САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

Овчинников Антон Андреевич

Восстановление памяти виртуальной машины из расширенного образа памяти базовой системы на платформе Windows

Дипломная работа

Допущена к защите.
Зав. кафедрой:
д.ф.-м.н., проф. Терехов А.Н.

Научный руководитель:
ст. преп. Губанов Ю.А.

Рецензент:
ст. преп. Зеленчук И.В.

Санкт-Петербург
2013

SAINT-PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics faculty

Software Engineering Chair

Anton Ovchinnikov

Recovery of virtual machine memory using extended host RAM-image on Windows platform

Graduation Thesis

Admitted for defence.
Head of the chair:
Dr. Sci, Prof. A.N. Terekhov

Scientific supervisor:
Ass. Prof. Yu.A. Gubanov

Reviewer:
Ass. Prof. I.V. Zelenchuk

Saint-Petersburg
2013

Оглавление

Введение

Компьютерный криминалистический анализ (computer forensics) представляет собой комплекс мероприятий, направленных на изучение вычислительных устройств и носителей информации в целях поиска и фиксирования доказательств (например, в ходе расследования гражданских или уголовных дел).

Типичный процесс экспертизы состоит из следующих основных шагов:

1. Снятие (acquisition) образа данных на цифровом устройстве.

Результатом этого шага может быть образ жесткого диска, файловой системы, образ памяти и т.п.

2. Анализ данных

Информация, представляющая интерес, может быть различной по виду и содержанию: история посещений веб-браузеров, журналы программ обмена мгновенными сообщениями, удаленные файлы различного типа и многое другое. Для извлечения и анализа данных могут применяться такие методы, как карвинг [7] или стохастическое моделирование [6].

3. Подготовка отчёта о собранных артефактах

По окончании этапа анализа формируется отчёт с подробным описанием всех использованных средств, методов, и полученными результатами.

До недавнего времени главным объектом изучения при компьютерном криминалистическом анализе были жесткие диски. Они являются долгосрочными хранилищами информации, а значит, данные не уничтожаются в течение длительного времени, и сам диск может быть представлен в качестве вещественного доказательства в ходе судебного процесса. Однако на данный момент с изъятием и изучением исключительно долгосрочных носителей данных связано несколько проблем, описанных далее.

Проблемы, связанные с анализом жесткого диска

1. Распространение облачных хранилищ данных.

Вся информация или ее часть может храниться на удаленных серверах в интернете, и изъятие этих данных может быть осложнено как с технической, так и с законодательной точки зрения. Подозреваемый может использовать свой компьютер (ноутбук, мобильное устройство) как терминал для доступа к данным, не загружая их на само устройство.

2. Распространение твердотельных накопителей (SSD, Solid State Drive)

Обладая многими достоинствами по сравнению с обычными жесткими дисками (HDD, Hard Disk Drive), твердотельные диски обладают особенностями функционирования,

значительно осложняющими восстановление удаленных файлов. Одна из проблем состоит в том, что SSD может самостоятельно очищать блоки памяти, если ОС помечает их как удаленные [8]. Это необходимо для подготовки к записи, потому что ячейки флеш-памяти должны быть очищены перед тем, как на них будут записаны новые данные (см. команду TRIM[1]).

3. Шифрование и запаковка файлов

Использование подозреваемым RAM-дисков, шифрованных контейнеров и разделов существенно препятствует анализу. Похожая ситуация наблюдается и с запакованными или обфусцированными файлами (например, запакованные вредоносные программы). Бурное развитие индустрии вредоносного программного обеспечения приводит к появлению новых, более совершенных методов упаковки (полиморфные упаковщики, протекторы[2]), целью которых является усложнение анализа.

4. Невозможность получения полного представления о системе

Анализируя жесткий диск компьютера, иногда трудно восстановить полную картину деятельности его обладателя. История браузеров, файлы журналов могут намеренно удаляться или их ведение может быть намерено отключено для уничтожение или сокрытия такой информации, как посещенные веб-сайты, использованные сетевые соединения и т. п.

Использование анализа памяти

Анализ образа памяти (memory forensics, live RAM forensics) в данный момент является важной частью криминалистического анализа. Образ памяти захватывается на работающей системе, после чего анализируется. В памяти может быть найдено большое количество информации, которую трудно или невозможно получить, располагая только жестким диском: расшифрованные файлы или пароли, распакованные вредоносные программы, открытые в момент захвата образа сетевые подключения.

Исследование запущенных виртуальных машин

Современные технологии виртуализации позволяют создавать и использовать независимые системы, которые в то же время будут работать на одном физическом оборудовании. Гипервизоры, работающие на основе базовой ОС (“хостовой”, host system), также называемые гипервизорами «второго типа», позволяют запускать произвольные операционные системы внутри уже запущенных. Данные средства могут быть использованы злоумышленниками для создания «чистой», легко настраиваемой среды, в которой можно

1 Команда TRIM: http://en.wikipedia.org/wiki/TRIM

2 Протектор – программа, преобразующая другую программу в вид, который осложняет исследование и отладку путем добавления антиотладочных приемов и шифрования файла.

осуществлять преступную деятельность, не оставляя улик на хостовой системе. А используя зашифрованный раздел для хранения файлов виртуальной машины на жестком диске, можно полностью скрыть ее наличие на компьютере.

Также средства виртуализации широко применяются в серверном сегменте. Использование классических методов исследования (как анализ историй и жестких дисков) в случае таких инцидентов как взлом и распространение вредоносного кода серьезно ограничивает эффективность анализа. А использованию анализа памяти в таких случаях препятствует наличие технологий виртуализации: на момент написания работы ни одна из имеющихся утилит для работы с образами памяти не умела извлекать информацию о виртуальных машинах, запущенных на хостовой системе.

Постановка задачи

Перед автором дипломной работы была поставлена задача предложить подход и реализовать анализ образа памяти на наличие запущенных виртуальных машин, и в случае их наличия производить извлечение данных о гостевой системе. В частности, одной из целей работы является извлечение оперативной памяти гостевой системы для дальнейшего применения стандартных методов анализа памяти.

Для успешного решения задачи необходимо:

- исследовать предметную область, а также особенности реализации целевой платформы
- сформулировать методику поиска и анализа процессов в расширенном образе памяти
- реализовать извлечение адресного пространства процессов виртуальных машин из расширенного образа памяти
- реализовать извлечение образа физической памяти гостевой машины из восстановленного адресного пространства процесса
- проверить корректность восстановленной памяти

В качестве хостовой системы будет рассмотрена Windows 7, как одна из самых распространенных систем на данное время. В качестве архитектуры выбрана x86-64, так как большинство современных процессоров поддерживает эту систему команд.

В качестве виртуальных машин для анализа были выбраны QEMU и VirtualBox. Эти проекты имеют открытый исходный код, что позволяет провести анализ внутреннего устройства механизма управления памятью.

Обзор

Методы захвата памяти

В этом разделе будут рассмотрены средства для захвата образов памяти на системе под управлением операционных систем семейства Windows. Для выполнения этой задачи могут использоваться как аппаратные, так и программные средства [5].

Аппаратные методы

А. Использование шины FireWire

Последовательная шина FireWire (IEEE 1394) была разработана компанией Apple в 1986 году для высокоскоростной передачи данных. Сейчас FireWire присутствует на большом количестве настольных систем и используется для подключения таких устройств, как камеры, сканеры, научное оборудование и т.п.: В 2004 году немецкие исследователи Maximilian Dornseif, Michael Becher и Christian Klein успешно использовали шину FireWire для захвата образа памяти на работающей системе [1]. Было установлено, что устройства, подключенные через FireWire, могут делать запросы на чтение и запись оперативной памяти, используя режим Direct Memory Access (DMA), в котором обращение к памяти происходит без участия центрального процессора. Таким образом, появляется возможность подключить свой компьютер к целевой системе по шине FireWire и считать содержимое памяти без какой-либо ее модификации [13].

При использовании данного подхода для захвата образа памяти операционной системы Windows необходимо также обойти защиту, основанную на том, что доступ к DMA могут получить только те устройства, у которых особым образом заполнен Config Status Register, (CSR — регистр, отвечающий за инициализацию устройства в системе). Данная защита легко обходится простым копированием CSR с устройств, которые предназначены для работы по шине FireWire (например, плееры iPod).

Б. Использование карт расширения

Кроме FireWire можно также использовать специальные устройства, подключающиеся к интерфейсам PCI, PCI Express и т.п. Одним из таких устройств является Tribble [3], представляющий собой PCI-карту, которую необходимо заранее установить на целевую систему. После регистрации в системе в момент включения или перезагрузки устройство переходит в режим ожидания, не отвечая на запросы по шине. В необходимый момент устройство активируется (с помощью переключателя на плате, а при доработке — и с помощью таймера или дистанционного управления) и начинает считывать память на

внешнее запоминающее устройство, используя аналогичный подход с доступом к памяти через DMA.

