Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский физико-технический институт
(государственный университет)»

Факультет радиотехники и кибернетики

Кафедра теоретической и прикладной информатики

СОЗДАНИЕ ОТЛАДОЧНЫХ ДАМПОВ ГОСТЕВОЙ ОС WINDOWS

Выпускная квалификационная работа
(бакалаврская работа)

Направление подготовки: 03.03.01 Прикладные математика и физика

Выполнил:
студент 311 группы _____________ Ануфриев Роман Андреевич

Научный руководитель: _____________ Костюшко Алексей Владиславович

Москва
2017

Оглавление

Оглавление 2
1. Введение 3
2. Актуальность 4
3. Постановка задачи 6
4. Обзор инструментов 7
5. Реализация 15
    5.1 Формат дамп файлов 15
    5.2 Подготовка окружения 19
    5.3 Proof of concept 23
    5.4 Архитектура ПО 31
6. Результаты 35
7. Дальнейшее развитие 36
8. Список литературы и ссылки на источники 37

1. Введение

Существует 3 основных вида виртуализации [15]: виртуализация уровня приложения, уровня операционной системы и уровня аппаратного обеспечения. В работе рассматривается виртуализация уровня аппаратного обеспечения, при которой можно на компьютере с операционной системой (ОС) одного семейства (например, Linux) при помощи виртуализационного программного обеспечения (ПО) запускать операционную систему другого семейства (например, Windows). При таком типе виртуализации, ОС, на которой запущено виртуализационное ПО (виртуальная машина) называется хостом (от англ. host – хозяин; человек, принимающий гостей), а ОС, запущенная внутри виртуальной машины называется гостем.

Виртуализация уровня аппаратного обеспечения является технически сложной процедурой, в процессе которой часто возникает необходимость проверять состояние гостевой операционной системы пока она работает или выяснять причину ошибки, которая привела к ее аварийному завершению. Одним из способов, как это можно делать является снятие дампов (от англ. dump – груда, куча) памяти.

Дамп памяти гостевой ОС представляет из себя файл, содержащий некоторое количество оперативной памяти гостя и заголовок, по которому хост может понять где в этой памяти лежит нужная ему информация.

Анализируя такие дампы можно не только понять почему возникла та или иная ошибка и исправить (отладить) ее, но и получить дополнительную информацию о внутренней структуре гостевой операционной системы, которая может помочь в улучшении производительности виртуальной машины и добавлении нужной функциональности.

2. Актуальность

Из предыдущего раздела стало ясно, что возможность удобного и быстрого создания отладочных дампов очень важна для разработчиков виртуализационного ПО, так как дампы помогают на этапе разработки и отладки, а также при анализе ошибок ПО, которые случаются у пользователей продукта.

Чтобы раскрыть суть проблемы, рассматриваемой, в данной работе, нужно перечислить стандартные способы создания дампов, предоставляемые ОС Windows и их недостатки:

1) ОС Windows в автоматическом режиме создает дампы памяти, если произошла ошибка, которая привела к аварийному завершению системы (BSOD – Blue Screen Of Death – синий экран смерти – изображение, которое показывается на экране компьютера при аварийном завершении ОС Windows). Недостаток этого способа в том, что невозможно получить дамп памяти работающей системы.
2) Можно подключить отладчик ядра ОС (WinDbg, подробнее о нем будет рассказано далее) и из него соответствующей командой инициировать создание дампа. При помощи этого способа можно получить дамп работающей системы, но использование отладчика предполагает довольно сложные действия по настройке окружения и отрицательно влияет на производительность системы в целом. Поэтому постоянное использование этого способа, а тем более необходимость рядовому пользователю продукта совершать сложную настройку для того, чтобы помочь разработчикам в решении их проблем, неприемлемо.

Получается, что стандартного способа создания дампов без недостатков, перечисленных выше, нет. Поэтому разработчики виртуальных машин пишут собственные программы, которые удовлетворяют их требованиям.

Как уже было сказано выше, дамп состоит из заголовка и оперативной памяти гостя. Разработчику виртуализационного ПО не составляет сложности взять оперативную память гостя, ведь виртуальной машине, внутри которой запущен гость, известно где она находится, проблемы возникают с тем, откуда взять заголовок. На сегодняшний момент, из-за закрытости исходных кодов ОС Windows и недокументированности стандартного формата дампа, у разработчиков нет возможности написать программу, которая бы надежно, по стандарту формировала заголовок дампа. Поэтому, на данный момент, приходится вручную, для каждой версии ОС Windows (а расположение тех или иных элементов, на которые должен ссылаться заголовок может меняться даже не от версии к версии, а от патча к патчу) составлять структуры для заголовка. Это негибкий и трудозатратный метод решения этой проблемы, и код библиотеки, который написан с использованием этого решения выглядит нелаконично – множество однотипных захардкоженных (от англ. hardcoding – считающаяся плохой практика программирования, которая заключается во встраивании значений в исходный код программы, вместо получения их динамически и, как следствие, невозможность изменить эти параметры без изменения исходного кода) структур, в которых меняется несколько полей в зависимости от версии Windows.

