Федеральное государственное автономное образовательное учреждение высшего образования
«Московский физико-технический институт (государственный университет)»
Факультет радиотехники и кибернетики
Кафедра теоретической и прикладной информатики

Направление подготовки: 03.03.01 «Прикладные математика и физика»
Направленность: Инфокоммуникационные и вычислительные системы и технологии
Форма обучения: очная

Допущен к ГИА
Руководитель учебного подразделения

_____

Отладка операционной системы Windows под гипервизором QEMU/KVM

Бакалаврская работа

Обучающийся
Прутьянов Виктор Владимирович

_____

Научный консультант
Каган Роман Владимирович

_____

Научный руководитель
Зырянов Андрей Валериевич

_____

Москва 2018

Аннотация

На данный момент рабочая нагрузка большинства серверов выполняется с использованием технологии виртуализации, одним из видов которой является гипервизорная виртуализация. Крайне популярным гипервизором, в том числе для запуска Windows, является гипервизор QEMU/KVM.

Зачастую во время работы гостевых ОС возникают проблемы, такие как аварийные завершения и зависания. Один из методов решения проблем, возникающих при работе ОС Windows, — снятие снимков состояния системы, называемых дампами, которые в случае виртуальной машины могут быть сделаны со стороны гипервизора, и их анализ с помощью отладчика WinDbg. Целью данной работы является создание метода анализа проблем Windows, работающей внутри виртуальной машины под управлением QEMU/KVM.

В работе рассмотрены существующие методы создания отладочных дампов Windows со стороны гипервизора и их недостатки. Исследована структура полного отладочного дампа Windows, представлен метод сбора данных для создания дампа, не зависящий от версии Windows, предложен новый метод передачи с помощью виртуального устройства VMCoreInfo необходимой информации о гостевой ОС в гипервизор и создания с ее помощью полного дампа памяти Windows.

Представлена программная реализация этих методов в виде драйвера для гостевой Windows и компонента гипервизора QEMU, позволяющих со стороны гипервизора создавать отладочный дамп, который может быть проанализирован WinDbg, как во время работы системы, так и после аварийного завершения.

Содержание

1. Введение

В настоящее время большая часть рабочей серверной нагрузки выполняется с помощью различных технологий виртуализации[18]. Виртуализация позволяет на основе одной физической системы запускать множество независимых друг от друга изолированных сред, между которыми могут быть распределены вычислительные ресурсы физической системы.

Одним из видов виртуализации является гипервизорная виртуализация. В этом случае, специальное ПО, называемое гипервизором, который может быть как частью операционной системы, так и непосредственно выполняться на физическом оборудовании, позволяет разделить физические ресурсы между изолированными средами, которые называются виртуальными машинами[29]. Тогда физическое оборудование, на котором запущен гипервизор называется хостом, а виртуальные машины называются гостями.

Сейчас в качестве хостовой ОС на серверах часто используется ОС Linux. Один из компонентов этой системы – гипервизор KVM (Kernel Virtual Machine), который работает в связке с эмулятором QEMU. Операционные системы, работающие внутри хоста и гостей, могут быть различными в случае гипервизорной виртуализации, поэтому внутри виртуальной машины под управлением QEMU/KVM может быть запущена почти любая ОС, в том числе Linux и Microsoft Windows.

Виртуализация, в частности гипервизорная, является достаточно сложной процедурой, поэтому при работе гостевой ОС могут возникать проблемы, такие как зависания или аварийные завершения. Эти проблемы могут быть вызваны как ошибками в работе гостевой ОС, так и ошибками, связанными с неправильной работой гипервизора. Их можно решать методами, разработанными для физических машин, но все они требуют специальной настройки ОС, что не всегда возможно сделать, а виртуализация позволяет этого избежать. Существуют два метода анализа подобных проблем: отладка ядра ОС и анализ снимков состояния ОС, называемых дампами.

Для гостевой Linux оба метода удобны и не представляют большой сложности при работе с QEMU:

- QEMU предоставляет интерфейс для отладчика GDB, через который можно отлаживать ядро гостевой ОС Linux как обычную программу[5].
- QEMU позволяет сделать снимок состояния виртуальной машины, называемый дампом, в формате ELF, который затем можно анализировать утилитой crash.

Аналоги этих методов существуют и для случая, когда виртуальная машина работает под управлением Windows:

- Подключение через виртуальный последовательный порт или Ethernet отладчика WinDbg[24].
- Создание отладочного дампа в формате DMP, понятном отладчику WinDbg, и его анализ.

Поскольку, как правило, администрируют хостовые системы и гостевые системы разные люди, наиболее предпочтительным для любой гостевой ОС является создание отладочных дампов, так как это требует наименьшее количество действий (или вовсе не требует) от пользователя виртуальной машины, который может не обладать соответствующей квалификацией или желанием для настройки гостевой ОС в режиме отладки[23]. Также, метод создания и анализа отладочных дампов не требует специального воспроизведения условий, приводящих к ошибке. Таким образом, возможность снятия отладочных дампов имеет наибольшую ценность для разработчиков ПО виртуализации.

На данный момент метод, связанный с анализом дампов, в случае гостевой ОС Windows под управлением QEMU имеет существенные проблемы:

- QEMU не имеет встроенного средства для создания дампов памяти в формате DMP. Для получения дампа в этом формате в произвольный момент времени на живой системе можно попытаться создать дамп в формате ELF и конвертировать его в формат DMP, но это не всегда возможно, особенно на последних версиях Windows.
- В случае аварийного завершения Windows автоматически сохраняет дамп на диске, что сопровождается так называемым BSoD (Blue Screen of Death – синий экран смерти), но для этого, во-первых, система должна быть специальным образом настроена на создание полного дампа памяти[9], который наиболее полно отражает состояние системы (по умолчанию же сохраняется практически бесполезный малый дамп памяти), а во-вторых, дамп может быть не создан, например, если не хватает свободного места на диске, или по другим причинам[22].

Таким образом, одним из важных сценариев является создание дампа в момент возникновения BSoD, когда гостевая система по каким-либо причинам не способна это сделать. Поэтому представляется актуальным разработка метода создания отладочного дампа гостевой Windows в формате, понятном отладчику WinDbg, как в случае живой системы, так и на этапе аварийного завершения.

На данный момент выпущено много различных версии Windows, из которых сейчас действительно популярны серверные системы начиная с Windows Server 2008 R2

и десктопные системы начиная с Windows 7, которые работают на основе одного ядра Windows версии 6.1. Также, наиболее популярными сейчас являются 64-битные версии, поэтому в работе будут рассмотрены 64-битные версии Windows на основе этой и более новых версии ядра.

Целью данной работы является создание метода анализа аварийных завершений и зависаний Windows, работающей внутри виртуальной машины под управлением QEMU/KVM. Для этого требуется исследовать необходимые для создания отладочного дампа данные, а также придумать и программно реализовать метод создания дампов со стороны гипервизора.

Так как в различных версиях Windows определения структур данных ядра могут различаться[8][21], хорошей чертой метода была бы независимость от версии Windows. Это бы означало использование более прозрачной логики и увеличивало бы шансы метода на корректную работу с новыми версиями Windows.

В настоящей работе рассмотрено, из чего состоит полный дамп памяти Windows, какие данные требуются для его создания, существующие методы их сбора и предложен подход для создания дампа во время BSoD, а также расширение этого подхода для работы на живой системе, описана реализация этих методов.

2. Постановка задачи

- Исследовать, какие данные необходимы для создания полного дампа памяти и как они могут быть получены.
- Разработать метод снятия полных дампов памяти в формате DMP с 64-разрядной гостевой ОС Windows с версией ядра $\geqslant 6.1$, работающей под управлением гипервизора QEMU/KVM, в момент аварийного завершения.
- Добавить возможность снятия дампов с живой системы.
- Программно реализовать эти методы.