Недостатком данного подхода является необходимость скрытно установить Tribble на компьютере, с которого в ближайшем будущем надо будет снимать образ памяти. Скрытная установка может быть затруднительна, так как связана с физическим доступом к компьютеру. Одной из разумных областей применения для данного устройства может быть установка на сервера или системы для быстрого снятия памяти в случае подозрения на наличие вредоносного исполняемого кода в памяти.

Коммерческое распространение также получило устройство WindowsSCOPE CaptureGUARD [20], которое выпускается в вариантах для шин PCI Express и ExpressCard.

B. Атака «холодной загрузки»

Так называемая атака «холодной загрузки» (cold boot attack) основывается на том факте, что после отключения питания содержимое памяти не сразу стирается, а остается на какое-то время (порядка несколько секунд). Атака заключается в том, что питание целевой системы резко отключается, после чего планки памяти быстро переставляются в другое заранее подготовленное устройство, которое считывает содержимое памяти. Например, можно модифицировать BIOS на этом устройстве таким образом, чтобы память никаким образом не очищалась и не проверялась в результате POST (power-on self test), тем самым избегая ее перезаписи.

Исследование [9] также продемонстрировало, что можно существенно уменьшить интенсивность утечки заряда из микросхем памяти путем их замораживания. Так, при охлаждении сжатым воздухом до -50°C меньше 1% содержимого памяти было потеряно через 10 минут после отключения питания.

Программные методы

Для операционных систем семейства Windows есть большое количество утилит, которые осуществляют захват памяти. Главный недостаток программных средств для захвата — это модификация памяти в том или ином виде во время исполнения. Данные утилиты должны запускаться на работающей системе, а поэтому взаимодействие с самой системой неизбежно, что приводит к изменению физической памяти.

Поэтому обязательными требованиями для таких программ являются небольшой размер, возможность запуска без установки, а также статическое связывание для предотвращения загрузки дополнительных библиотек.

К программным средствам для захвата памяти относятся DumpIt, WinPmem, FastDump, Belkasoft Live RAM Capturer.

Методы анализа памяти

До середины 2000-х анализ памяти в основном состоял в использовании утилит strings и grep [2]. Летом 2005-го года во время конференции Digital Forensic Research Workshop (DFRWS) был устроен конкурс «Memory Analysis Challenge», в ходе которого участникам были предложены два образа памяти с компьютеров под управлением Windows XP. Результатом конкурса стало несколько работ и утилит, посвященных основным подходам к поиску и анализу разнообразных структур в образе памяти.

Первыми значительными работами в этой области также являются [11], [14], [18].

PTFinder

Автор работы [18], Andreas Schuster, во время своих исследований разработал утилиту под названием PTFinder (Process and Thread Finder, последняя версия вышла в ноябре 2007-го года). Основными возможностями программы являются поиск в образе памяти информации о запущенных и недавно завершенных процессах и потоках. PTFinder, написанный на языке Perl, представляет собой реализацию так называемого «сигнатурного» метода поиска структур ядра в памяти.

Volatility Framework

На конференции Black Hat в марте 2007 года Aaron Walters и Nick Petroni представили набор утилит, написанных на Python, под названием volatools, предназначенных для разностороннего анализа памяти. После конференции volatools были дополнены и перевыпущены под названием Volatility Framework [19]. Сейчас Volatility поддерживается компанией Volatile Systems и является одним из самых функциональных пакетов для исследования памяти. В его возможности входит извлечение информации о запущенных процессах, открытых сетевых соединениях, открытых файлах, открытых записей реестра, извлечение образов процессов, динамических библиотек и пр. На данный момент предоставляется поддержка образов памяти Windows (от Windows XP до Windows 7, 32 и 64 бита). Активно развивается анализ образов Linux, а также имеется гибкая система плагинов.

Технологии виртуализации

Виртуализация — технология, которая позволяет эмулировать аппаратное обеспечение разных платформ (целевых или гостевых) на базовой платформе («хост-платформе»), а также дает возможность соответствующим образом распределять и изолировать ресурсы между несколькими запущенными экземплярами гостевых систем.

Виртуализация может быть как полностью программной, так и аппаратной. Во втором случае используются дополнительные аппаратные средства (например, специальные расширения системы команд процессора, такие как Intel VT и AMD-V).

В данной работе будут рассмотрены средства программной виртуализации как более простые в использовании и имеющие в данный момент больший интерес с точки зрения криминалистического анализа.

Центральной частью любого средства виртуализации является гипервизор (монитор виртуальных машин). Гипервизор — это программа или аппаратная схема, которая обеспечивает одновременное и параллельное выполнение нескольких систем на одном физическом оборудовании, а также изоляцию этих систем друг от друга.

В следующем разделе будет описана классификация гипервизоров.

Типы гипервизоров

Выделяют два типа гипервизоров:

1. Гипервизор первого типа (native hypervisor, автономный гипервизор) исполняется прямо на физическом оборудовании и служит необходимой прослойкой между аппаратными средствами и гостевыми системами. Представляет собой минималистическую операционную систему (представленных в виде микроядра или наноядра), которая имеет прямой доступ к физическому оборудованию и отвечает за выполнение инструкций гостевых систем и распределение ресурсов между ними.

К гипервизорам первого типа относят Xen, Kernel Virtual Machine, Microsoft Hyper-V

2. Гипервизор второго типа (hosted hypervisor) может исполняться только в контексте другой системы, например, как процесс внутри операционной системы. Непривилегированные инструкции гостевой операционной системы могут выполняться прямо на физическом процессоре, но, например, инструкции, получающие доступ к устройствам ввода-вывода, а также привилегированные инструкции процессора преобразовываются процессом виртуальной машины специальным образом, тем самым создавая виртуализированную аппаратную среду.

Гипервизорами второго типа являются VMware Workstation, Windows Virtual PC, а также рассмотренные далее QEMU и VirtualBox.

Средства виртуализации

В данном разделе описаны средства виртуализации, чья физическая память в дальнейшем будет восстанавливаться.

QEMU ("Quick EMUlator") [16] — свободная программа с открытым исходным кодом для эмуляции аппаратного обеспечения различных платформ. Относится ко второму типу гипервизоров. Может осуществлять эмуляцию в двух режимах: эмуляция компьютера (computer emulation), когда QEMU производит эмуляцию всей аппаратной составляющей, а также эмуляция пользовательского режима (user-mode emulation), в котором QEMU

исполняет программы, скомпилированные для другой архитектуры процессора. Поддерживаются архитектуры x86, x86_64, MIPS, SPARC, PowerPC, ARM и т.д.

QEMU играет значительную роль и в составе других средств виртуализации. Например, VirtualBox использует некоторые виртуальные устройства от QEMU, а также имеет встроенный динамический рекомпилятор, основанный на QEMU. Эмуляция оборудования в гипервизоре Xen также основана на разработках проекта QEMU. Еще одной системой виртуализации, использующей модули QEMU, является KVM.

Преимуществом QEMU является отсутствие требования прав администратора для запуска виртуальных машин, в отличие от других решений виртуализации.

VirtualBox (Oracle VM VirtualBox) [15] — система виртуализации для архитектуры x86, является свободным программным обеспечением, начиная с версии 4. Также принадлежит ко второму типу по классификации гипервизоров. Поддерживаются расширения процессора для аппаратной виртуализации (Intel VT и AMD-V), что позволяет на 32-битных хостовых системах запускать 64-битные гостевые системы.

VirtualBox позволяет быстро создавать и эффективно управлять виртуальными машинами, поддерживаются функции создания снимков (snapshots), организация общих папок, «проброс» портов и т.п. Всё это делает VirtualBox отличным решением для создания чистой среды для разработки и экспериментов, от которой можно быстро избавиться, не оставляя следов на хостовой системе. Таким образом, VirtualBox является привлекательным средством для злоумышленников.

Необходимые сведения о подсистеме памяти Windows

Ниже приведен краткий обзор подсистемы виртуальной памяти Windows на платформе x86-64. Эти сведения будут необходимы для полного восстановления адресного пространства процесса.