Получается, что разработка решения, позволившего бы в автоматическом режиме создавать дампы для любой версии ОС Windows позволила бы существенно сократить затраты ресурсов разработчиков на адаптацию кода под новые версии ОС. Исследование возможности написания такого решения и приводится в данной работе.

3. Постановка задачи

Цель данной работы состоит в исследовании возможности реализации автоматического создания отладочных дампов гостевой ОС Windows и написании proof of concept (доказательство концепции – реализация идеи для того, чтобы показать ее осуществимость) программ для более плотного знакомства с технологиями.

Задачи, которые нужно выполнить:

- Узнать стандартный формат дампа
- Узнать откуда можно динамически получать поля заголовка дампа
- Подготовить окружение для сборки и отладки программ и драйверов (драйвер – это ПО, с помощью которого операционная система получает доступ к аппаратному обеспечению компьютера) под ОС Windows
- Разработать архитектуру готового программного решения для создания дампов в автоматическом режиме
- Написать самому или найти готовые proof of concept программы, подтверждающие реализуемость используемых методов.

4. Обзор инструментов

В этом разделе будет дан обзор всех используемых в данной работе программ, инструментов и технологий, а также краткое обоснование почему были выбраны именно они.

ОС Windows

На сегодняшний день наиболее популярными для виртуализации операционными системами являются ОС семейства Windows и Linux [27]. На Linux нет проблемы создания дампов живой системы (live dump), так как исходные коды ОС открыты и форматы дампов известны, поэтому предметом исследования была выбрана ОС семейства Windows.

Visual Studio + Windows Driver Kit

Visual Studio – это интегрированная среда разработки от Microsoft, которая вместе с Windows Driver Kit позволяет удобно разрабатывать драйвера под Windows, а также подписывать их тестовой подписью для установки на тестировочный компьютер (если разработчик устанавливает свой драйвер на 64-битную версию Windows, то он должен его подписать, иначе ОС после установки драйвера не загрузится потому, что Windows не подписанные драйверы (т.е. драйверы из ненадежного источника) считает потенциально опасными для своей работоспособности).

WinDbg

Универсальный (может быть использован для отладки пользовательских приложений, драйверов и ядра операционной системы) отладчик для ОС Windows. Также может быть использован для просмотра и отладки дампов памяти.

QEMU

Популярная [29], свободно распространяемая программа, с открытым исходным кодом, которая обеспечивает виртуализацию уровня аппаратного обеспечения. Без дополнительных модулей работает в режиме программной эмуляции аппаратного обеспечения, что значительно замедляет работу виртуальной машины относительно исполнения этой ОС на реальном оборудовании.

KVM

Программное решение с открытым исходным кодом, которое обеспечивает поддержку аппаратной виртуализации (не путать с виртуализацией уровня аппаратного обеспечения) в среде Linux. Состоит из загружаемого модуля ядра (kvm.ko), обеспечивающего базовую функциональность, и процессорно-специфического модуля (kvm-amd.ko или kvm-intel.ko). Сама по себе не выполняет эмуляцию, а предоставляет в пространство пользователя интерфейс, позволяющий настроить гостевую ОС для использования аппаратной виртуализации, что значительно ускоряет работу виртуальной машины, относительно метода программной эмуляции.

QEMU/KVM

Очень популярная связка [28]: QEMU работает в пространстве пользователя и использует интерфейс, предоставляемый KVM для доступа к возможностям аппаратной виртуализации процессора.