3. Основные используемые технологии и инструменты

QEMU

QEMU – эмулятор аппаратного обеспечения с открытым исходным кодом, основанный на динамической трансляции. Позволяет эмулировать процессоры различных архитектур (x86, PowerPC, ARM и другие) на различных хостовых процессорах (x86, PowerPC, ARM, MIPS и другие). Поддерживает два режима: эмуляция полной системы, при которой на виртуальном процессоре запускается операционная система (такая как Windows или Linux), и эмуляция пользовательского режима Linux, которая позволяет запустить Linux-программу, собранную для процессора одной архитектуры, на процессоре другой архитектуры[7]. В режиме системной эмуляции QEMU может эмулировать не только процессор, но и периферийные устройства, такие как сетевые карты и видеоадаптеры. В случае, когда архитектура хоста и целевая архитектура совпадают, часть кода может быть исполнена прямо на хостовом процессоре, но часть инструкций могут нарушить изоляцию виртуальной машины и должны быть эмулированы. В случае виртуализации x86-совместимого процессора на процессоре той же архитектуры, QEMU может использовать преимущества аппаратной виртуализации, при которой процессор, исполняя код, учитывает хостовый он или гостевой, упрощая таким образом задачу изоляции, и минимизируя долю кода, требующего эмуляции. Для этого требуется один из ускорителей (accelerators), такой как KVM, HAXM, Apple Hypervisor Framework или другой, кроме того, аппаратная виртуализация должна поддерживаться процессором[25].

KVM

KVM (Kernel-based Virtual Machine) – технология виртуализации, встроенная в ядро Linux[11]. Состоит из загружаемого модуля ядра kvm.ko и модуля, специфичного для процессора – kvm-intel.ko или kvm-amd.ko. Представляет из себя гипервизор второго типа – специальный дополнительный программный слой поверх хостовой ОС, который управляет гостевыми ОС, а эмуляцию и управление оборудованием делегирует хостовой операционной системе[28][2].

Для исполнения кода гостевых ОС на хостовом процессоре KVM использует одну из технологий аппаратной поддержки виртуализации, например, Intel VT-x или AMD-V (в зависимости от производителя хостового процессора), производя обработку инструкций, которые непосредственно на процессоре выполниться не могут. Исполнение прерывается каждый раз при возникновении особого события в хостовом процессоре – VMexit. Возникновение события VMexit означает, что инструкция, которая вызвала его появление, по какой-либо причине не может быть исполнена (например если это инструкция ввода-вывода или остановки процессора) на хостовом процессоре и должна быть обработана[10]. В этом случае KVM либо самостоятельно производит эмуляцию, не тратя время на переключения контекста, либо возращает управление userspace-программе, которая управляет KVM через устройство `/dev/kvm`.

Технологии аппартной виртуализации Intel VT-x и AMD-V заключаются именно в возможности запуска изолированного гостевого кода и обработки особых ситуаций, возникающих во время его выполнения.

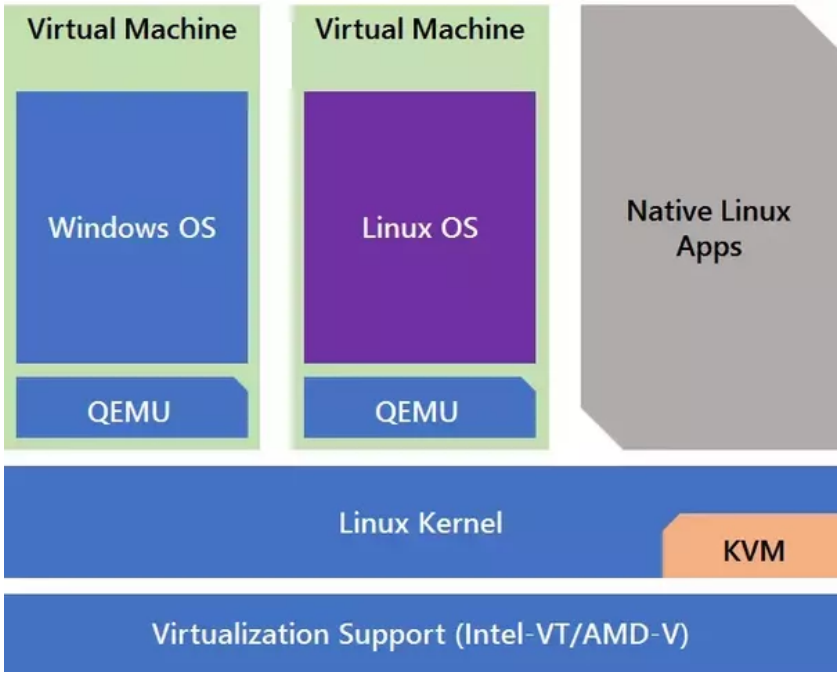

Рис. 1: QEMU/KVM

QEMU/KVM

Одним из наиболее популярных вариантов, в том числе на корпоративном рынке, является связка QEMU/KVM, которая позволяет виртуальной машине достичь производительности, близкой к производительности хостовой машины.

В таком режиме работы QEMU через системный интерфейс `/dev/kvm` создает виртуальные машины и виртуальные процессоры (vCPU), а затем, через тот же интер-

фейс, передает управление KVM для исполнения кода гостевой системы, а получает управление назад, когда KVM прерывает исполнение гостевого кода[17]. В частности, QEMU занимается эмуляцией работы периферии[1]. После того как QEMU завершит обработку, работу vCPU можно возобновлять.

QEMU является обычным пользовательским приложением, запускающим один процесс для каждого виртуального процессора KVM. Работа такого процесса, как и остальных процессов системы, управляется обычным планировщиком процессов Linux[17]. Ниже приведен псевдокод процесса QEMU, использующего KVM для исполнения гостевого кода:

```
open("/dev/kvm")
ioctl(KVM_CREATE_VM)
ioctl(KVM_CREATE_VCPU)
for (;;) {
    ioctl(KVM_RUN)
    switch (exit_reason) {
    case KVM_EXIT_IO:  /* ... */
    case KVM_EXIT_HLT: /* ... */
    }
}
```

WDK

WDK (Windows Driver Kit) – набор инструментов и библиотек для разработки драйверов под ОС Windows. В частности, включает в себя отладчики WinDbg и kd, а также утилиту dumpchk, которая предназначена для обнаружения ошибок в дампах аварийных завершений. Кроме того, набор содержит заголовочные файлы с описаниями некоторых структур ядра Windows.

WinDbg и kd

WinDbg и kd – отладчики от корпорации Microsoft, предназначенные для работы с пользовательским и системным кодом под ОС Windows. Оба эти инструмента содержат возможность анализа дампов аварийных завершений Windows. Данные отладчики позволяют использовать отладочную информацию для более удобного анализа дампа и определения причины аварийного завершения системы. Отладчики могут отобразить информацию из любой доступной отлаживаемой системе области вирутальной и физической памяти.