Обзор механизма виртуальной памяти в архитектуре x86-64

Виртуальное адресное пространство процесса в архитектуре x86-64 теоретически может составлять 16 экзабайт (2^64 байт), но современные реализации данной архитектуры используют младшие 48 бит (из общих 64) для адресуемой виртуальной памяти, что позволяет адресовать 256 терабайт (2^48 байт). Также в современных реализациях существует требование, чтобы 16 старших неиспользуемых бит адреса (биты 48-63) были равны биту с номером 47. Адреса, которые удовлетворяют данному свойству, называются «адресами в канонической форме», или «каноническими адресами». Таким образом, каноническими являются адреса 0x00000000'00000000 — 0x00007fff'ffffffff, а также

0xffff8000'00000000 — 0xffffffff'ffffffff. Если совершается попытка обращения по неканоническому адресу, процессор вызывает исключение [10].

Исходя из этого, виртуальное адресное пространство процесса разбивается на две части: младшую (биты 48-64 установлены в 0) и старшую (биты 48-64 установлены в 1). 64-битная версия ОС Windows использует старшую часть адресного пространства как пространство ядра, общее для всех процессов, а младшую часть отдаёт в распоряжение пользовательскому пространству процесса, где может храниться исполняемый код, стеки потоков, куча и т.п.
Виртуальное пространство разделено на страницы, которые могут быть малыми (small pages), размером по четыре килобайта, и большими (large pages), по два мегабайта каждая.

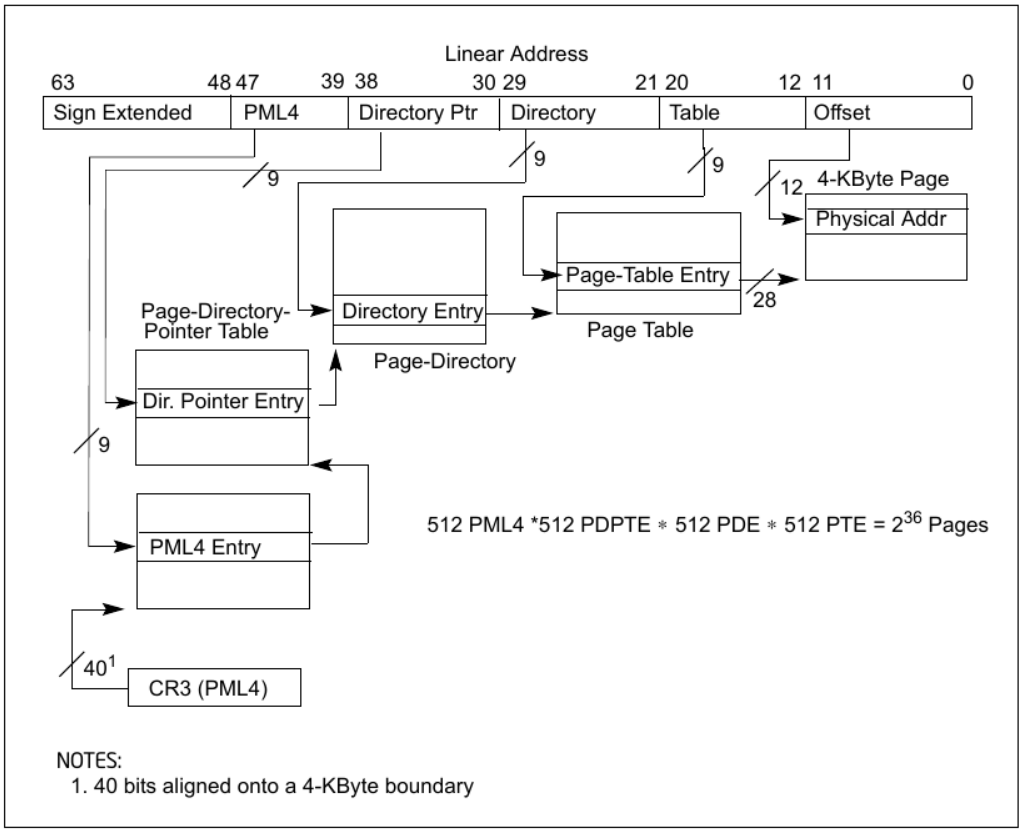

Рис. 1 Преобразование виртуального адреса на платформе x86-64 (из [10])

Важным механизмом подсистемы памяти является преобразование (или «трансляция») виртуального адреса в физический (Рис. 1). Трансляция происходит следующим образом: виртуальный адрес разделяется на пять частей. Каждая из этих частей дает возможность получить записи различных системных таблиц преобразования адресов:

1. Биты 47-39 описывают смещение в таблице PML4, на которую указывает регистр процессора CR3. Из полученной записи PML4 выделяется физический адрес таблицы PDPT (page directory pointer table).

2. Биты 38-30 описывают смещение в таблице PDPT. По этому смещению находится запись PDPE (page directory pointer entry), из которой выделяется физический адрес таблицы PD (page directory).

3. Биты 29-21 описывают смещение в таблице PD, по которому находится запись PDE (page directory entry). Из этой записи можно найти физический адрес таблицы страниц (page table). Если в PDE бит, отвечающий за тип страницы, установлен в 1, то данная PDE указывает на большую страницу, и оставшиеся биты (20-0) являются смещением внутри страницы.

4. Биты 20-12 описывают смещение в таблице страниц, по этому смещению находится запись таблицы страниц (PTE, page table entry). Из PTE выделяется физический адрес соответствующей страницы.

5. По битам 11-0 происходит адресация внутри полученной страницы.

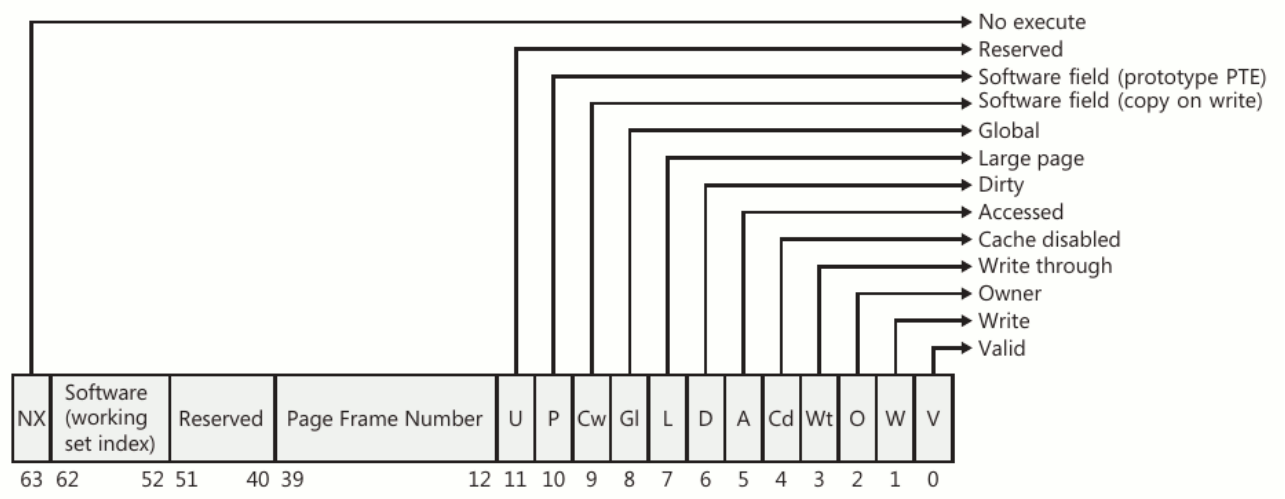

Рис. 2 Формат PTE (из [17])

Виды записей в таблицах преобразования

Все записи в таблицах преобразования (таких, как PML4, PDPT, PD, PT) представляют собой 8-байтные значения. Биты с номерами 12-39 образуют значения PFN (page frame number), которые указывают на соответствующую таблицу более низкого уровня [17]. Чтобы получить конкретный физический адрес из PFN, необходимо произвести битовый сдвиг PFN на 12 бит влево. Например, в записи PDE директории страниц значение PFN, сдвинутое на 12 влево является физическим адресом соответствующей таблицы страниц.

Оставшиеся биты — это флаги, в зависимости от которых страницы могут иметь различные состояния (Рис. 2).

За конкретное местоположение страницы отвечают записи PTE, поэтому необходимо рассмотреть разнообразные виды PTE. Если V (бит номер 0) равен 1 в PTE, то соответствующие этому PTE страницы являются действительными (valid), то есть они

присутствуют в оперативной памяти. Если PTE является недействительным (V=0), то необходима классификация таких PTE.

Типы недействительных PTE

Zero PTE

Весь PTE равен нулю. Про страницу почти что ничего не известно: можно рассмотреть соответствующую запись в VAD-дереве (Virtual Address Descriptor tree), чтобы определить статус страницы (к примеру, выделена ли она), но даже в этом случае более содержательной информации получить не удается.