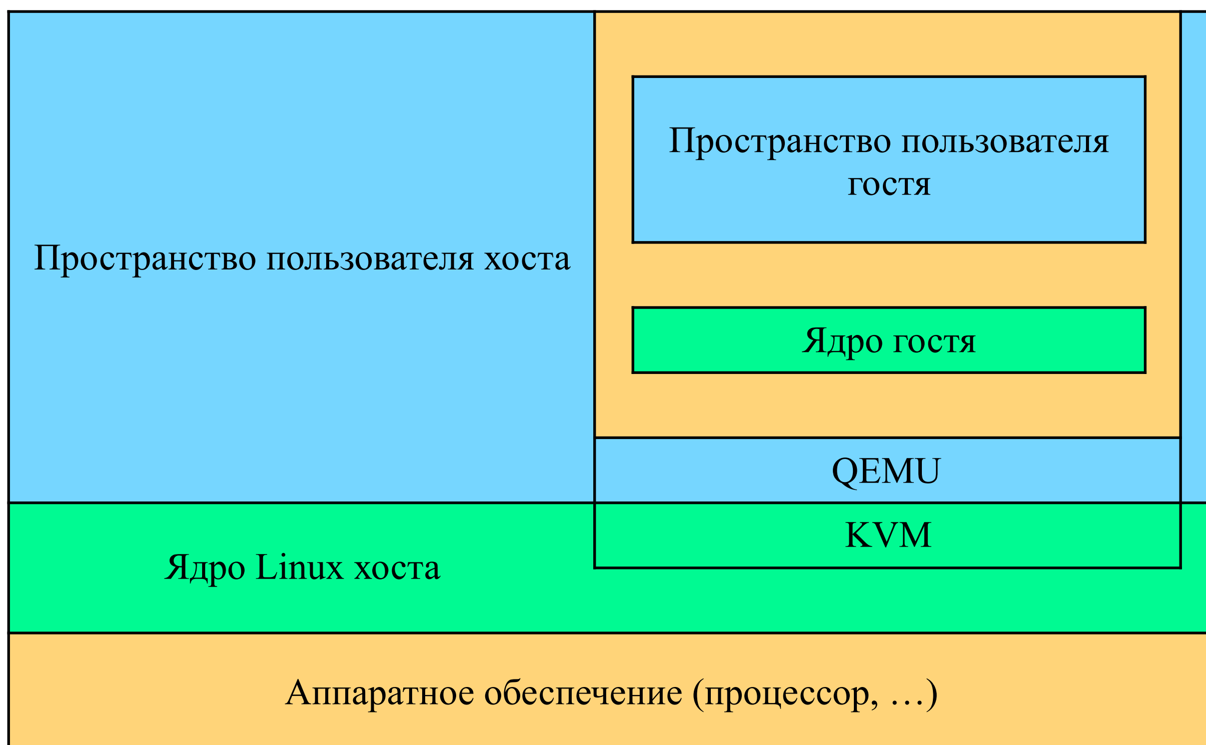

Рисунок 4.1. Архитектура QEMU/KVM [1]

Поскольку KVM существует только для ядра Linux, то в качестве хоста может использоваться только ОС семейства Linux, что не является минусом, так как Linux – удобная для разработки ОС с открытым исходным кодом.

VirtIO

Полная виртуализация или программная эмуляция аппаратного обеспечения удобна тем, что она позволяет запустить любую операционную систему без изменений внутри виртуализационного ПО. Но этот подход имеет плохую производительность и довольно сложный в реализации, так как приходиться эмулировать физические устройства, такие как сетевые карты, жесткие диски и т.д.

Другой подход – это паравиртуализация [16]. При его использовании гостевая ОС или какая-то ее часть (например, драйвер устройства) «знает», что она запущена в виртуальной машине, а не на реальном аппаратном обеспечении. Таким образом, работая в паре с гипервизором (виртуализационным ПО), паравиртуализированное ПО «знает», как

обмениваться с ним информацией не спускаясь на физический уровень. Этот подход намного производительнее и проще в разработке, но подразумевает изменение исходных кодов ПО. Так как в работе рассматривается ОС семейства Windows, у которой исходный код закрыт, то паравиртуализация самой ОС затруднена, но можно паравиртуализировать драйверы устройств.

Virtio – это основная платформа для паравиртуализации устройств ввода-вывода в KVM [30]. Технология имеет понятную структуру и предоставляет удобный интерфейс для написания гостевого (front-end драйвер) и хостового драйвера (back-end драйвер), которые общаясь между собой через высокоуровневые абстракции обеспечивают виртуализацию устройства (сетевую карту, жесткий диск и т.д.) с хорошей производительностью.

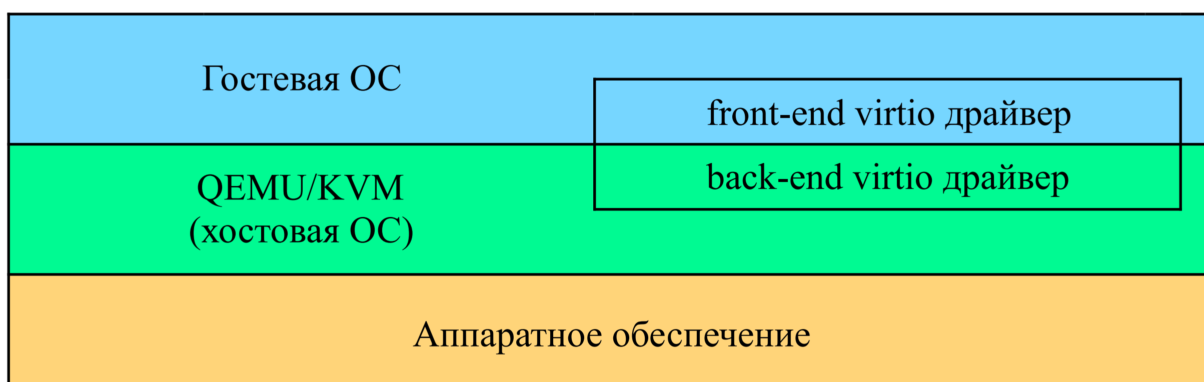

Рисунок 4.2. Упрощенная схема работы паравиртуализированных virtio драйверов [2]