Кроме того, отладчики обладают такими возможностями как отображение содержимого регистров (рис. 2) и стека вызовов (рис. 3). Если для загруженных модулей

доступна отладочная информация, то элементами стека вызовов будут не просто адреса, а названия функций.

```
0: kd> r
rax=0000000000000008 rbx=0000000000000000 rcx=00000000ffffffff
rdx=00007ffb945467b0 rsi=0000000000000000 rdi=0000000000000000
rip=00007ff77b0f1088 rsp=000000ee424fff10 rbp=0000000000000000
 r8=000000ee424fe338  r9=0000024f8f7e1ba8 r10=0000000000000000
r11=000000ee424ffe10 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl nz ac po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000216
DummyTask!threadRoutine+0x18:
00007ff7`7b0f1088 ebf6            jmp     DummyTask!threadRoutine+0x10 (00007f
f7`7b0f1080)
```

Рис. 2: kd.exe отображает регистры и текущую функцию

```
0: kd> k
Child-SP          RetAddr           Call Site
000000ee`424fff10 00007ffb`978b1fe4 DummyTask!threadRoutine+0x18
000000ee`424fff40 00007ffb`97b3f061 KERNEL32!BaseThreadInitThunk+0x14
000000ee`424fff70 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

Рис. 3: kd.exe отображает стек вызовов

4. Структура отладочных дампов Windows

В ОС Windows существует несколько различных форматов дампов, набор которых различается в зависимости от версии Windows[19]. В данной работе рассматривается полный дамп памяти (complete memory dump), который доступен во всех рассматриваемых версиях Windows. Этот тип дампа содержит наибольшее количество информации и позволяет отладчику анализировать весь объем используемой системой физической памяти[3].

К сожалению, официальной спецификации формата DMP от компании Microsoft не существует. Наиболее полное описание структуры заголовка полного дампа может быть найдено в коде операционной системы Singularity (экспериментальная ОС с открытым исходным кодом от Microsoft)[15], хотя объяснения смысла полей нет и там.

Структура полного дампа (рис. 4) в упрощенном виде представляет из себя несколько крупных частей:

- Заголовок длиной в одну страницу (4 килобайт) на 32-разрядной системе и длиной 2 страницы (8 килобайт) на 64-разрядной системе.
- Снимки непрерывных регионов физической памяти (Run 0 – Run N). Их количество, начальные адреса и длины хранятся в заголовке.

Информацию о сохраненной в файле дампа физической памяти может отображать утилита dumpchk.exe (рис. 5), поставляющаяся вместе с отладчиком WinDbg. Как правило, регионов больше одного из-за так называемых PCI-окон (PCI-hole), поскольку часть физического адресного пространства используется для коммуникации с периферийными устройствами и не подходит сохранения данных. Таким образом, не все физическое адресное пространство сохраняется в файле дампа.

Каждый регион физической памяти описывается в заголовке дампа структурой _PHYSICAL_MEMORY_RUN64, где хранится начало и длина региона:

```
struct _PHYSICAL_MEMORY_RUN64 {
    ULONG64 BasePage;
    ULONG64 PageCount;
}
```

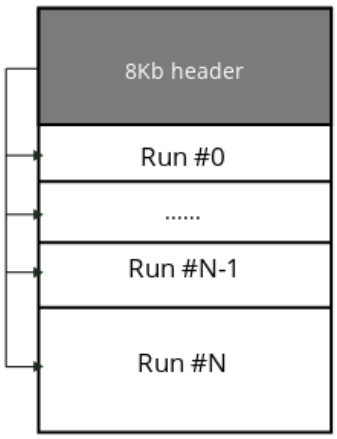

Рис. 4: Упрощенная схема полного дампа

```
Physical Memory Description:
Number of runs: 2 (limited to 2)
          FileOffset           Start Address           Length
           00000000`00001000     00000000`00001000     00000000`0009e000
           00000000`0009f000     00000000`00100000     00000000`7fedc000
Last Page: 00000000`7ff7a000     00000000`7ffdb000
```

Рис. 5: Список регионов физической памяти отображается утилитой dumpchk.exe

Кроме информации о регионах физической памяти, заголовок дампа содержит другие необходимые для работы отладчика поля, в том числе:

- BugcheckData – код ошибки и 4 параметра, которые описывают причину аварийного завершения (падения) системы. Их можно увидеть на синем экране во время падения (рис. 6). Отладчик WinDbg анализирует эти значения и выводит примерную причину падения (рис. 7).
- RequiredDumpSpace – суммарный размер дампа в байтах.
- DirectoryTableBase – физический адрес корня таблицы страниц, на основе которой отладчик будет производить страничное преобразование для доступа к данным по виртуальным адресам.
- PfnDatabase – виртуальный адрес базы данных PFN-номеров[26].
- PsLoadedModuleList – виртуальный адрес списка загруженных исполняемых модулей.
- MinorVersion, MajorVersion – два поля, вместе определяющие версию Windows.
- KdDebuggerDataBlock – виртуальный адрес специальной структуры ядра Windows, хранящей необходимую для работы отладчика информацию.

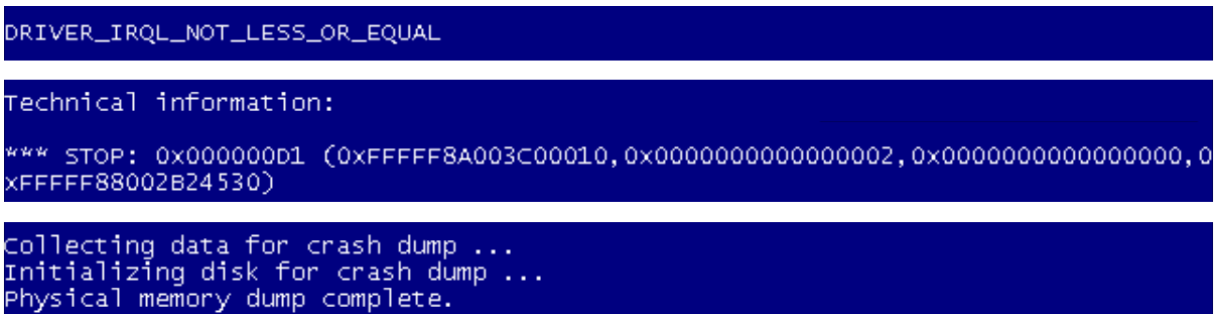

Рис. 6: Фрагменты BSoD