Страницы из файла подкачки (Pagefile PTE)

У таких PTE V=U=P=0 (биты 0, 10, 11 равны нулю). В таких случаях PFN является смещением искомой страницы в файле подкачки, а биты 1-4 составляют номер файла подкачки (Windows позволяет создавать до 16 файлов подкачки).

Demand Zero PTE

Биты 10-39 и 0-4 равны нулю. Страницы такого типа являются своеобразными «заглушками»: при обращении по этому PTE ОС будет выдавать страницу, полностью состоящую из нулей.

Transition PTE

U=1, P=V=0. Такие страницы присутствуют в оперативной памяти, но находятся в «переходном» состоянии, то есть были изменены, но еще не записаны на диск.

Prototype PTE

P=1, V=0. В этом случае данный PTE является ссылкой на прототипный PTE. Прототипные PTE используются для реализации механизма разделяемой памяти и спроецированных в память файлов. С их помощью одну страницу могут использовать несколько процессов (например, в случае разделяемой библиотеки). Чтобы не обновлять статус страницы в каждом адресном пространстве процесса, который использует эту страницу, сделано так, что PTE разных процессов указывают на один прототипный PTE. Таким образом, при перемещении/удалении страницы нужно обновлять только один прототипный PTE.

Архитектура решения

Общую архитектуру решения можно представить в виде следующей схемы:

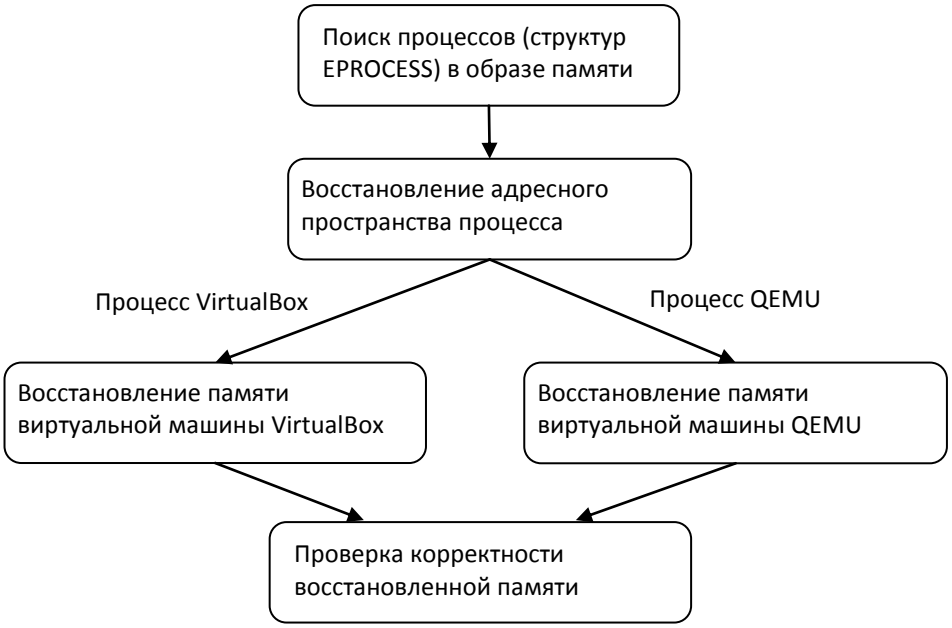

Рис. 3 Общая архитектура решения

В следующих разделах подробнее рассмотрен каждый из шагов, изображенных на схеме. Предполагается, что в распоряжении имеется образ памяти, снятый с целевой системы, файл подкачки, а также образ жесткого диска целевой системы.

Поиск процессов

Для успешного анализа образа памяти необходимо уметь находить структуры EPROCESS. В семействе ОС Windows NT данная структура описывает процесс для ядра ОС и находится в адресном пространстве ядра. Существует несколько основных методов, используемых для поиска в образе памяти структур EPROCESS ([4], [18], [21]). Предполагается, что случаи руткитов[3], намеренно изменяющих сигнатуры структур ядра [4], рассматриваться не будут, поэтому можно использовать любой из методов, учитывая версию и архитектуру целевой системы.

Восстановление адресного пространства процесса

Восстановление адресного пространства процесса состоит из нескольких этапов:

1. Необходимо определить, адресное пространство какого процесса требуется получить. Определение осуществляется путём выбора нужной структуры EPROCESS, полученной с помощью одного из методов сканирования.
2. Получение физического адреса (смещения от начала файла образа) таблицы PML4.

[3] Руткит – программа, основной задачей которой является скрытие следов присутствия в системе себя или другой программы.

3. Перебор всех PML4-записей. Если PML4-запись является действительной, то из нее можно получить физический адрес соответствующей таблицы PDPT. Про недействительные записи будет сказано позже.
4. Получив физический адрес таблицы страниц в образе, можно аналогичным образом перебрать все PDPT-записи, а затем и записи PDE, PTE.
5. Если запись PTE действительная, то из нее можно извлечь физический адрес соответствующей страницы, после чего остаётся скопировать найденную страницу в «выходной» файл.

Таким образом, осуществляется обход в глубину таблиц преобразования PML4, PDPT, PD и PT.

Главной трудностью, с которой можно столкнуться при таком подходе, являются недействительные страницы. Если их игнорировать и восстанавливать только действительные страницы, то есть помеченные системой как присутствующие в памяти в момент ее захвата, то в некоторых случаях можно упустить более 95% адресного пространства (несбрасываемыми на диск являются таблицы страниц и некоторые структуры ядра, максимум — несколько десятков мегабайт).

Недействительные записи и соответствующие им страницы и таблицы бывают нескольких типов, и каждый тип требует своего подхода в обработке. Далее рассмотрена обработка таких записей.

Обработка недействительных записей

Классифицировать обработку недействительных PTE-записей можно следующим образом [11]:

Zero

PTE этого типа приходится игнорировать ввиду отсутствия какой-либо содержательной информации о состоянии страницы.

Страницы из файла подкачки (Pagefile PTE)

PTE, ссылающиеся на находящиеся в файле подкачки страницы, содержат номер файла подкачки (по этому номеру можно получить путь к самому файлу, данное соответствие можно найти в реестре) и смещение страницы внутри этого файла.

Demand Zero PTE

При запросах к странице с PTE такого типа операционная система должна возвращать страницу, полностью состоящую из нулей. При обработке образа памяти можно делать то же самое: считать, что страница данного PTE заполнена одними нулями.

Transition

Состояние страниц с PTE этого типа можно считать актуальным, недействительными они считаются только в смысле несогласованности состояния в памяти и на диске. То есть при обработке такие страницы предлагается обрабатывать как действительные.

Prototype

PTE данного типа содержит адрес прототипного PTE. Адрес получается следующим образом:

addr = (PTE >> 16) + 0xffff000000000000

Страница, на которую ссылается полученный прототипный PTE, может находиться в одном из шести состояний, напоминающих уже описанные PTE-состояния, за одним главным исключением: установленный бит P указывает на то, что данный прототипный PTE описывает страницу в файле, спроецированном в память (т.е. запрещены ссылки одних прототипных PTE на другие). PTE данного типа подробнее рассмотрены в следующем разделе.

Если недействительной является PML4-запись, PDPT-запись или PDE, то она может относиться к типу Pagefile или Transition, их обработка аналогична обработке PTE-записей.

Обработка прототипных PTE

Обрабатывать прототипные PTE предлагается следующим образом:

- Active (V=1)

Прототипный PTE указывают на действительную страницу в памяти. Аналогично обычным действительным PTE, адрес страницы получается по значению PFN.

- Transition (V=0, U=1, D=0)

Аналогично непрототипным Transition-PTE.

- Modified No-Write (V=0, U=D=1)

PTE данного типа указывают на страницы, которые находятся в списке модифицированных, но запрещенных на запись страниц. Обрабатываются аналогично Transition-случаю, смещение такой страницы в файле образа получается по её PFN.

- Pagefile

Делается то же самое, что и в случае непрототипных Pagefile PTE.

- Demand Zero

Аналогично непрототипному случаю.

- Mapped File (P=1)

Данный прототипный PTE описывает страницу, которая находится в файле, проецируемом на память. Нужно определить, к какому файлу относится данная страница, а также узнать смещение внутри этого файла, по которому страница располагается.

Организуется следующий порядок действий:

1. Определяется узел VAD-дерева, к диапазону которого принадлежит адрес изучаемой страницы.
2. По узлу и содержанию прототипной PTE-записи определяется нужная структура SUBSECTION, отвечающая за проецирование частей файла на определённые виртуальные адреса.
3. В структуре SUBSECTION есть ссылка на структуру CONTROL_AREA, которая описывает проецирование в память определённого файла. Также из SUBSECTION можно узнать смещение (оно выражено в секторах — блоках данных размером по 512 байт) для страницы, на которую ссылается рассматриваемый прототипный PTE.
4. В CONTROL_AREA имеется ссылка на структуру типа FILE_OBJECT, откуда можно узнать требуемый путь к файлу.