Вместе с гостевым и хостовым драйверами, в virtio определяется еще два уровня, предназначенные для поддержки общения гостя с гипервизором. На верхнем уровне находится интерфейс для работы с виртуальными очередями, который соединяет front-end и back-end драйверы. Драйверы могут использовать произвольное количество очередей, в зависимости от своих потребностей. Например, virtio драйвер сетевой карты обычно использует две виртуальные очереди (одна для приема и другая для передачи данных), а virtio драйвер блочного устройства только одну [17].

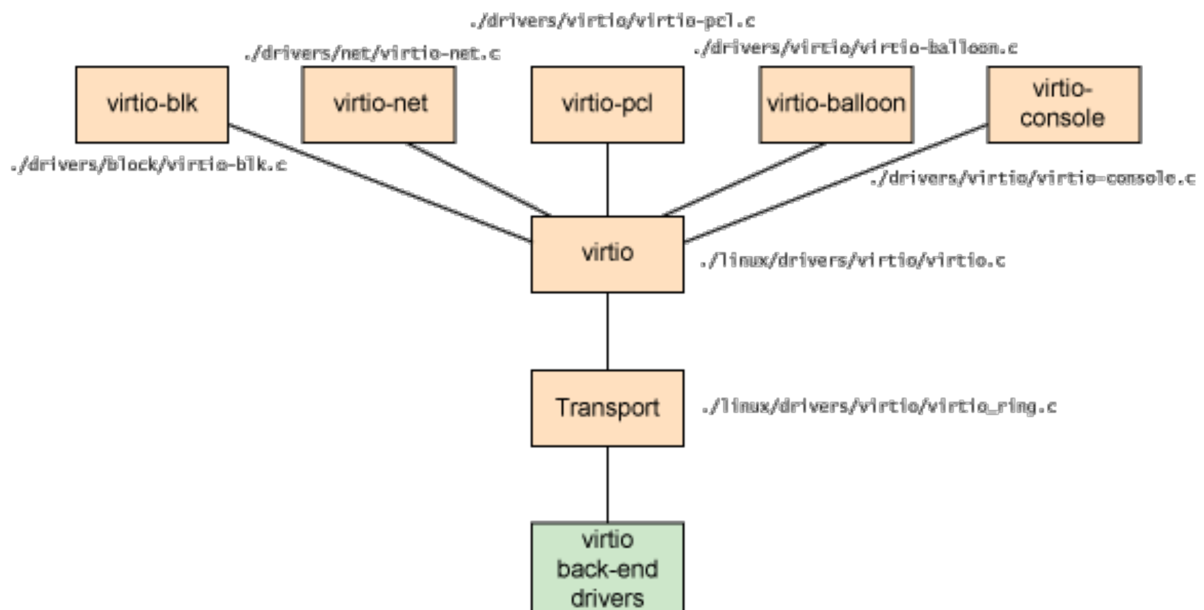

Рисунок 4.3. Высокоуровневая архитектура фреймворка virtio

Виртуальные очереди, на самом деле, реализованы в виде кольцевых структур (virtio_ring), через которые и происходит общение между гостем и гипервизором. Но их можно реализовать любым удобным способом, главное чтобы в госте и гипервизоре реализации совпадали.

На Рисунке 4.3 слева направо изображены 5 front-end драйверов: для блочных устройств (например жестких дисков), сетевых устройств, для эмуляции шины PCI, balloon драйвер (для динамического управления памятью гостя) и драйвер консоли. Каждый front-end драйвер имеет соответствующий back-end драйвер в гипервизоре.

С точки зрения гостя, иерархия объектов фреймворка virtio определена так, как показано на Рисунке 4.4. Сверху находится virtio_driver, который представляет собой front-end драйвер в госте. Устройства, которые соответствуют этому драйверу описываются структурами virtio_device. При помощи virtio_config_ops описываются операции для конфигурирования соответствующего virtio_device.