```
DRIVER_IRQL_NOT_LESS_OR_EQUAL (d1)
An attempt was made to access a pageable (or completely invalid) address at an
interrupt request level (IRQL) that is too high.  This is usually
caused by drivers using improper addresses.
If kernel debugger is available get stack backtrace.
Arguments:
Arg1: fffff8a005504010, memory referenced
Arg2: 0000000000000002, IRQL
Arg3: 0000000000000000, value 0 = read operation, 1 = write operation
Arg4: fffff880049cf530, address which referenced memory
```

Рис. 7: Фрагмент вывода команды `"!analyze -v"` WinDbg. Отладчик выводит причину падения.

KdDebuggerDataBlock

Путем модификации полей заголовка и анализа модифицированного дампа выяснилось, что данных из заголовка недостаточно для правильной работы отладчика, поскольку некоторые поля заголовка ссылаются на структуры, сохраненные в самой памяти. Наиболее важная из таких структур – KdDebuggerDataBlock, без адреса которой в заголовке анализ дампа практически невозможен отладчиком. Она содержит адреса других важных структуры данных ядра и смещения внутри них, например:

- KernBase – виртуальный адрес загруженного в память образа ядра Windows (ntoskrnl.exe). WinDbg использует этот адрес для загрузки подходящих отладочной информации, которая затем позволяет преобразовывать имена отладочных символов в адреса.

- KiProcessorBlock – указатель на массив указателей на PRCB (processor control block) – структуры, в которые ОС сохраняет данные по каждому используемому процессору[27].

- OffsetPrcbContext – в соответствии с названием, смещение структуры внутри PRCB, в которую ОС сохраняет регистровый контекст при аварийном завершении.

- OwnerTag – сигнатура KdDebuggerDataBlock – ASCII символы `"KDBG"`.

Следующие поля, содержащиеся в KdDebuggerDataBlock, имеют аналоги в виде полей из заголовка полного дампа памяти.

- KiBugcheckData – указатель на структуру с данными о падении, которые автоматически сохраняются в это поле системой во время аварийного завершения.
- MmPfnDatabase – указатель на базу данных PFN-номеров.
- PsLoadedModuleList – список загруженных модулей.

В отличие от заголовка дампа, структура содержимого KdDebuggerDataBlock описаны под названием _KDDEBUGGER_DATA64 в файле `wdbgexts.h`, который поставляется вместе с WinDbg. Эта структура обладает таким свойством, что с появлением новой версии Windows в ее конец только добавляются новые поля, а уже существующие остаются по тем же смещениям, на это можно полагаться независимо от версии Windows.

Можно заметить, что если такие поля заголовка как RequiredDumpSpace при определенных условиях можно вычислить и заполнить на основе данных из гипервизора, то такие поля как KdDebuggerDataBlock и PsLoadedModuleList известны только гостевой ОС.

Таким образом, для получения правильного дампа, нужно собрать вместе корректный заголовок и снимки непрерывных участков физической памяти. Снимки несложно сделать из QEMU, поскольку он имеет удобный доступ к физической памяти виртуальной машины. Получение же заголовка, как можно было понять из описания его полей, является нетривиальной задачей.

5. Обзор существующих решений

Поскольку проблема получения заголовка полного дампа памяти актуальна, попытки ее решения уже предпринимались. Поэтому рассмотрим несколько существующих решений и их недостатки.

Volatility

Volatility – набор программ с открытым исходным кодом для анализа памяти различных операционных систем. Содержит расширение под названием raw2dmp, которое позволяет конвертировать сырой (raw) снимок памяти Windows в формат, который может быть прочитан WinDbg.

Процедура получения дампа с использованием Volatility выглядит таким образом:

- Сбор дампа в формате ELF с помощью QEMU.
- Конвертация в формат RAW, который содержит только физическую память виртуальной машины, расширением imagecopy.
- Преобразование в формат DMP с помощью расширения raw2dmp[12].

Алгоритм работы расширения raw2dmp состоит в последовательном сканировании памяти и нахождении по сигнатуре структуры KdDebuggerDataBlock, который затем помогает создать корректный заголовок дампа.

На данный момент этот метод имеет следующие проблемы:

- После конвертации raw-дамп не хранит в себе информации о содержимом регистров виртуальных процессоров, поэтому после преобразования в дамп для WinDbg, содержимое регистров будет взято отладчиком из сохраненного системой контекста и будет неактуальным. В частности, это помешает восстановлению стеков вызовов (рис. 8).
- Начиная с Windows 8, система может на этапе загрузки зашифровать область памяти, в которой находится KdDebuggerDataBlock, поэтому его невозможно будет найти простым сканированием памяти и сравнением сигнатур.

```
Loading Dump File [D:\win7.dmp]
Kernel Complete Dump File: Full address space is available

Comment: 'File was converted with Volatility'

************* Symbol Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*c:\Symbols*http://msdl.microsoft.com/download/symbols
Symbol search path is: srv*c:\Symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows 7 Kernel Version 7601 (Service Pack 1) MP (2 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 7601.18247.amd64fre.win7sp1_gdr.130828-1532
Machine Name:
Kernel base = 0xfffff800`0284e000 PsLoadedModuleList = 0xfffff800`02a916d0
Debug session time: Tue Jun 12 06:36:00.312 2018 (UTC + 3:00)
System Uptime: 0 days 0:00:31.342
Loading Kernel Symbols
...............................................................
................................................................
...
Loading User Symbols

Loading unloaded module list
.....
Unknown exception - code 45474150 (first/second chance not available)
The context is partially valid. Only x86 user-mode context is available.
The wow64exts extension must be loaded to access 32-bit state.
.load wow64exts will do this if you haven't loaded it already.
*******************************************************************************
*                                                                             *
*                        Exception Analysis                                   *
*                                                                             *
*******************************************************************************

Use !analyze -v to get detailed debugging information.

***** Debugger could not find ntdll in module list, module list might be corrupt, error 0x80070057.

Probably caused by : ntkrnlmp.exe ( ntkrnlmp.exe!unknown_error_in_process )

Followup: MachineOwner
---------

16.1: kd:x86> k
ChildEBP          RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
00000000 00000000 0x0
16.1: kd:x86>
```

Рис. 8: WinDbg ошибочно интерпретирует отсутствующий контекст как 32-битный и не может отобразить стек вызовов на Windows 7

PVpanic и QEMU Guest Agent

PVpanic – виртуальное устройство на шине ISA, основным предназначением которого является генерация события в QEMU при аварийном завершении работы гостевой ОС.

QEMU Guest Agent – специальная служба внутри гостевой ОС, которая позволяет со стороны гипервизора производить различные операции, связанные с управлением гостевой ОС.

В 2017 году в драйвер PVpanic для Windows был добавлен ioctl-интерфейс, который позволяет пользовательскому приложению получать заголовок полного дампа памяти, с помощью вызова KeInitializeCrashDumpHeader (функция ядра Windows, сохраняющая заголовок полного дампа памяти, доступная драйверам[14]). По замыслу автора, этим приложением должен был быть QEMU Guest Agent[16]. QEMU Guest

Agent позволяет со стороны гипервизора производить различные действия внутри виртуальной машины. Для работы данного метода требовалось добавить возможность делать со стороны хоста ioctl-запросы к файлам в гостевой ОС. Но сообщество разработчиков QEMU не приняло соответствующий патч, объяснив это тем, что добавляемая логика работы слишком сложна и сильно зависит от типа гостевой ОС. Таким образом, этот метод так и остался не до конца реализованным.

6. Описание метода

Собрать заголовок на хосте из содержимого памяти и регистров сложно по нескольким причинам:

- Неизвестно, где находится начало таблицы страниц для системного контекста. Для этого можно было бы использовать доступное гипервизору содержимое регистров CR3. Но возможна ситуация, при которой на всех виртуальных процессорах происходит исполнение пользовательских приложений, а в последних версиях современных ОС для x86 большая часть системной памяти не отображается в адресное пространство пользовательских процессов.
- Поиск KdDebuggerDataBlock по сигнатуре затруднен, поскольку эта структура зашифрована в памяти новых версий Windows[20].
- Поиск начала образа ядра Windows в памяти тоже нетривиален, поскольку при загрузке системы он принимает случайное значение благодаря технологии KASLR (Kernel Address Space Layout Randomizing – рандомизация размещения адресного пространства ядра), которая используется в новых версиях Windows.

Руководствуясь этими фактами, а так же тем, что Windows предоставляет драйверам специальный интерфейс для получения заголовка дампа, решено разработать драйвер для гостевой системы, который сохранял бы заголовок в память и передавал его в гипервизор.

KeInitializeCrashDumpHeader

KeInitializeCrashDumpHeader – частично документированная функция, которая может быть вызвана драйвером для получения заголовка дампа[14]. В соответствии с официальной документацией, функция позволяет получить заголовок, который будет корректным всё время жизни системы, хотя и обладает следующими ограничениями:

- При изменении количества физической памяти заголовок должен быть получен снова.

- Полученный таким образом заголовок не содержит данных о возникшей ошибке (поле BugcheckData).

Также, согласно документации, начиная с Windows 8, адрес корня таблицы страниц (DirectoryTableBase) в полученном заголовке всегда соответствует системному контексту, но для более ранних версий контекст будет совпадать с контекстом текущего процесса, в том числе может быть пользовательским, что затем может помешать отладчику осуществлять доступ к системным структурам по виртуальным адресам.

Кроме того, путем замены заголовка в сохраненном системой дампе (например после искусственно вызванного BSoD) на заголовок, полученный драйвером, и анализа такого модифицированного дампа обнаруживаются ещё несколько несоответствий:

- Не заполнено поле RequiredDumpSpace.
- Не заполнено поле Context (содержимое регистров одного из процессоров).
- Поле PfnDatabase имеет неверное значение.

Выяснилось, что нулевое значение RequiredDumpSpace не позволяет WinDbg правильно анализировать память. Степень заполнения структуры Context не влияет на результат, но влияет значение аналогичных структур в самой памяти – это не позволяет WinDbg отобразить значения регистров и стека вызовов. Влияние значение PfnDatabase вообще никак не проявляется.

Как уже было сказанно выше, структура KdDebuggerDataBlock содержит поля, дублирующие поля в заголовке, и это можно использовать для их исправления. Способы восстановления полей заголовка собраны в соответствующей таблице:

Поле заголовка	Дамп на BSoD	Дамп живой системы
BugcheckData	Можно взять по адресу из поля KiBugcheckData внутри KdDebuggerDataBlock	Использовать код 0x161 (LIVE_SYSTEM_DUMP)
PfnDatabase	Можно взять из поля MmPfnDatabase внутри структуры KdDebuggerDataBlock	
RequiredDumpSpace	Можно вычислить на основе информации о сохраненных регионах физической памяти	
Context	Поле можно не заполнять, отладчик получает данные о значениях регистров каждого процессора из соответствующей ему структуры PRCB	

Таблица 1: Восстановление полей заголовка

Транспорт для заголовка

После того как заголовок получен, он должен быть передан в хост. Для этого можно использовать виртуальное устройство VMCoreInfo, которое изначально предназначено для передачи вспомогательных данных о ядре при создании дампов гостевой ОС Linux. VMCoreInfo является надстройкой над другим виртуальным аппаратным интерфейсом, который называется FwCfg и будет подробно расмотрен далее.

FwCfg

FwCfg – виртуальный аппаратный интерфейс, предоставляемый QEMU, позволяющий гостевому ПО обмениваться данными с хостом. С точки зрения гостевого ПО, представляет из себя устройство с портами ввода-вывода с номерами в диапазоне 0x510–0x51B. Предоставляет доступ к массиву записей (entry), которые являются просто блоками данных, и которым сопоставлено строковое имя. В зависимости от настроек, определяемых при создании каждой записи, может быть осуществлен доступ из гостевой ОС для чтения или модификации данных.

VMCoreInfo

VMCoreInfo – виртуальное устройство, доступ к которому осуществляется через запись FwCfg под названием `"etc/vmcoreinfo"`. Изначально создано для добавления информации о ядре Linux. При создании дампа в ELF-формате из QEMU эти данные автоматически присоединятся к нему. Обработка VMCoreInfo уже является частью функциональности создания дампов в формате ELF в QEMU, поэтому легко адаптируется для решаемой задачи. Также хорошим побочным эффектом является появление возможности создавать ELF-дампы, данных в которых потенциально достаточно их для конвертации в понятный WinDbg формат каким-либо внешним по отношению к QEMU инструментом.

Взаимодействие драйвера и VMCoreInfo

Для того чтобы осуществлять запись или чтение из любой FwCfg-записи, в частности VMCoreInfo, драйвер использует порты ввода-вывода в диапазоне 0x510–0x51B.

Чтение данных из записи производится по следующему алгоритму:

1. В порт 0x510 отправляется номер записи, данные которой нужно прочитать.
2. Из порта 0x511 побайтно считываются данные.

Чтобы записать данные в VMCoreInfo, нужно сначала эту найти запись по её названию (в данном случае – `"etc/vmcoreinfo"`), а также проверить что в данной версии FwCfg возможна передача данных из гостя в хост:

1. Считывается запись с номером 0x19 длиной 4 байта – это суммарное количество записей.

2. Считываются все записи и находится номер с подходящим названием.

3. Проверяется значение восьми байт, считанных из портов 0x514–0x51B, они должны иметь значение 0x51454d5520434647 – это означает, что устройством FwCfg поддерживается DMA-подобный (Direct Memory Access – прямой доступ к памяти) интерфейс записи.

В случае VMCoreInfo передается упакованная структура следующего вида:

```
struct FWCfgVMCoreInfo {
    uint16_t host_format;
    uint16_t guest_format;
    uint32_t size;
    uint64_t paddr;
} QEMU_PACKED;
```

Поля host_format и guest_format должны быть равны соответственно 0 и 1, size – размер передаваемых данных, paddr – их физический адрес.

Отправка данных осуществляется через DMA-подобный интерфейс:

1. В порты 0x514–0x51B записывается физический адрес экземпляра структуры FWCfgDmaAccess, содержащей физический адрес записываемых данных, их длину, а также номер записи и управляющие биты.

2. Происходит проверка управляющих битов, если они все равны 0, то передача произошла успешно.

После этого, QEMU будут доступны данные по адресу paddr. QEMU интерпретирует их как секцию ELF Note (рис. 9), которая состоит из названия (в данном случае `"VMCOREINFO"`) и содержимого – заголовка дампа[6]. Если структура секции корректна, то она становится доступна подсистеме создания дампов, и при создании ELF-дампа автоматически присоединяется к нему.

Схема связи между структурами, участвующими в передаче заголовка показана на рис. 10.

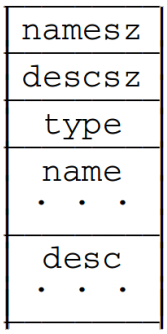

Рис. 9: Структура секции ELF Note

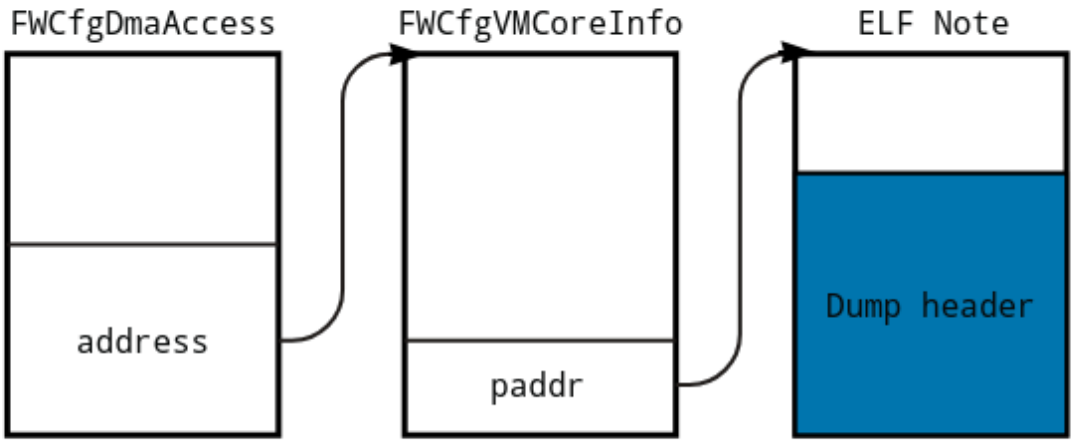

Рис. 10: Упрощенная схема связи между структурами

Команда dump-guest-memory в QEMU Monitor

QEMU предоставляет пользователю и программам специальный интерфейс управления собой – монитор, поддерживающий два протокола: HMP (Human Monitor Protocol), предназначенный для взаимодействия QEMU и пользователя через консоль, и QMP (QEMU Machine Protocol), основанный на формате JSON и позволяющим другим приложениям управлять работой QEMU. Оба эти механизма управления поддерживают команду dump-guest-memory, которая создает дамп гостевой памяти в указанном формате, например в формате ELF. Как уже было сказано выше, во время работы этой команды QEMU выгружает данные VMCoreInfo из соответствующего виртуального устройства, что делает работу с ними достаточно удобной. Кроме того, код QEMU содержит специальные средства для работы с физической и виртуальной памятью гостевой системы, а также для работы с регистрами виртуальных процессоров. Принимая во внимание эти факторы, а так же то, что часть процедуры создания

дампа общая для всех форматов, решено не делегировать создание дампа в формате DMP внешнему инструменту (хотя такая возможность по-прежнему остается), а добавить возможность сохранения в формате DMP в команду dump-guest-memory.

Регистровый контекст

В процессе анализа отладочного дампа значения значения регистров важны сами по себе, кроме того, их значения необходимы для корректного восстановления стека вызовов.

В файле `wdm.h`, который поставляется как часть WDK, есть определение структуры под названием CONTEXT. После сравнения этого определения, значений регистров из QEMU и поля Context из сохраненного Windows дампа, становится понятным, что поле Context содержит экрепляр именно этой структуры. Но поскольку на современных системах количество процессоров больше одного, регистровые контексты всех процессоров не могут поместиться в заголовке дампа (там выделено место ровно под один экземпляр контекста).

В структуре PRCB содержится поле ContextFrame. В сохраненном Windows полном дампе по адресу из этого поля содержится регистровый контекст процессора, соответствующего этому экземпляру структуры PRCB. Если эти структуры заполнены нулевыми значениями, то WinDbg не может восстановить контекст. Кроме того, в соответствии с описанием, структура контеста содержит поле флагов, один из которых обозначает, что контекст – 64-разрядный, поэтому, когда структура не заполнена, отладчик выводит сообщение, что доступен только 32-битный контекст (рис. 11). После заполнения структур контекста эти проблемы не возникают (рис. 12). Поэтому можно утверждать, что WinDbg берет содержимое регистров именно из этих структур.

