения. Это избавляет нас от API-вызовов, обеспечивающих межпроцессорное взаимодействие, которые элементарно перехватываются хакером. Увы! Если защищаемое приложение использует SEH (а подавляющее большинство приложений его используют), наш обработчик окажется перекрыт другим. Столкнувшись с «нашим» исключением, он попросту не будет знать, что с ним делать, и с вероятностью, близкой к единице, просто завершит приложение в аварийном режиме.

Теоретически установку нового обработчика легко отследить, установив аппаратную точку останова по доступу к памяти на адрес FS:[00000000h]. Операционные системы семейства NT позволяют прикладным приложениям манипулировать с отладочными регистрами через контекст, причем отладочный регистр действует лишь в рамках «своего» процесса, не мешая работать всем остальным, но 9x «забывает» сохранять отладочные регистры в контексте «своего» процесса, и они приобретают глобальный характер, воздействующий на все процессы! Так что в ней этот трюк не проходит.

А вот другой способ: устанавливаем драйвер, перехватывающий исключения на уровне IDT и взаимодействующий со своим процессом либо через DeviceIoControl, либо через NtReadVirtualMemory/NtWriteVirtualMemory/KeDeattachProcess/KeAttachProcess. Это вполне надежно, однако написание драйверов – занятие утомительное и совсем небезопасное в плане «голубых экранов смерти». К тому же разработчику защиты придется либо наотрез отказываться от поддержки 9x (которая все еще жива!), либо реализовывать сразу два драйвера! Тем не менее защиты такого типа все-таки встречаются. Независимо от того, как происходит обработка исключения, сломать такую защиту очень просто! Читаем первую страницу, дожидаемся завершения расшифровки, сохраняем ее на диск, обращаемся к следующей странице и... действуем так до тех пор, пока в наших руках не окажется весь образ целиком. Стоит только внедрить код дампера в адресное пространство защищенного процесса… и протектору будет очень сложно отличить обращения самой программы от обращений дампера.

Последние версии протектора Armadillo, недавно переименованного в Software Passport, реализуют намного более надежный, хотя и чрезвычайно низко производительный механизм трассирующей расшифровки, при котором весь код программы зашифрован целиком. Сервер трассирует клиента, расшифровывая по одной инструкции за раз (предыдущая инструкция при этом зашифровывается). Снять дамп тупым обращением к памяти уже не получается, поскольку защиту интересуют только исключения, возникающие при исполнении. Все, что мы можем, это «вклиниться» между зашифрованным приложением и расшифровщиком, «коллекционируя» расшифрованные инструкции, образующие трассу потока выполнения. Поскольку, достичь 100% покрытия кода практически невозможно, полученный дамп будет неполноценным, но тут есть один маленький нюанс. Помимо создания переходников некоторые функции копируются протектором целиком! Подробнее об этом приеме можно прочитать в статье «Точки останова на win32 API и противодействие им» – раздел «Копирование API – функций целиком», которая находится на http://kpnc.opennet.ru/adt.zip.

Замена jx с последующей эмуляцией. При «отвязке» программ от протектора Armadillo самое сложное – это восстановление оригинального кода программы. Защита дизассемблирует обрабатываемый файл, находит в нем условные и безусловные переходы, записывает поверх них команду INT 03h, а сам переход сохраняется в своей внутренней таблице переходов. Процесс-сервер перехватывает исключение, возбуждаемое инструкцией INT 03, «смотрит», откуда оно пришло, извлекает из таблицы соответствующий этому адресу переход и эмулирует его выполнение с помощью арифметических манипуляций с регистром флагов (то есть в явном виде переходы нигде не хранятся!). Вот три главных минуса такого решения: во-первых, нет никакой гарантии, что защита правильно дизассемблирует обрабатываемую программу и не спутает переход с другой командой; во-вторых, эмуляция требует времени, существенно снижая производительность и, наконец, в-третьих, от взлома это все равно не спасает! Дизассемблировав эмулятор переходов (а дизассемблировать его несложно) и обнаружив таблицу переходов, хакер в считанные минуты напишет скрипт для IDA Pro или OllyDbg, удаляющий все INT 03 и восстанавливающий оригинальные переходы. Существует даже полуавтоматический взломщик Armadillo, написанный двумя богами распаковки – infern0 и dragon (http://www.wasm.ru/baixado.php?mode=tool&id=220), в следующих версиях которого обещана полная автоматическая распаковка и дезактивация Armadillo.