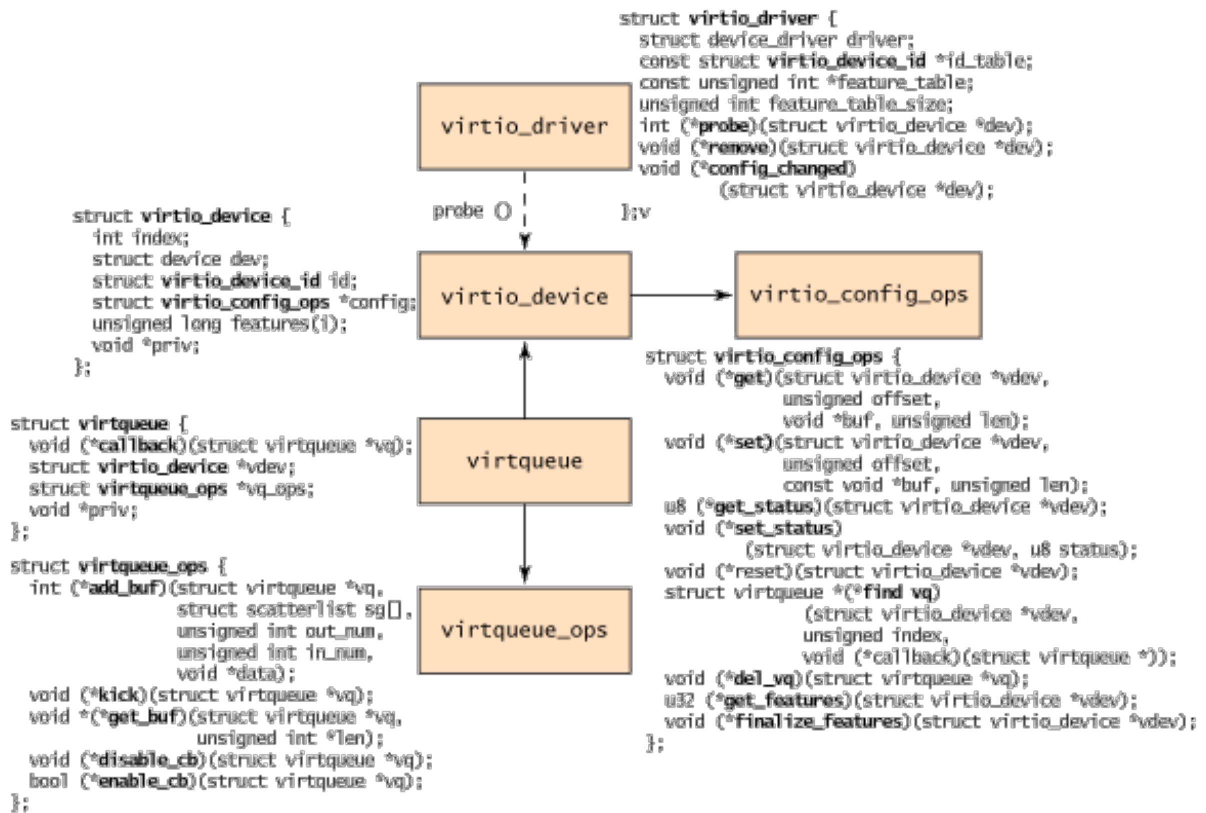

Рисунок 4.4. Иерархия объектов front-end драйвера virtio

virtqueue – структура виртуальной очереди, содержит указатель на тот virtio_device, который она обслуживает и, также, она ссылается на структуру virtqueue_ops, которая определяет все операции над этой очередью при работе с back-end драйвером в гипервизоре.

Процесс обнаружения устройства начинается с создания virtio_driver и последующей его регистрации при помощи register_virtio_driver. Структура virtio_driver, помимо всего прочего, содержит в себе массив с идентификаторами устройств (id_table), которые этот драйвер поддерживает. Когда гипервизор обнаруживает новое устройство, чей идентификатор совпадает с одним из идентификаторов из id_table какого-то virtio_driver, он вызывает функцию probe из этого драйвера, чтобы передать ему структуру virtio_device, представляющую новое устройство, которое может быть дополнительно настроено через вызов функций из структуры virtio_config_ops.

Для общения с устройством нужно узнать какие виртуальные очереди (virtqueue) обслуживают этот virtio_device, это делается при помощи вызова функции find_vq из virtio_config_ops.

Виртуальные очереди содержат внутри себя буферы, в которых и происходит передача информации.

В найденной virtqueue содержится указатель на структуру virtqueue_ops, которая определяет список функций, с помощью которых команды и данные перемещаются между гостем и гипервизором. Используя функцию add_buf гость предоставляет запрос гипервизору, где указывает в каких буферах виртуальной очереди записана информация для гипервизора, а в каких гость ожидает увидеть ответ от него. После того, как гость сформировал все запросы (т.е. заполнил все буферы, которые он хотел), он уведомляет гипервизор о них используя функцию kick.

Ответы от гипервизора можно получить используя функцию get_buf. Гость может просто время от времени вызывать эту функцию, пока не получит положительный результат, либо ожидать, пока его уведомит гипервизор через функцию callback, о том, что он сформировал ответ и только после этого вызвать get_buf. Механизм callback может быть включен или выключен используя функции enable_cb и disable_cb соответственно.