```
BugCheck 161, {0, 0, 0, 0}

Probably caused by : Unknown_Image ( ANALYSIS_INCONCLUSIVE )

Followup:     MachineOwner
---------

16.0: kd:x86> r
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=00000000 edi=00000000
eip=00000000 esp=00000000 ebp=00000000 iopl=0         nv up di pl nz na po nc
cs=0000  ss=0000  ds=0000  es=0000  fs=0000  gs=0000             efl=00000000
00000000`00000000 ??              ???
16.0: kd:x86> k
ChildEBP          RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
00000000 00000000 0x0
16.0: kd:x86> .effmach amd64
Effective machine: x64 (AMD64)
16.0: kd> r
rax=0000000000000000 rbx=0000000000000000 rcx=0000000000000000
rdx=0000000000000000 rsi=0000000000000000 rdi=0000000000000000
rip=0000000000000000 rsp=0000000000000000 rbp=0000000000000000
 r8=0000000000000000  r9=0000000000000000 r10=0000000000000000
r11=0000000000000000 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up di pl nz na pe nc
cs=0000  ss=0000  ds=0000  es=0000  fs=0000  gs=0000             efl=00000000
00000000`00000000 ??              ???
16.0: kd> k
Child-SP          RetAddr           Call Site
00000000`00000000 00000000`00000000 0x0
```

Рис. 11: WinDbg определяет отсутствующий регистровый контекст как 32-битный и устанавливает режим x86. Нельзя увидеть ни стек, ни содержимое регистров.

Таким образом, чтобы WinDbg смог получить актуальные значения регистров, они должны быть сохранены в соответствующих структурах внутри дампа памяти. Независимая от версии Windows процедура восстановления регистровых контекстов может выглядеть так:

- Загрузка адресов PRCB всех процессоров из массива $KdDebuggerDataBlock \rightarrow KiProcessorBlock$.
- Нахождение адресов структур контекстов ContextFrame по их смещению внутри PRCB на основе значения поля $KdDebuggerDataBlock \rightarrow OffsetPrcbContext$.
- Заполнение контекстов, находящихся по этим адресам, значениями регистров.

KiBugcheckData

BugcheckData автоматически сохраняется при BSoD и эти данные можно просто скопировать в заголовок. Но оказывается, что при создании дампа живой системы недостаточно записать BugcheckData в заголовок. Для того чтобы отладчик мог воспользоваться этими данными, они также должны быть сохранены в $KdDebuggerDataBlock \rightarrow KiBugcheckData$.

```
BugCheck 161, {0, 0, 0, 0}

Probably caused by : Unknown_Image ( ANALYSIS_INCONCLUSIVE )

Followup:     MachineOwner
---------

0: kd> 0r
rax=0000000000000008 rbx=0000000000000000 rcx=000007fef2fe2108
rdx=000007fef2fe3400 rsi=0000000000000000 rdi=0000000000000000
rip=00000013f8f1088 rsp=00000000001ffb30 rbp=0000000000000000
 r8=00000000001fde38  r9=00000000004ea088 r10=0000000000000000
r11=00000000001ffa30 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl nz na pe nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
DummyTask!threadRoutine+0x18:
00000001`3f8f1088 ebf6            jmp     DummyTask!threadRoutine+0x10 (00000001`3f8f1080)
0: kd> 0k
Child-SP          RetAddr           Call Site
00000000`001ffb30 00000001`3f8f2220 DummyTask!threadRoutine+0x18
00000000`001ffb38 00000001`3f8f1070 DummyTask!`string'
00000000`001ffb40 00000000`00000000 DummyTask!threadRoutine
```

Рис. 12: Отладчик после загрузки регистрового контекста может определить стек вызовов, исполняющуюся инструкцию, исполняемый образ и с помощью отладочной информации вывести названия функций.

Алгоритм работы компонента QEMU

После того как заголовок доступен гипервизору, QEMU может приступать к созданию дампа.

1.* Виртуальная машина ставится на паузу.

2.* Производится синхронизация с KVM.

3. Значение поля RequiredDumpSpace вычисляется как сумма размеров непрерывных регионов физической памяти, присутствующих в дампе, и подставляется в заголовок.

4. Значение поля DirectoryTableBase из заголовка используется как значение регистра CR3 при дальнейшем доступе к виртуальному адресному пространству гостевой ОС из QEMU.

5. Адрес KdDebuggerDataBlock принимается равным адресу из соответствующего поля заголовка.

6. Значения поля PfnDatabase подставляется из поля $KdDebuggerDataBlock \rightarrow MmPfnDatabase$.

7. Значения полей BugcheckData (код и параметры ошибки)

   - В случае дампа в момент аварийного завершения подставляются из структуры по адресу $KdDebuggerDataBlock \rightarrow KiBugcheckData$.

- Код 0x161 (LIVE_SYSTEM_DUMP) и нулевые параметры ошибки подставляются в заголовок и в структуру по адресу $KdDebuggerDataBlock \rightarrow KiBugcheckData$ в случае дампа живой системы.

8. Регистровый контекст сохраняется по адресу из поля $KdDebuggerDataBlock \rightarrow KiProcessorBlock[i] \rightarrow ContextFrame$, где i – номер процессора, на основе значений регистров из QEMU (на этом этапе они уже имеют согласованное с KVM значение).

9. Заголовок записывается в файл.

10. Регионы физической памяти гостевой ОС на основе их описания в заголовке записываются в файл.

11.* Виртуальная машина продолжает работу.

Программная реализация помеченных * пунктов не требуется, так как эта часть общая для всех форматов и уже реализована в QEMU.

KdDebuggerDataBlock в Windows 8 и более новых версиях

Практика показывает, что KdDebuggerDataBlock, начиная с Windows 8, может быть зашифрован системой на этапе ее загрузки и быть недоступным во время работы (рис. 14), а расшифровывается только во время аварийного завершения (рис. 15, сделанный для той же самой системы). Поэтому описанный выше метод будет работать в случае BSoD на любой из рассматриваемых версий Windows, но для снятия дампа с живой системы алгоритмы работы как драйвера, так и гипервизора должны быть дополнены с учетом этого факта. Поскольку метод основан на работе гостевого драйвера, то представляется наиболее простым получать расшифрованный KdDebuggerDataBlock так же с помощью драйвера.

KdDebuggerDataBlock и Small Memory Dump

Наряду с полным дампом памяти, достаточно понятной является структура малого дампа памяти (Small Memory Dump). Windows создает дамп в этом формате после BSoD при настройках по умолчанию. Такой дамп содержит следующие данные:

- Структура BugcheckData
- Данные о процессе, в котором возникла ошибка (структура EPROCESS)
- Информация о потоке, в котором возникла ошибка (структура ETHREAD)

- Контекст процессора (структура PRCB), на котором возникла ошибка
- Список загруженных модулей
- Структура KdDebuggerDataBlock

В отличие от полного дампа памяти, KdDebuggerDataBlock хранится в файле дампа по записанному в его заголовке смещению, поэтому страничное преобразвание не требуется. Таким образом, имея малый дамп, можно с его помощью создать полный дамп памяти.

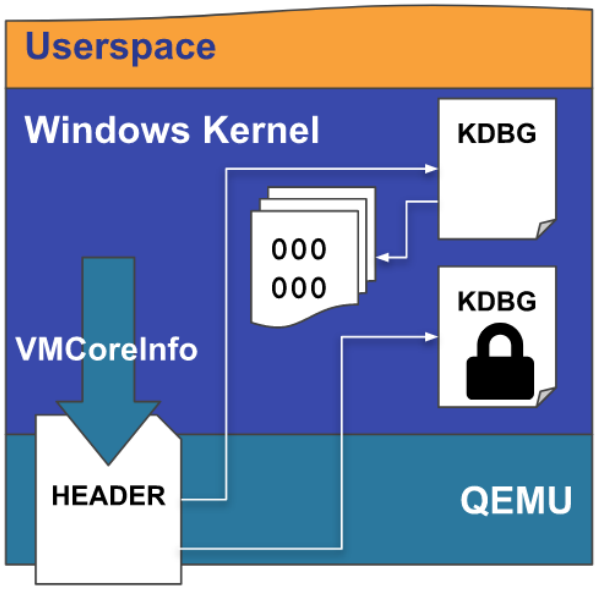

Рис. 13: Схема связи между элементами дампа на этапе их выгрузки из гостевой памяти

KeCapturePersistentThreadState

KeCapturePersistentThreadState – недокументированная функция ядра Windows. Драйвер может использовать ее для сохранения в памяти Small Memory Dump[4]. KdDebuggerDataBlock внутри такого дампа будет расшифрован.

Гостевой драйвер может передавать в гипервизор не только адрес оригинального KdDebuggerDataBlock (который может быть зашифрован), который система кладет в заголовок дампа, но и адрес его расшифрованной версии, которая будет храниться в памяти драйвера.

Для того чтобы гипервизор мог воспользоваться этой возможностью, он должен проверить сигнатуру KdDebuggerDataBlock, адрес которого лежит в своём обычном месте в заголовке, и, если сигнатура не совпадает, использовать KdDebuggerDataBlock, адрес которого может быть передан драйвером через одно из неиспользуемых полей

заголовка, например, BugcheckParameter1, поскольку это поле всё равно имеет нулевое значение и должно быть заполнено гипервизором.

Именно такой подход для передачи и обработки запасного KdDebuggerDataBlock и реализован в данной работе в гостевом драйвере и QEMU (рис. 13).