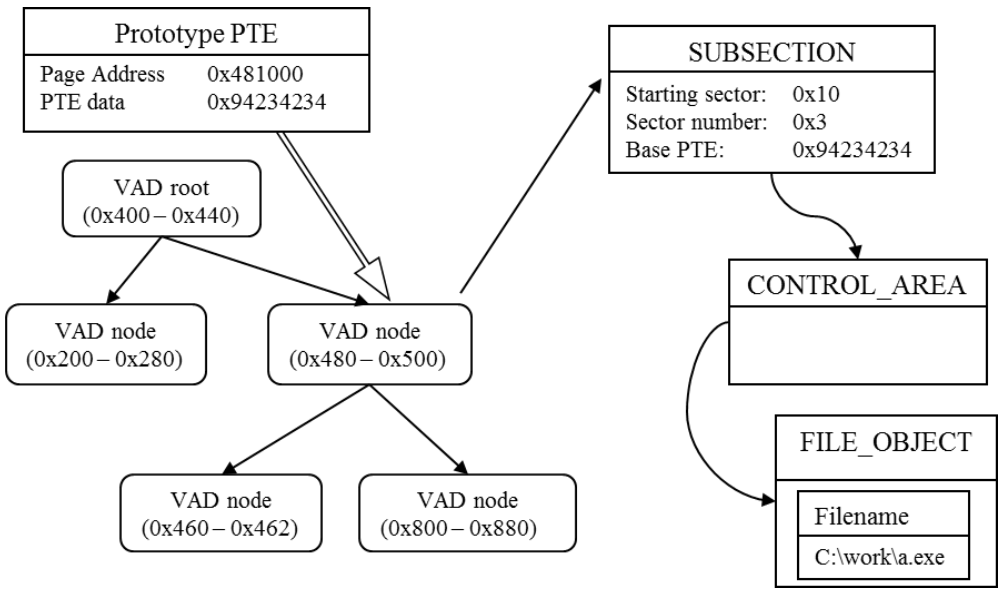

Рис. 4 Нахождение пути к файлу по прототипному PTE

Используя описанный алгоритм (Рис. 4), можно получить пути к проецируемым файлам, а также смещения страниц внутри файлов. Таким образом, если скопировать эти файлы с исследуемого компьютера, данные из них будут использованы в ходе восстановления адресного пространства.

Подводя итог, общую классификацию типов PTE-записей и действий с ними можно изобразить следующим образом:

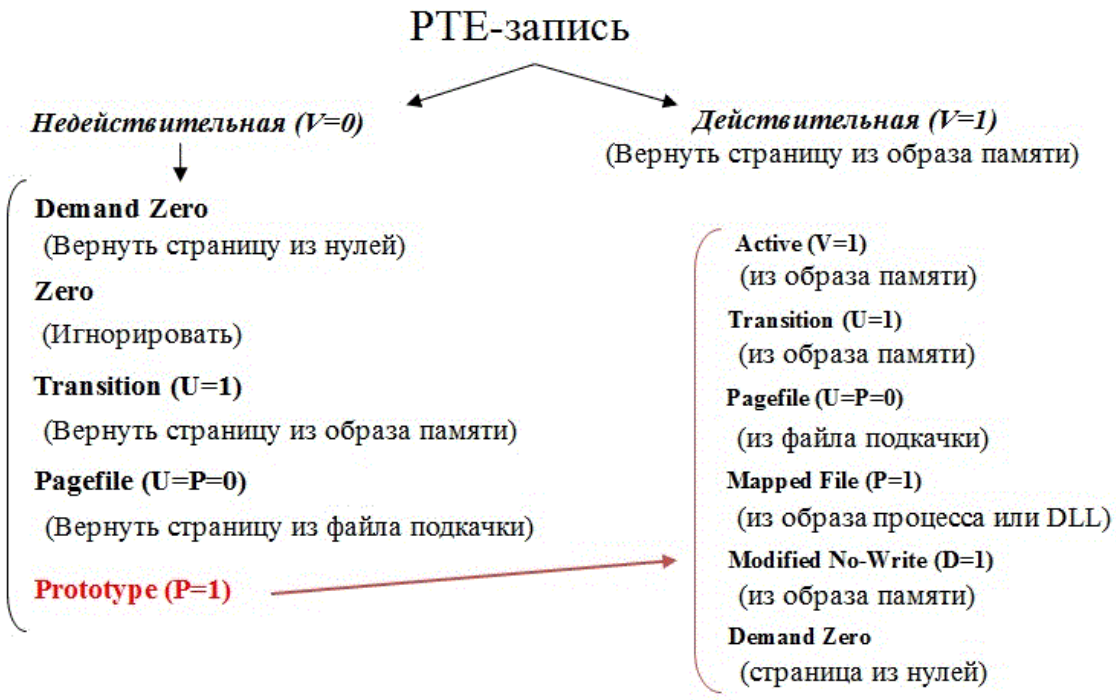

Рис. 5 Источники страниц для различных типов PTE

Обработав каждый тип PTE соответствующим образом, предоставляется возможность узнать местоположение страниц адресного пространства, после чего скопировать их в выходной файл.

Восстановление физической памяти QEMU

Различные фрагменты адресуемой памяти в QEMU хранится в виде двусвязного списка ram_list, описываемого структурой RAMList (Рис. 6). Каждый элемент в списке представляет собой структуру RAMBlock, описывающую конкретный фрагмент памяти: это может быть и физическая память, и видеопамять, и системный BIOS.

```
typedef struct RAMList {
    QemuMutex mutex;
    uint8_t *phys_dirty;
    RAMBlock *mru_block;
    struct {
        struct RAMBlock *tqh_first;
        struct RAMBlock * *tqh_last;
    } blocks;
    uint32_t version;
} RAMList;
```

Рис. 6 Структура RAMList

```
typedef struct RAMBlock {
    struct MemoryRegion *mr;
    uint8_t *host;
    ram_addr_t offset;
    ram_addr_t length;
```

```
    uint32_t flags;
    char idstr[256];
    struct {
        struct RAMBlock *tqe_next;
        struct RAMBlock * *tqe_prev;
    } next;
} RAMBlock;
```

Рис. 7 Структура RAMBlock

В структуре RamBlock (Рис. 7) содержатся:

1. mr — ссылка на структуру типа MemoryRegion. В данный момент QEMU переходит на новый API управления памяти, основанный на описании регионов памяти с помощью структур типа MemoryRegion, а не RAMBlock. Наличие поля mr является временной мерой, принятой для облегчения перехода на новый API.
2. host — виртуальный адрес описываемого блока памяти в адресном пространстве процесса QEMU.
3. offset — смещение блока памяти в адресном пространстве гостевой машины. Для физической памяти равен 0.
4. length — длина блока памяти.
5. flags — флаги, описывающие состояние блока.
6. idstr — ссылка на ASCII-строку, которая является человекочитаемым идентификатором данного блока.
7. next — структура, описывающая ссылки на следующий и предыдущий элементы списка ram_list.

Пример. Инициализированные блоки RAMBlock в составе ram_list могут выглядеть следующим образом:

```
block offset: 0x00000000, length: 0x08000000, host: 0x0x7fffdf600000, idstr: ‘pc.ram’
block offset: 0x08000000, length: 0x00020000, host: 0x0x7ffff7fd8000, idstr: ‘system.flash’
block offset: 0x08020000, length: 0x00020000, host: 0x0x7ffff7e95000, idstr: ‘isa-bios’
block offset: 0x08040000, length: 0x00020000, host: 0x0x7ffff7e73000, idstr: ‘pc.rom’
block offset: 0x08060000, length: 0x00800000, host: 0x0x7fffde400000, idstr: ‘vga.vram’
block offset: 0x08860000, length: 0x00010000, host: 0x0x5555564e4000, idstr: ‘0000:00:02.0/cirrus_vga.rom’
```

Рис. 8 Инициализированные блоки RAMBlock

Требуется найти структуру RAMBlock, которая отвечает за отображение физической памяти.

После исследования исходного кода было установлено, что у блока, описывающего физическую память гостевой машины идентификатор равен «pc.ram». Поэтому логичным шагом является поиск в образе памяти процесса данной строки. После нахождения смещения идентификатора, можно получить смещение предполагаемой структуры типа RAMBlock.

Получив очередное смещение, требуется проверка (валидация) того, что по данному смещению находится действительная структура RAMBlock.