Преобразование в байт-код. Протекторы Themida и Start-Force позволяют преобразовывать часть машинного кода защищаемой программы в язык виртуальной машины, то есть в байт-код (также называемый p-кодом). Если виртуальная машина глубоко «вживлена» внутрь протектора, то отломать защиту, не «умертвив» при этом приложение, становится практически невозможно, как невозможно непосредственно дизассемблировать байт-код. По меньшей мере для этого необходимо разобраться с алгоритмом работы виртуальной машины и написать специальный процессорный модуль для IDA Pro или свой собственный дизассемблер. Это очень трудоемкое занятие, отнимающее у исследователя кучу сил и времени, а ведь байт-код виртуальной машины в следующих версиях протектора может быть изменен и ранее написанный процессорный модуль/дизассемблер окажется непригодным. Это наиболее стойкая защита из всех, существующих на сегодняшний день, однако не стоит забывать о двух вещах: во-первых, если протектор становится популярным, а его новые версии выходят редко, создание процессорных модулей становится экономически оправданным и защиту начинают ломать все желающие, если же новые версии выходят чуть ли не ежедневно, то навряд ли у разработчика протектора будет достаточно времени для радикальной перестройки виртуальной машины и ему приходится ограничиваться мелкими изменениями байт-кода, которые выливаются в мелкие изменения процессорного модуля, и протектор продолжат ломать. Во-вторых, хакер может «отодрать» виртуальную машину от протектора, совершенно не вникая в тонкости интерпретации байт-кода, лишний раз подтверждая известный тезис: сломать можно все… со временем.

командная расшифровка не может использовать ни блочные, ни контекстно-зависимые криптоалгоритмы, поскольку трассирующий расшифровщик никогда не знает наперед, какая инструкция будет выполнена следующей. Остаются только потоковые алгоритмы типа XOR или RC4, которые очень легко расшифровать – стоит только найти гамму, которую протектор, несмотря ни на какие усилия, слишком глубоко запрятать все равно не сможет! Естественно, полностью автоматизировать процесс снятия дампа в этом случае уже не удастся и придется прибегнуть к дизассемблированию, а быть может, даже к отладке. К счастью, подобные схемы защиты не получили широкого распространения и навряд ли получат его в обозримом будущем. Трассировка замедляет скорость работы приложения в десятки раз, в результате чего оно становится неконкурентоспособным.

Дамп изнутри

Снятие дампа через механизмы межпроцессорного взаимодействия – это вчерашний день. Для борьбы с активными защитами хакеры внедряют код дампера непосредственно в «подопытный» процесс, что позволяет обойти как перехват API-функций, так и победить динамическую шифровку типа CopyMem.

Классический способ внедрения кода реализуется так: открываем процесс функцией OpenProcess, выделяем блок памяти вызовом VirtualAllocEx, копируем код дампера через WriteProcessMemory, а затем либо создаем удаленный поток функцией CreateRemoteThread (только на NT-подобных системах), либо изменяем регистр EIP в контексте чужого потока, обращаясь к SetThreadContext (действует на всех системах). Естественно, предыдущий EIP должен быть сохранен, а сам поток – остановлен.

Постойте! Но ведь это мало чем отличается от обычного межпроцессорного взаимодействия! Функции NtAllocateVirtualMemory/NtSetContextThread/NtCreateThread любая защита перехватит со смаком! (Никакой ошибки тут нет, API-функция CreateRemote Thread в действительности представляет собой «обертку» вокруг ядерной функции NtCreateThread.)

Хорошо, вот другой классический путь. Помещаем дампер в DLL и прописываем ее в HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs, в результате чего она будет отображаться на все процессы, какие только есть в системе, и перед передачей управления на очередной запускаемый процесс первой получит управление наша DLL! К сожалению, об этой ветке знают не только протекторы, но и другие программы (антивирусы, персональные брандмауэры) и следят за ней. Наша запись может быть удалена еще до того, как дампер приступит к работе! Если же ему все-таки удастся получить управление, первое, что он должен сделать, – выделить себе блок памяти внутри процесса, скопировать туда весь необходимый код и вернуть ветку AppInit_DLLs в исходное состояние. Поскольку дампер получает управление еще до того, как защита начнет работать, она никак не сможет обнаружить, что здесь кто-то уже побывал.

Исключение составляют активные защиты резидентного типа, постоянно присутствующие в системе даже ес-

ли защищаемый файл не был запущен. Но в этом случае они сталкиваются со следующей проблемой – как отличить «свой» процесс от всех остальных? По имени файла? Это не слишком надежно… Лучше использовать «метку» – уникальное сочетание байт по определенному адресу. С такими защитами справиться очень сложно, но все-таки возможно. Перепробовав несколько вариантов, автор остановился на следующем алгоритме, который обходит все существующие на сегодняшний день активные и пассивные защиты:

- Копируем оригинальный файл (с защитой) в tmp.tmp.
- Открываем оригинальный файл в hiew, переходим в точку входа (EP) и ставим jmp на свободное место, где и размещаем код дампера, который при получении управления осуществляет следующие действия:
  - выделяет блок памяти и копирует туда свое тело, обычно подгружаемое с диска (динамическую библиотеку лучше не загружать, поскольку некоторые защиты контролируют список DLL и если вызов идет из неизвестной динамической библиотеки, расценивают это как вторжение);
  - устанавливает таймер через API-функцию SetTimer с таким расчетом, чтобы процедура дампера получила управление, когда весь код будет полностью распакован или, в случае с CopyMem, когда защита успеет установить отладочный процесс (конечно, снять дамп в OEP в этом случае уже не получится, но даже такой дамп лучше, чем совсем ничего);
  - переименовывает оригинальный файл (тот, что исполняется в данный момент!) в tmp.xxx, а файлу tmp.tmp возвращает оригинальное имя;
  - вычищает себя из памяти, восстанавливает EP и передает управление защищенной программе;
  - если активная защита охраняет свой файл, опознавая его по сигнатуре, используем какой-нибудь безобидный упаковщик с «нулевым побочным эффектом» типа UPX, при этом все вышеуказанные действия следует выполнять на отдельной заведомо «стерильной» машине;
- Запускаем модифицированный файл на выполнение.

Таким образом, защита не сможет обнаружить изменений ни в файле, ни в памяти (при попытке определения имени текущего файла операционная система будет возвращать то имя файла, какое он имел на момент запуска, игнорируя факт его «онлайнового» переименования). Но это слишком громоздкий и навороченный алгоритм, к тому же активной защите ничего не стоит перехватить SetTimer и запретить установку таймера внутри «своего» процесса до завершения распаковки/передачи управления на OEP.

Забавно, но многие защиты забывают о функции SetWindowsHookEx, позволяющей внедрять свою DLL в адресное пространство чужого процесса. Впрочем, даже если бы они помнили о ней, осуществить корректный перехват весьма не просто. Многие легальные приложения (например, мультимедийные клавиатуры или мыши с дополнительными кнопками по бокам) используют SetWindowsHookEx для расширения функциональности системы. Не существует никакого способа отличить «честное» приложение от дампера. Защита может распознать факт внедрения чужой DLL в адресное пространство охраняемого ее процесса, но откуда ей знать, что эта DLL делает?! Можно, конечно, просто выгрузить ее из памяти (или воспрепятствовать загрузке), но какому пользователю понравится, что легально приобретенная программа конфликтует с его крутой клавиатурой, мышью или другим устройством? Так что SetWindowsHookEx при всей своей незатейливости – довольно неплохой выбор для хакера!

Самый радикальный способ внедрения в чужое адресное пространство – это правка системных библиотек, таких как KERNEL32.DLL или USER32.DLL. Править можно как на диске, так и в памяти, однако в последнем случае защита может легко разоблачить факт вторжения простым сравнением системных библиотек с их образом. Внедрившись в системную библиотеку, не забудьте скорректировать контрольную сумму в PE-заголовке, иначе NT откажется ее загружать. Сделать это можно как с помощью PE-TOOLS, так и утилитой rebuild.exe, входящей в состав SDK. Внедряться лучше всего в API-функции, вызываемые стартовым кодом оригинального приложения (GetVersion, GetModuleHandleA и т. д.), определяя «свой» процесс функцией GetCurrentProcessId или по содержимому файла (последнее – надежнее, т. к. GetCurrentProcessId может быть перехвачена защитой, которая очень сильно «удивится», если API-функция GetVersion неожиданно заинтересуется идентификатором текущего процесса). Во избежание побочных эффектов запускать такой дампер следует на «выделенной» операционной системе, специально предназначенной для варварских экспериментов и обычно работающей под виртуальной машиной типа BOCHS или VMWare.

Заключение

Правильно спроектированная и должным образом реализованная защита должна препятствовать нелегальному использованию программы, но не имеет ни морального, ни юридического права мешать честным пользователям и уж тем более вторгаться в операционную систему, производя никем не санкционированные изменения. Последние версии протекторов Themida и Software Passport вплотную приближаются к rootkit. Еще немного и они превратятся в настоящие вирусы, создание которых преследуется по закону.

Полезные ссылки:

1. **Современные технологии дампинга и защиты от него**. Отличная статья от создателя eXtremeDumper доступно рассказывающая о том, как протекторы защищаются от снятия дампа, и объясняющая, как эти защиты обойти (на русском языке): http://www.wasm.ru/article.php?article=dumping.
2. **Об упаковщиках в последний раз**. Объемный труд, созданный коллективом лучших отечественных хакеров во главе с легендарным Володей и охватывающий все аспекты работы упаковщиков, протекторов и самой операционной системы (на русском языке): http://www.wasm.ru/article.php?article=packlast01 (первая часть); http://www.wasm.ru/article.php?article=packers2 (вторая часть).
3. Касперски К. Генная инженерия на службе распаковки PE-файлов. – Журнал «Системный администратор», №5, май 2006 г. – 58-68 с.