```
C:\WINDOWS\system32>kd.exe -z D:\0.DMP

Microsoft (R) Windows Debugger Version 10.0.15063.400 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.


Loading Dump File [D:\0.DMP]
Kernel Complete Dump File: Full address space is available


************* Symbol Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*c:\Symbols*http://msdl.microsoft.c
Symbol search path is: srv*c:\Symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
*******************************************************************************
THIS DUMP FILE IS PARTIALLY CORRUPT.
KdDebuggerDataBlock is not present or unreadable.
*******************************************************************************

Windows 10 Kernel Version 16299 MP (4 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS Personal
Machine Name:
Kernel base = 0xfffff801`cb806000 PsLoadedModuleList = 0xfffff801`cbb6c030
Debug session time: Wed Jun 13 11:29:15.163 2018 (UTC - 7:00)
System Uptime: 0 days 0:00:09.776

Loading Kernel Symbols
...............................................................
................................................................
..............................
Loading User Symbols
Unable to read KTHREAD address 0100000000001f80
Unable to get PEB pointer
Loading unloaded module list
....
CS descriptor lookup failed
Unable to get program counter
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

Use !analyze -v to get detailed debugging information.

BugCheck 161, {0, 0, 0, 0}

Unable to read KTHREAD address 0100000000001f80
Probably caused by : Unknown_Image ( ANALYSIS_INCONCLUSIVE )

Followup:     MachineOwner
---------

Unable to read KTHREAD address 0100000000001f80
WARNING: Unable to reset page directories
0: kd> db nt!KdDebuggerDataBlock L40
fffff801`cbb55910  e2 d4 19 8e 4c a0 c9 c4-e2 d4 19 8e 4c a0 c9 c4  ....L.......L...
fffff801`cbb55920  1d 0b 56 ed 98 67 93 39-e2 d4 19 8e 8c 41 ca c3  ..V..g.9.....A..
fffff801`cbb55930  e2 d4 19 8e 4c 81 c8 c1-1d 13 16 d6 8a 45 c9 3b  ....L........E.;
fffff801`cbb55940  15 13 16 d6 8a 45 c9 3b-e2 d4 19 8e 34 19 4e c6  .....E.;....4.N.
```

Рис. 14: KdDebuggerDataBlock зашифрован на Windows 10 в дампе, снятом во время работы системы

```
C:\WINDOWS\system32>kd.exe -z D:\1.DMP

Microsoft (R) Windows Debugger Version 10.0.15063.400 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.


Loading Dump File [D:\1.DMP]
Kernel Complete Dump File: Full address space is available


************* Symbol Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*c:\Symbols*http://msdl.microsoft.c
Symbol search path is: srv*c:\Symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows 10 Kernel Version 16299 MP (4 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS Personal
Built by: 16299.431.amd64fre.rs3_release_svc_escrow.180502-1908
Machine Name:
Kernel base = 0xfffff801`cb806000 PsLoadedModuleList = 0xfffff801`cbb6c030
Debug session time: Wed Jun 13 11:29:15.163 2018 (UTC - 7:00)
System Uptime: 0 days 0:00:09.776
Loading Kernel Symbols
...............................................................
................................................................
.............................
Loading User Symbols

Loading unloaded module list
....
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

Use !analyze -v to get detailed debugging information.

BugCheck D1, {ffffa9028b538010, 2, 0, fffff80756eb1530}

*** ERROR: Module load completed but symbols could not be loaded for myfault.sys
Probably caused by : myfault.sys ( myfault+1530 )