Можно выделить несколько правил для проверки:

1. Поле mr размером должно являться действительным ненулевым указателем в адресном пространстве пользователя процесса. То есть для Windows на платформе x64 должно выполняться соотношение 0 < mr < 0x8000'00000000 при общем размере поля в 8 байт. В дальнейшем к значению 0x8000'00000000 как к верхней границе адресного пространства пользователя будем обращаться как к MAX_USERSPACE_x64.
2. Для поля host должно выполняться аналогичное соотношение: 0 < host < MAX_USERSPACE_x64
3. Поле offset должно быть равно 0 для физической памяти.
4. Как уже было сказано, поле, описываемое массивом символов idstr должно содержать идентификатор «pc.ram». Еще одной особенностью данного поля является то, что кроме первых шести байтов, представляющих собой строку «pc.ram», оставшиеся 250 байт являются нуль-байтами. Это связано с тем, что память для блока RAMBlock выделяется с помощью функции mmap с параметром MAP_ANON, который приводит к принудительной инициализации нулями выделяемой памяти.

Алгоритм восстановления физической памяти гостевой машины под управлением QEMU в общем виде:

1. Восстановление адресного пространства процесса QEMU
2. Поиск идентификатора «pc.ram» в восстановленном адресном пространстве
3. Нахождение смещения предполагаемой структуры RAMBlock
4. Проверка найденной структуры
5. Сохранение смещения структуры RAMBlock в случае успешной проверки

Для каждого полученного смещения корректной структуры RAMBlock:

6. Получение значений полей host и length.
7. Чтение блока длины length, начиная с виртуального адреса host в адресном пространстве процесса QEMU.

В результате работы алгоритма получается файл, который является образом физической памяти гостевой машины.

Восстановление физической памяти VirtualBox

В VirtualBox структура VM (описание содержится в файле include/VBox/vmm/vm.h дерева исходных кодов VirtualBox) содержит всю информацию, относящуюся к конкретной

виртуальной машине, в том числе и описание отображений памяти. Данная структура имеет большое количество полей и фиксированный размер в 128 Кб вне зависимости от архитектуры.

```
typedef struct VM {
   <...>
   /** Size of the VM structure including the VMCPU array. */
   uint32_t cbSelf;
   /** Offset to the VMCPU array starting from beginning of this structure. */
   uint32_t offVMCPU;
   <...>
   /** PGM part. */
   union {
      struct PGM  s;
      uint8_t     padding[4096*2+6080];
   } pgm;
   <...>
} VM;
```

Рис. 9 Структура VM

Для поиска структуры VM (Рис. 9) предлагается использовать следующие наблюдения:

1. Поле cbSelf описывает размер структуры VM, который составляет 131072 байт (128 Кбайт) и меняется только в случае серьёзных изменений в ядре VirtualBox, а это происходит редко: значительных модификаций этой структуры не происходило начиная с VirtualBox версии 4.0.0, вышедшей в декабре 2010 года.

2. Поле offVMCPU описывает смещение массива VMCPU от начала структуры VM, что составляет 110592 байт (108 Кбайт). Данное значение также меняется редко, по описанным в пункте 1 причинам.

Таким образом, можно использовать сигнатуру 0x00000200'00b00100 для первичного поиска структуры VM: значение 0x00000200 представляет собой поле cbSelf в little-endian, 0x00b00100 — поле offVMCPU.

После получения смещения предполагаемой структуры VM необходимо проверить её на корректность. Так как в структуре VM находится большое количество указателей на другие структуры, можно проверять, являются ли каждый из них (допустим, указатель Ptr) ненулевым указателем в адресном пространстве пользователя, т.е. $0 < Ptr < MAX\_USERSPACE\_x64$. Например, таким образом можно проверить поля pSession, pUVM, pVMR3, pVMR0, pVMRC, которые находятся в начале структуры VM.

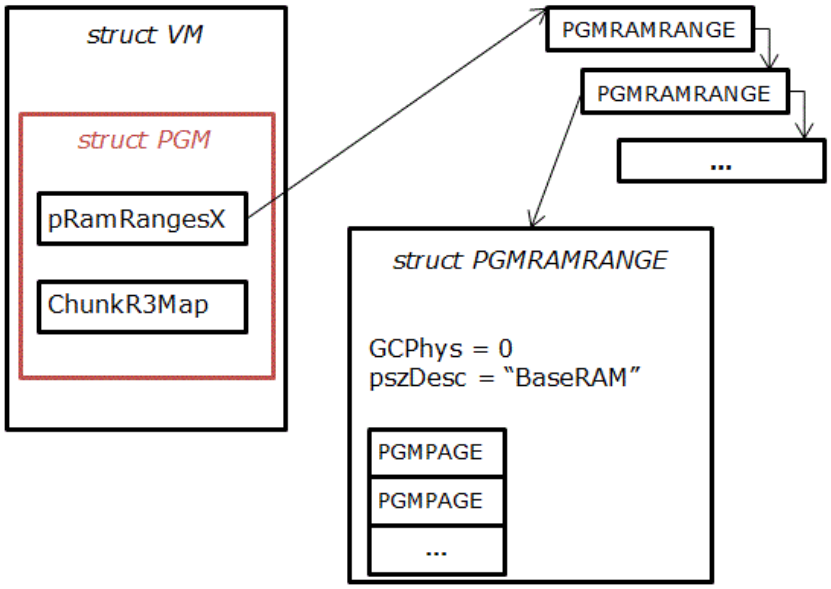

Рис. 10 Структуры VM, PGM и PGMRAMRANGE

Важную роль в механизме управления памятью VirtualBox играет структура PGM (Page Monitor), которая является подструктурой структуры VM.

PGM содержит указатель pRamRangesXR3 на список из структур PGMRAMRANGE (Рис. 10), каждая из которых описывает область памяти, аналогично RAMBlock для QEMU: один PGMRAMRANGE описывает физическую память, другой — видеопамять и т.д.

```
typedef struct PGMRAMRANGE {
   /** Start of the range. Page aligned. */
   RTGCPHYS GCPhys;
   /** Size of the range. (Page aligned of course). */
   RTGCPHYS cb;
  <...>
   /** The range description. */
   R3PTRTYPE(const char *) pszDesc;
   <...>
}
```

Рис. 11 Структура PGMRAMRANGE

Требуется найти структуру PGMRAMRANGE (Рис. 11), которая описывает физическую память гостевой машины. Отличительной особенностью искомой структуры является значение GCPhys (начало региона памяти), равное 0, а также строка pszDesc, равная "Base RAM".

Полученная структура PGMRAMRANGE содержит массив структур PGMPAGE (Рис. 10), каждая из которых является своеобразным дескриптором страницы гостевой системы. Структура PGMPAGE имеет размер 16 байт и содержат битовые поля, описывающие тип и статус страницы: является ли страница выделенной, нулевой, разделяемой между несколькими виртуальными машинами и т.п. Также у каждой страницы есть идентификатор — idPage.

```
typedef struct PGMCHUNKR3MAP
{
   /** The key is the chunk id. */
   AVLU32NODECORE Core;
   <...>
   /** The mapping address. */
   void *pv;
} PGMCHUNKR3MAP;
```

Рис. 12 Структура PGMCHUNKR3MAP

Но, к сожалению, в структуре PGMPAGE не содержится виртуальный адрес страницы в пространстве процесса VirtualBox. Для того чтобы определить его, необходимо обратиться к структуре ChunkR3Map, являющейся подструктурой PGM. ChunkR3Map содержит ссылку на AVL-дерево pTree из структур типа PGMCHUNKR3MAP (Рис. 12). Ключами дерева являются chunkId (которые получаются из idPage битовым сдвигом вправо на 9), а значениями — виртуальные адреса pv, которые являются базовыми адресами для групп страниц с одинаковым chunkId.

Итоговый виртуальный адрес для страницы, описываемой структурой PGMPAGE, получается следующим образом (предположим, что имеется значение pgmpage типа PGMPAGE):

1. Вычисляются idPage и chunkId:

```
uint32_t idPage = pgmpage.idPage
uint32_t chunkId = idPage >> 9
```

2. В дереве pTree ищется узел со значением ключа chunkId:

```
PGMCHUNKR3MAP pageNode = pTree.find(chunkId)
```

3. Вычисляется базовый адрес nodePv

```
void *nodePv = pageNode.pv
```

4. Вычисляется итоговое значение pageVaddr:

```
void *pageVaddr = nodePv + ((idPage & 0x1ff) << 12)
```

Назовем данный алгоритм из четырех шагов «алгоритмом нахождения виртуального адреса страницы из структуры PGMPAGE».