Также важно заметить, что формат, порядок и содержимое буферов является значимым только только для front-end и back-end драйверов. Все механизмы передачи данных осуществляют только перемещение этих буферов и ничего не знают об их внутренней структуре.

QEMU Guest Agent

Это программа в госте, работающая в фоновом режиме, которая предоставляет приложениям в хосте, интерфейс для выполнения некоторых системных функций внутри гостя. Например, с ее помощью можно программой из хоста прочитать или записать информацию из файла, лежащего на диске гостя. Эти и другие функции QEMU Guest Agent

для передачи информации между гостем и хостом реализованы при помощи механизма virtio, описанного выше.

Parallels Desktop for Mac

Программный продукт, который обеспечивает виртуализацию уровня аппаратного обеспечения на операционной системе macOS. Одна из самых удобных и производительных виртуальных машин для этой платформы. Используется в данной работе, так как один из ноутбуков, на котором производилась настройка окружения – это MacBook Air на macOS.

socat

Утилита командной строки на Unix-подобных системах (Linux, macOS, …), которая служит для двунаправленной передачи данных по различным протоколам между двумя независимыми каналами. В качестве каналов могут выступать файлы, устройства, сокеты и многое другое.

5. Реализация

В этом разделе будут подробно рассмотрены этапы достижения задач, описанных в разделе 3. Сначала будет приведено описание формата дамп файла и того, как его можно создать программным путем. Затем, станет понятно каким образом нужно настроить окружение для того, чтобы удобно писать, отлаживать и запускать драйвера ОС Windows. И, наконец, будет приведена полная идея работы, подтвержденная proof of concept программами.

5.1 Формат дамп файлов

Всего существует 5 разновидностей дамп файлов ядра Windows [3], основное различие между которыми это размер и, соответственно, количество информации, которое в них содержится:

1. **Complete Memory Dump** (полный дамп памяти) – самый большой по размеру и количеству информации дамп, содержит всю физическую оперативную память, которую использует Windows.
2. **Kernel Memory Dump** (дамп памяти ядра) – значительно меньше полного дампа, содержит всю память, используемую ядром Windows на момент снятия дампа.
3. **Small Memory Dump** (малый дамп памяти) – намного меньше полного и ядерного дампа, имеет фиксированный размер в 64 килобайта, полный список того, что он содержит можно посмотреть в источнике [4].
4. **Automatic Memory Dump** – содержит такую же информацию, как и в дампе памяти ядра, но Windows может более свободно выбирать размер файла подкачки.
5. **Active Memory Dump** – похож на полный дамп, но обычно меньше в размере, так как в него не включаются страницы памяти, которые скорее всего не помогут в диагностике неисправностей.

Наиболее интересен полный дамп памяти, так как он содержит наиболее полную информацию о системе и при помощи него наиболее вероятно отыскать и исправить какую-либо ошибку.

Как ранее было сказано, дамп файл состоит из заголовка и некоторого количества оперативной памяти. Так как исследуется возможность написания программы для создания дампов ОС Windows, запущенной под виртуальной машиной, то необходимо лишь узнать каким образом формировать заголовок, потому что при помощи виртуализационного ПО можно легко достать всю оперативную память гостя.

Формат заголовков дамп файлов ОС Windows, который понятен WinDbg является проприетарным и официально задокументирован довольно плохо (пропущено много полей структур). Но в интернете можно найти его полное описание (например, ресурсы [18], [19] и [20]).

Для 64-битной версии ОС Windows он выглядит следующим образом:

```
typedef struct _DUMP_HEADER64 {
	ULONG Signature;
	ULONG ValidDump;
	ULONG MajorVersion;
	ULONG MinorVersion;
	ULONG64 DirectoryTableBase;
	ULONG64 PfnDataBase;
	ULONG64 PsLoadedModuleList;
	ULONG64 PsActiveProcessHead;
	ULONG MachineImageType;
	ULONG NumberProcessors;
	ULONG BugCheckCode;
	ULONG __Padding1;
	ULONG64 BugCheckParameter1;
	ULONG64 BugCheckParameter2;
	ULONG64 BugCheckParameter3;
	ULONG64 BugCheckParameter4;
	UCHAR VersionUser[32];
	ULONG64 KdDebuggerDataBlock;
```

```
    union {
        PHYSICAL_MEMORY_DESCRIPTOR64 PhysicalMemoryBlock;
        UCHAR PhysicalMemoryBlockBuffer[704];
    };
    UCHAR ContextRecord[3000];
    MS_EXCEPTION_RECORD64 Exception;
    ULONG DumpType;
    ULONG __Padding2;
    ULONG64 RequiredDumpSpace;
    ULONG64 SystemTime;
    CHAR Comment[128];
    ULONG64 SystemUpTime;
    ULONG MiniDumpFields;
    ULONG SecondaryDataState;
    ULONG ProductType;
    ULONG SuiteMask;
    ULONG WriterStatus;
    UCHAR Unused1;
    UCHAR KdSecondaryVersion;
    UCHAR _reserved0[4018];
} DUMP_HEADER64, *PDUMP_HEADER64;
```