Followup:     MachineOwner
---------

0: kd> db nt!KdDebuggerDataBlock L40
fffff801`cbb55910  e0 1c b8 cb 01 f8 ff ff-e0 1c b8 cb 01 f8 ff ff  ................
fffff801`cbb55920  4b 44 42 47 68 03 00 00-00 60 80 cb 01 f8 ff ff  KDBGh....`......
fffff801`cbb55930  40 38 98 cb 01 f8 ff ff-00 00 00 00 00 00 00 00  @8..............
fffff801`cbb55940  00 00 00 00 00 00 01 00-b0 eb 97 cb 01 f8 ff ff  ................
```

Рис. 15: KdDebuggerDataBlock расшифрован на Windows 10 в дампе после BSoD. Можно заметить, что он находится по тому же адресу, что и на рис. 14

7. Заключение

В работе проведено исследование структуры полного дампа памяти ОС Windows. Для 64-разрядной ОС Windows на основе ядра с номером 6.1 или выше представлен метод получения этих данных, который не зависит от конкретной версии Windows и, возможно, будет работать и с появлением новых версий. Разработаны методы передачи необходимой информации об ОС в гипервизор и создания с ее помощью полного дампа памяти Windows. Таким образом выполнена задача по исследованию необходимых для отладочного дампа Windows данных и задача по разработке метода создания дампов со стороны гипервизора. Достигнута цель по созданию метода решения проблем в ОС Windows, работающей под управлением гипервизора QEMU/KVM, поскольку полученный таким способом дамп содержит всю требуемую отладчиком информацию о системе, который позволяет анализировать виртуальную память, регистры, стек вызовов и другие данные о системных и пользовательских процессах, присутствующих на системе в момент аварийного завершения или в момент создания дампа в случае дампа живой системы (может использоваться для анализа зависаний).

Методы сбора и передачи данных реализованы в виде драйвера, непосредственно собирающего необходимую для создания дампа информацию из гостевой системы, и патча для гипервизора QEMU, добавляющего команде dump-guest-memory новую опцию -w, которая позволяет напрямую, без конвертации, создавать дампы в формате DMP (может быть проанализирован WinDbg или kd.exe), если внутри гостевой системы работает драйвер. Кроме того, дампы в формате ELF также будут содержать всю необходимую для дальнейшей конвертации в формат DMP информацию.

Код драйвера одобрен сообществом разработчиков проекта KVM Windows Guest Drivers и включен в проект. Код компонента QEMU добавлен в один из продуктов компании Virtuozzo – Virtuozzo Hypervisor, а также находится в процессе принятия в основную ветку проекта QEMU.

8. Возможные направления дальнейшего развития проекта

Вариантами возможного продолжения данной работы являются следующие направления:

- Поскольку одним из вариантов использования технологии виртуализации является отказ от устаревшей аппаратной платформы и перенос ПО в соответствующую виртуализованную среду, следует рассмотреть добавление возможности работы представленного механизма с 32-разрядными версиями Windows.

- Существует способ динамического увеличения размера доступной виртуальной машине памяти под названием RAM hot-plug, который заключается в динамическом добавлении виртуальных модулей памяти. В настоящее время драйвер не создает новый заголовок в такой ситуации, но возможно сделать обработку этого случая с помощью стандартных событий.

- Одним из наиболее популярных при запуске виртуальных машин с операционной системой Windows является проприентарный гипервизор Hyper-V от корпорации Microsoft.
  Для того чтобы сделать дамп памяти гостевой Windows, работающей под управлением Microsoft Hyper-V, можно использовать одну из утилит Sysinternals Suite – LiveKd[13]. Эта программа позволяет по идентификатору виртуальной машины, работающей под управлением Hyper-V, сделать полный дамп памяти этой виртуальной машины и сохранить его на файловой системе хостовой операционной системы. Примечательно, что для работы этого механизма не требуется делать никаких настроек внутри виртуальной машины, в том числе не требуется устанавливать дополнительные драйверы.
  Таким образом, одним из возможных направлений дальнейшей работы является подробное исследование работы этого инструмента и попытка реализовать аналогичный механизм как компонент QEMU/KVM.

- В рассмотреной в работе схеме заголовок полного дампа памяти может быть получен как одна из секций дампа в формате ELF. На данный момент, такие утилиты как Volatility при попытке конвертации дампа из ELF формата не используют данные VMCOREINFO. Возможно добавить такой сценарий работы в Volatility, что позволило бы Volatility правильно производить конвертацию.

Список литературы

[1] Eric Blake. *Understanding QEMU devices*. URL: https://www.qemu.org/2018/02/09/understanding-qemu-devices/.

[2] Gillen A. Chen G. *KVM for Server Virtualization: An Open Source Solution Comes of Age*. 2011. URL: https://cs.nyu.edu/courses/fall14/CSCI-GA.3033-010/kvm4server.pdf.

[3] *Complete Memory Dump*. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/complete-memory-dump.

[4] *Crash dump u KeCapturePersistentThreadState*. URL: https://habr.com/post/40504/.

[5] *Debugging with QEMU*. URL: https://en.wikibooks.org/wiki/QEMU/Debugging_with_QEMU.

[6] *ELF Specification*. URL: http://www.cs.cmu.edu/afs/cs/academic/class/15213-s00/doc/elf.pdf.

[7] Bellard F. "QEMU, a Fast and Portable Dynamic Translator". B: *USENIX Annual Technical Conference* (2005). URL: https://www.usenix.org/legacy/event/usenix05/tech/freenix/full_papers/bellard/bellard.pdf.

[8] Chappell G. *KPRCB*. URL: http://geoffchappell.com/studies/windows/km/ntoskrnl/structs/kprcb/index.htm.

[9] *How to generate a kernel or a complete memory dump file in Windows Server 2008 and Windows Server 2008 R2*. URL: https://support.microsoft.com/en-us/help/969028/how-to-generate-a-kernel-or-a-complete-memory-dump-file-in-windows-ser.

[10] *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3*. Intel Corporation. Сент. 2016. URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf.

[11] *Kernel Virtual Machine*. URL: https://www.linux-kvm.org/page/Main_Page.

[12] Prosek L. *Extracting Windows VM crash dumps*. URL: https://ladipro.wordpress.com/2017/01/06/extracting-windows-vm-crash-dumps/.

[13] Russinovich M. *LiveKd for Virtual Machine Debugging.* URL: https://blogs.technet.microsoft.com/markrussinovich/2010/10/09/livekd-for-virtual-machine-debugging/.

[14] *Microsoft Docs. KeInitializeCrashDumpHeader function.* URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-keinitializecrashdumpheader.

[15] *Microsoft Research Singularity Source Code.* URL: https://searchcode.com/file/10186296/base/Windows/Inc/Dump.h.

[16] *[Qemu-devel] [PATCH] qga: implement guest-file-ioctl.* URL: https://lists.gnu.org/archive/html/qemu-devel/2017-02/msg00035.html.

[17] Hajnoczi S. *KVM Architecture Overview.* URL: https://vmsplice.net/~stefan/qemu-kvm-architecture-2015.pdf.

[18] *Server Virtualization and OS Trends.* URL: https://community.spiceworks.com/networking/articles/2462-server-virtualization-and-os-trends.

[19] *Varieties of Kernel-Mode Dump Files.* URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/varieties-of-kernel-mode-dump-files.

[20] *Volatility Labs: The Secret to 64-bit Windows 8 and 2012 Raw Memory Dump Forensics.* URL: https://volatility-labs.blogspot.com/2014/01/the-secret-to-64-bit-windows-8-and-2012.html.

[21] *Windows 10 Internal Kernel Structures.* URL: http://www.tssc.de/winint/Win10_16299_ntoskrnl/_KPRCB.htm.

[22] *Windows does not save memory dump file after a crash.* URL: https://support.microsoft.com/en-us/help/130536/windows-does-not-save-memory-dump-file-after-a-crash.

[23] *Windows driver debugging with WinDbg and VMWare.* URL: https://briolidz.wordpress.com/2012/03/28/windows-driver-debugging-with-windbg-and-vmware/.

[24] *Windows Guest Drivers. Windows Guest Debugging.* URL: https://www.linux-kvm.org/page/WindowsGuestDrivers/GuestDebugging.

[25] Джонс М. *Эмуляция систем с помощью QEMU.* URL: https://www.ibm.com/developerworks/ru/library/l-qemu/index.html.

[26] Ионеску А. Руссинович М. Соломон Д. *Внутреннее устройство MS Windows. 6-е изд. Основные подсистемы ОС.* Санкт-Петербург, 2014. ISBN: 978-5-496-00791-7.

[27] Соломон Д. Руссинович М. *Внутреннее устройство Microsoft Windows. 6-е изд.* ISBN: 978-5-459-01730-4.

[28] Бос Х. Таненбаум Э. *Современные операционные системы, 4-е изд.* Санкт-Петербург, 2015, с. 95—98. ISBN: 978-5-496-01395-6.

[29] Остин Т. Таненбаум Э. *Архитектура компьютера, 6-е изд.* Санкт-Петербург, 2013, с. 500—501. ISBN: 978-5-496-00337-7.