Алгоритм восстановления физической памяти из гостевой машины под управлением VirtualBox в общем виде можно описать следующим образом:

1. Восстановление адресного пространства процесса VirtualBox
2. Поиск сигнатуры 0x00000200'00b00100 в восстановленном адресном пространстве
3. Нахождение смещения предполагаемой структуры VM
4. Проверка корректности структуры
5. В случае валидной структуры — сохранение смещения структуры VM

Для каждого полученного смещения корректной структуры VM:

6. Получение смещения поля pgm, pgm.pRamRangesXR3, pgm.ChunkR3Map.pTree

7. Нахождение структуры PGMRAMRANGE, описывающей физическую память гостевой машины

Для каждой структуры PGMPAGE в составе найденной PGMRAMRANGE:

8. Применение алгоритма нахождения виртуального адреса страницы из структуры PGMPAGE
9. Копирование страницы с полученного виртуального адреса пространства процесса VirtualBox в выходной файл.

В результате работы алгоритма получается файл, представляющий собой образ физической памяти гостевой машины.

Проверка корректности восстановленной памяти

После восстановления физической памяти гостевой машины необходимо проверить её корректность, то есть показать, что она максимально соответствует реальной физической памяти гостевой системы.

В качестве проверки предлагается использовать Volatility Framework, запуск которого сможет показать, выглядит ли полученный файл как образ памяти и сколько полезной информации можно будет из него извлечь.

Также предлагается использовать разработанные в рамках данной работы инструменты для восстановления адресного пространства процессов из полученного образа памяти гостевой машины.

Особенности реализации

В рамках данной работы предполагается, что используемые образы памяти, файлы подкачки, а также образы жесткого диска являются корректными, то есть снятыми с помощью аппаратных методов в одно и то же время для согласованности данных.

На языке Ruby был реализован набор программ, обладающих следующей функциональностью:

1. Сигнатурный поиск структур EPROCESS в образах памяти Windows 7 на платформах x86 и x86-64. Программа производит данный тип поиска и выводит все найденные процессы с указанием Process ID, названием процесса, виртуального и физического адреса EPROCESS, а также физического адреса директории страниц.
2. Восстановление адресного пространства для конкретного процесса (указание производится по PID). В ходе этого действия происходит:
   a. Построение VAD-дерева для обработки прототипных PTE
   b. Перебор записей преобразования (PTE, PDE, PXE, PPE), выполнение соответствующих действий для каждого типа записей

   В результате алгоритма восстановления создаётся два файла: само адресное пространство и специальный индексный файл, формат которого описан в следующем разделе.
3. Если на первом шаге был найден процесс средства виртуализации QEMU, программа восстанавливает физическую память гостевой системы, реализуя описанный в соответствующем разделе алгоритм.
4. Аналогично пункту 3 в случае найденного процесса VirtualBox.

Формат восстановленного адресного пространства

Во время работы алгоритма восстановления адресного пространства процесса страницы, получаемые из соответствующих источников (образ памяти, файл подкачки), копируются последовательно в выходной файл.

Также необходимо поддерживать специальную индексную информацию, позволяющую понять, какое смещение в получаемом файле соответствует данному виртуальному адресу. Данная информация необходима для дальнейших операций с адресным пространством, таких как, например, обращение по виртуальным адресам.

Одним из возможных подходов является запись дополнительного заголовка перед каждым непрерывным диапазоном памяти, в котором описан размер диапазона. Данный подход применяется в формате образов памяти LiME (Linux Memory Extractor) [12].

Несмотря на распространённость формата LiME, было решено записывать индексную информацию в отдельный файл. Отсутствие не относящихся к памяти процесса заголовков позволяет улучшить качество карвинга адресного пространства, что повышает шансы нахождения полезной информации.

Таким образом, индексный файл представляет собой массив структур, состоящих из двух частей: диапазона и смещения. Если использовать синтаксис языка C, описать данную структуру можно следующим образом:

```
struct INDEX_ENTRY {
   unsigned int startVa;       // начальный виртуальный адрес диапазона
   unsigned int endVa;         // конечный виртуальный адрес диапазона
   unsigned int physicalOffset; // смещение начала диапазона в выходном файле
};
```

Рис. 13 Запись в индексном файле

Тестирование восстановления адресного пространства

Для проверки корректности восстановления адресного пространства процесса была написана тестовая программа, которая выделяла несколько крупных блоков памяти и заполняла их специальным образом: страница, имеющая порядковый номер i внутри блока, заполнялась значением i % 256. Также в начале каждого блока записывалась сигнатура специального вида, чтобы проще было найти начало блока в восстановленном адресном пространстве. Размеры блоков подбирались специальным образом, чтобы значительная часть адресного пространства была вытеснена в файл подкачки.

Разработанной программе удалось восстановить 480 мегабайт из 493, которые были выделены операционной системой процессу тестовой программы. Была также написана другая программа, которая проверяла корректность данных, созданных тестовой. Доля корректных данных составила около 85%, остальные данные восстановить не удалось. Главная причина этого — не совсем корректная обработка записей PTE типа Transition. Было установлено, что записи данного типа при описанном способе обработки не всегда указывают на корректную страницу в образе памяти, даже при условии того, что требуемая страница в образе памяти содержится. Для полного понимания механизма обработки данных записей требуются дальнейшие исследования.

Тестирование восстановления памяти гостевой системы

Для тестирования успешности восстановления памяти гостевой системы (и в случае QEMU, и в случае VirtualBox) в первую очередь был использован Volatility Framework (далее — просто Volatility). Были использованы следующие команды Volatility:

- imageinfo — выводит информацию об образе памяти, проводя проверку на различные профили — набор правил и констант, описывающих образ памяти конкретной системы на конкретной платформе.
- pslist — ищет и выводит в образе памяти запущенные и недавно завершённые процессы с указанием идентификатора найденного процесса, идентификатора родительского процесса, названием, а также временем запуска и завершения.
- dlllist — для каждого процесса выводит пути к исполняемому образу процесса, а также к загруженным DLL-библиотекам, с указанием размера и базового виртуального адреса.
- netscan — выводит все активные сетевые подключения.

Команда imageinfo успешно идентифицировала восстановленную память гостевой системы с помощью профиля Win7SP1x86, который соответствует системе Windows 7 SP1 на платформе x86.

Команды pslist, dlllist, netscan отобразили правильную информацию о процессах и сетевых подключениях гостевой системы.

Также была использована разработанная в рамках данной работы программа для восстановления адресного пространства процесса: из образа памяти гостевой системы было извлечено виртуальное адресное пространство браузера Firefox, после чего в нём было успешно найдено и восстановлено изображение в формате JPEG размером 103 Кбайт, открытое в одной из вкладок браузера.

Внедрение

После реализации и успешного тестирования прототипа было решено интегрировать разработанное решение в продукт Belkasoft Evidence Center в рамках исследований, проводимых компанией Belkasoft и направленных на анализ образов памяти.

Была разработана архитектура, объединяющая написанные программы для разных архитектур. С самого начала во внимание принималась возможность добавления новых систем, архитектур и форматов представления образов памяти.

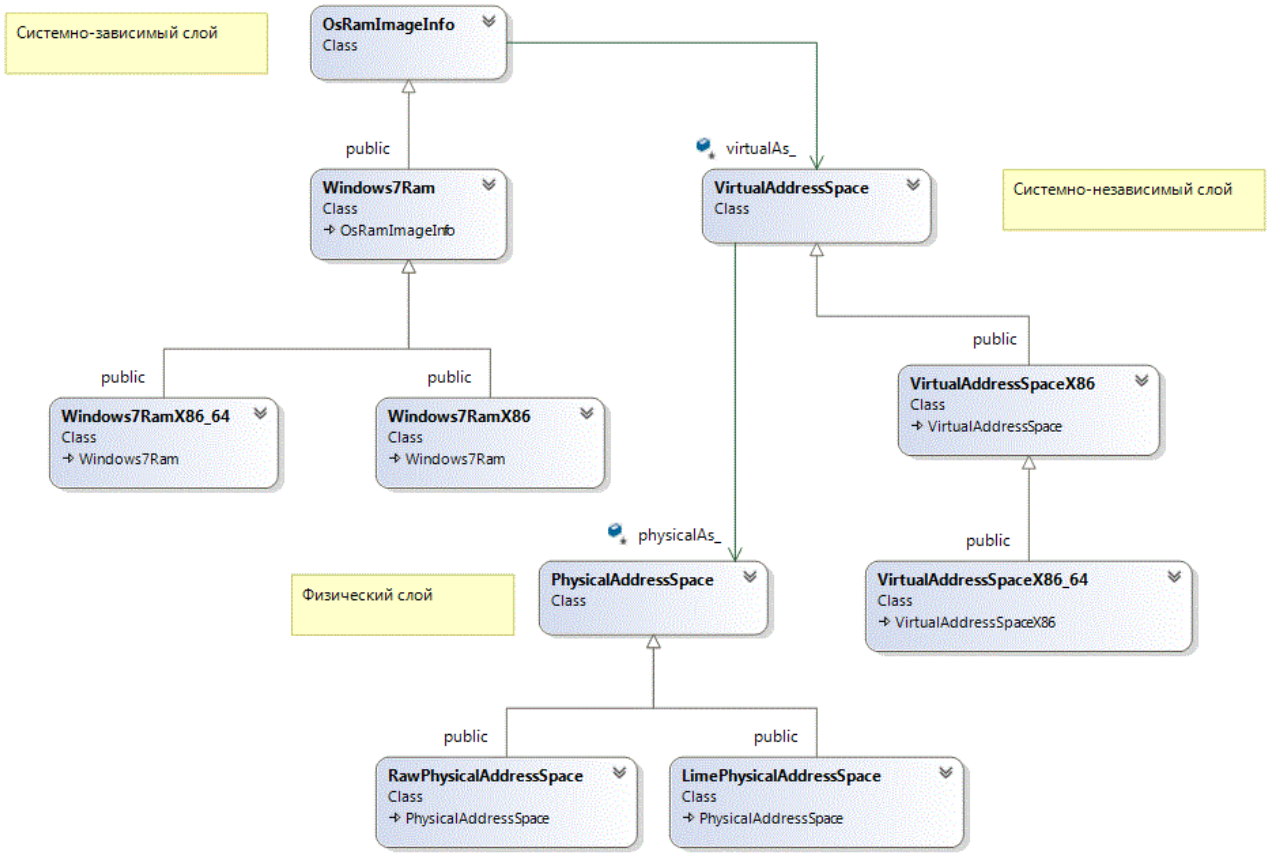

Рис. 14 Диаграмма классов разработанной архитектуры

Было выделено три основных слоя-интерфейса (Рис. 14):

1. Физический слой (PhysicalAddressSpace), абстрагирующий формат представления образов памяти. Например, образ может быть в «сыром» формате (получается при захвате памяти с помощью FastDump, RAM Capturer) или в формате LiME, но операция должна поддерживаться общая: чтение заданного количества данных, начиная с заданного смещения.
2. Системно-независимый слой (VirtualAddressSpace). Данный интерфейс объединяет операции, специфичные для разных аппаратных платформ (x86, x86_64, в будущем — ARM), поддерживая такие системно-независимые операции, как чтение данных по заданному виртуальному адресу.
3. Системно-зависимый слой. В него вынесена обработка страничных записей (PTE, PDE и т.п.), характер которой зависит от конкретного сочетания «операционная система — аппаратная платформа». Здесь обрабатываются, например, Prototype PTE.

Данная архитектура была реализована на языке C++ и успешно интегрирована в имеющийся программный продукт.

Заключение

Автором данной дипломной работы были получены следующие результаты:

- Проведено исследование предметной области компьютерного криминалистического анализа, а также механизмов управления памятью ОС Windows 7
- Предложена методика восстановления адресного пространства процесса из расширенного образа памяти
- Разработана и протестирована программа, реализующая упомянутое восстановление
- Разработанная методика восстановления адресного пространства интегрирована в коммерческий продукт
- Предложена методика восстановления физической памяти гостевой машины и реализована программа, осуществляющая восстановление памяти виртуальных машин под управлением VirtualBox и QEMU
- Проведена проверка корректности восстановленной памяти

Таким образом, все поставленные перед автором задачи были полностью решены.

Дальнейшие исследования

Логичным продолжением данной работы является исследование возможности восстановления физической памяти систем, исполняющихся под управлением гипервизоров первого типа (например, систем виртуализации KVM, Xen). Данные средства виртуализации широко применяются для организации облачных платформ и виртуального хостинга.

Ещё одним направлением исследований является анализ образа процесса виртуальной машины как чёрного ящика. Таким образом, возможно, получится решить проблему, связанную с недоступностью исходных кодов средств виртуализации (таких как VMware, Hyper-V). Одним из возможных подходов является создание приложения, которое выделяет в гостевой системе большие объёмы памяти, заполняя ее специальным образом, чтобы потом, сняв образ, можно было изучить характер распределения заполненных страниц по образу. Например, с помощью данного метода можно было бы легко понять, что в QEMU память гостевой машины выделяется одним большим блоком.

Список литературы

[1] Becher M., Dornseif M., Klein C., FireWire: all your memory are belong to us. PacSec conference, 2004, URL: http://pi1.informatik.uni-mannheim.de/filepool/presentations/0wned-by-an-ipod-hacking-by-firewire.pdf (дата обращения: 29.05.2013).

[2] Burdach M., An Introduction to Windows Memory Forensic. 2005, URL: http://forensic.seccure.net/pdf/introduction_to_windows_memory_forensic.pdf (дата обращения: 29.05.2013).

[3] Carrier B.D., Grand J., A Hardware-Based Memory Acquisition Procedure for Digital Investigations // Digital Investigation, 2004, Vol. 1, I. 1.

[4] Dolan-Gavitt B., Srivastava A., Traynor P., Robust Signatures for Kernel Data Structures. ACM Conference on Computer and Communications Security, 2009,
URL: http://www.cc.gatech.edu/~brendan/ccs09_siggen.pdf (дата обращения: 29.05.2013).

[5] Garcia G. L., Forensic physical memory analysis: an overview of tools and techniques. TKK T-110.5290 Seminar on Network Security, 2007,
URL: http://www.ccse.kfupm.edu.sa/~ahmadsm/coe589-121/garcia2007-memory-analysis-overview.pdf (дата обращения: 29.05.2013).

[6] Grier J., Detecting data theft using stochastic forensics // Digital Investigation, 2011, Vol. 8, P. 71-77.

[7] Gubanov Yu., Retrieving Digital Evidence: Methods, Techniques and Issues. URL: http://forensic.belkasoft.com/download/info/Retrieving%20Digital%20Evidence%20-%20Methods,%20Techniques%20and%20Issues.pdf (дата обращения: 29.05.2013).

[8] Gubanov Yu. Afonin O., Why SSD Drives Destroy Court Evidence, and What Can Be Done About It. URL: http://forensic.belkasoft.com/download/info/SSD%20Forensics%202012.pdf (дата обращения: 29.05.2013).

[9] Halderman J.A., Schoen S.D., Lest We Remember: Cold Boot Attacks on Encryption Keys // Communications of the ACM, 2009, Vol. 52, I. 5, P. 91-98.

[10] Intel 64 and IA-32 Architectures Software Developer's Manual, Volumes 1-5. 2008, URL: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (дата обращения: 29.05.2013).

[11] Kornblum J., Using Every Part of the Buffalo in Windows Memory Analysis // Digital Investigation, 2007, Vol. 4, I. 1, P. 24-29.

[12] LiME — Linux Memory Extractor. URL: http://lime-forensics.googlecode.com/files/LiME_Documentation_1.1.pdf (дата обращения: 29.05.2013).

[13] Martin A., FireWire Memory Dump of a Windows XP Computer: A Forensic Approach. 2007, URL: http://www.friendsglobal.com/papers/FireWire%20Memory%20Dump%20of%20Windows%20XP.pdf (дата обращения: 29.05.2013).

[14] Okolica J., Peterson G. L., Windows operating systems agnostic memory analysis // Digital Investigation, 2010, Vol. 7, Supplement, P. S48-S56.

[15] Oracle VM VirtualBox. URL: https://www.virtualbox.org/ (дата обращения: 29.05.2013).

[16] QEMU: open source processor emulator. URL: http://wiki.qemu.org/Main_Page (дата обращения: 29.05.2013).

[17] Russinovich M., Solomon D., Windows Internals, Fifth Edition ("Внутреннее устройство Windows, пятое издание"). Microsoft Press, 2009.

[18] Schuster A., Searching for Processes and Threads in Microsoft Windows Memory Dumps // Digital Investigation, 2006, Vol. 3, P. 10-16.

[19] The Volatility Framework: volatile memory artifact extraction utility framework. URL: https://www.volatilesystems.com/default/volatility (дата обращения: 29.05.2013).

[20] WindowsSCOPE: Forensics & Cyber Security Tools. URL: http://windowsscope.com/ (дата обращения: 29.05.2013).

[21] Zhang R., Wang L., Zhang S., Windows Memory Analysis Based on KPCR // IAS '09 Proceedings of the 2009 Fifth International Conference on Information Assurance and Security, 2009, Vol. 2, P. 677-680.