Также в интернете есть ресурсы (например, [5]), где описывается, что если вызвать функцию **KeInitializeCrashDumpHeader()** из драйвера, то можно получить большую часть заголовка, пригодную для конструирования дампа. Но, начиная с Windows 8 [21], нельзя просто прикрепить к этому заголовку гостевую память, поскольку лежащая по адресу **KdDebuggerDataBlock** из заголовка дампа структура **_KDDEBUGGER_DATA64** (для 64-битной архитектуры или **_KDDEBUGGER_DATA32** для 32-битной) лежит в памяти зашифрованной. А эта структура является очень важной, так как содержит множество информации о ядре ОС (KernBase – адрес начала ядерного кода, PaeEnabled – включен ли режим PAE и другие) и активно используется WinDbg при анализе дампа.

```
typedef struct _KDDEBUGGER_DATA64 {

        DBGKD_DEBUG_DATA_HEADER64 Header;
        ULONG64   KernBase;
        ULONG64   BreakpointWithStatus;
        ULONG64   SavedContext;

        USHORT  ThCallbackStack;
        USHORT  NextCallback;
        USHORT  FramePointer;

        USHORT  PaeEnabled;

        ULONG64   KiCallUserMode;

        ULONG64   KeUserCallbackDispatcher;

        ...

    } KDDEBUGGER_DATA64, *PKDDEBUGGER_DATA64;
```

Но, вместо расшифровки, его можно получить пользуясь недокументированной, но экспортируемой ядром Windows функцией **KeCapturePersistentThreadState()** [6], которую также надо вызывать из пространства ядра ОС.

Отсюда становиться ясно, что для получения полного заголовка нужно скомбинировать возвращаемые значения этих двух функций, но использовать их можно только в ядре операционной системы, и так как в силу закрытости исходного кода Windows дополнить/расширить само ядро этой функциональностью нельзя, то можно написать драйвер, который будет вызывать эти функции и некоторым образом передавать возвращаемую ими информацию в пространство пользователя (подробнее о методе передачи далее).

5.2 Подготовка окружения

Для написания и отладки драйверов под ОС Windows обычно используется конфигурация из двух компьютеров [7]. На одном из них ведется разработка драйвера, а на другом запуск и тестирование. Это делается из-за того, что некорректно написанный драйвер может привести к аварийному завершению операционной системы, которую потом будет сложно восстановить, что в свою очередь может привести к потере важных данных, в том числе исходных кодов разрабатываемого драйвера.

В качестве тестовой машины часто выступает виртуальная машина с запущенной ОС Windows [8], так как виртуализационное ПО (в том числе QEMU/KVM и Parallels Desktop for Mac) обычно предоставляет возможность снятия снэпшотов (от англ. snapshot – снимок, кадр; в данном контексте – снимок состояния) виртуальной машины и, таким образом, ускоряет процесс разработки и отладки из-за возможности в любой момент вернуться к предыдущему рабочему состоянию виртуальной машины или, наоборот, воспроизвести возникающую ошибку нужное количество раз для ее исправления.

По причине отсутствия второго компьютера на Windows, было решено разработку драйвера производить также на виртуальной машине. Этот подход имеет недостаток в более сложной настройке, но в замен предоставляет все преимущества, описанные выше.

WinDbg, через который будет производиться отладка драйверов, может подключаться к ОС Windows как установленной на физическом компьютере, так и внутри виртуальной машины.

Общая идея произведенной настройки окружения может быть проиллюстрирована следующей схемой:

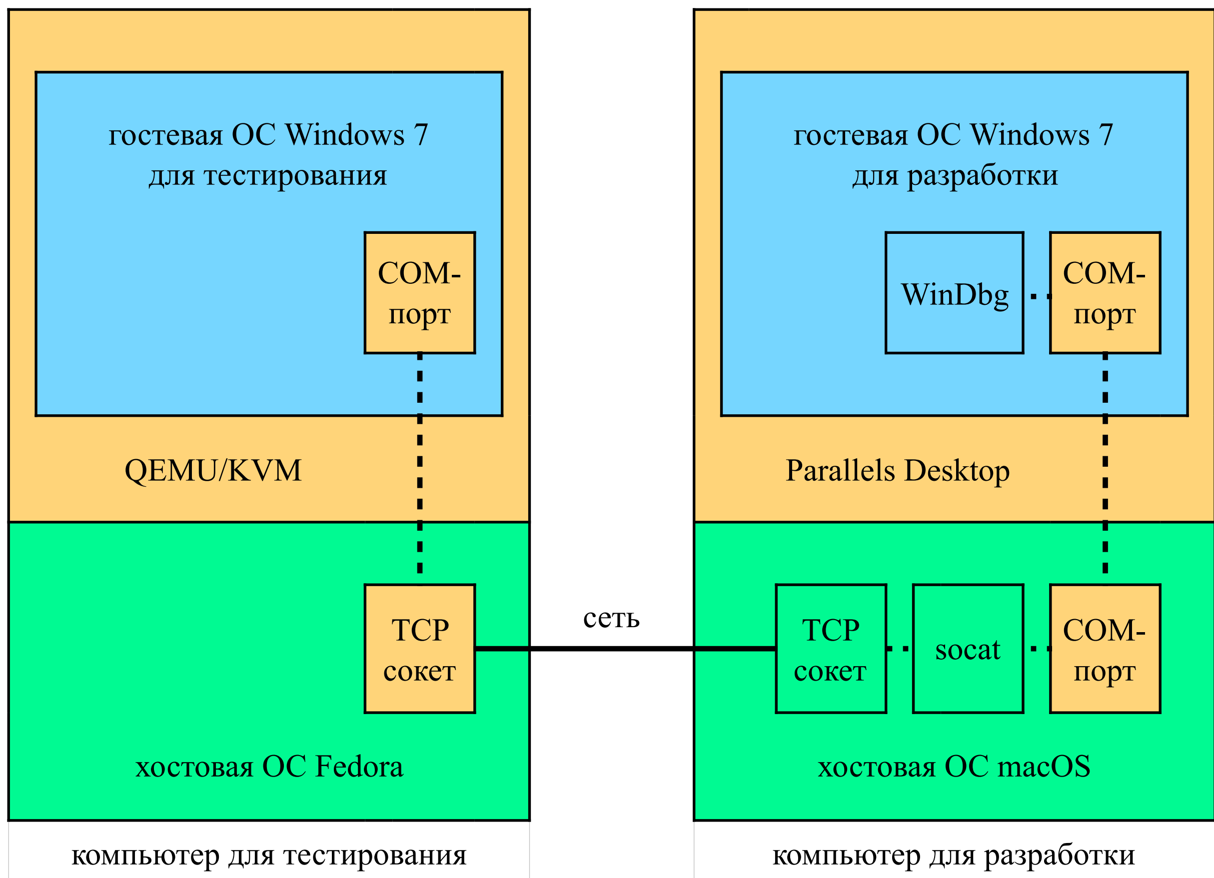

Рисунок 5.1. Схема настройки окружения для разработки и отладки драйверов

1. Настройка машины для разработки:

   1.1. На компьютер для разработки с операционной системой macOS было установлено виртуализационное ПО Parallels Desktop.

   1.2. Затем была создана виртуальная машина с ОС Windows 7. Была выбрана именно 7-ая версия операционной системы, так как во-первых, это самая новая версия ОС, на которой можно собирать драйвера не только при помощи Visual Studio, а еще и так называемым legacy (перев. с англ. – устаревший, унаследованный) путем, при помощи утилит командной строки из набора Driver Development Kit, что может быть полезно, поскольку в интернете существует большое количество хороших уроков по разработке драйверов под Windows, написанных достаточно давно, когда еще не было возможности собирать их используя Visual Studio (например, [8]). Во-вторых, это самая старая пользовательская

версия ОС Windows, на которой можно разрабатывать драйвера при помощи Visual Studio (что удобнее, чем делать это при помощи командной строки), и это важно, так как в каждой последующей версии ОС, минимальные системные требования, в том числе из-за усложнения и совершенствования графического интерфейса, увеличиваются, а скорость работы на том же аппаратном обеспечении – падает, и, если красота графики на процесс разработки и отладки влияет мало, то производительность системы, особенно запущенной в виртуальном окружении, крайне важна.

1.3. Так же были установлены все необходимые программы для написания и отладки драйверов: Visual Studio, WDK, WinDbg и другие.

2. Настройка тестовой машины:

2.1. На компьютер для тестирования была установлена последняя на момент начала исследования (начало 2017 года) версия дистрибутива ОС Linux – Fedora 25. Была выбрана именно она, поскольку компания Red Hat спонсирует разработку дистрибутива Fedora, так как тестирует в ней новые технологии, которые затем включает в дистрибутив собственной разработки Red Hat Enterprise Linux. Более того, Red Hat владеет компанией, которая начинала разрабатывать KVM, и сейчас на его основе также делает собственное виртуализационное решение Red Hat Virtualization. Из-за всего вышеперечисленного можно ожидать хорошую интегрированность решений и наличие качественной документации, что можно видеть, например в [9].

2.2. Так же были установлены все необходимые пакеты для работы связки QEMU/KVM.

2.3. Затем была создана виртуальная машина для тестирования с ОС Windows 7. Выбрана также 7-ая версия для единообразности настройки тестовой и рабочей операционных систем.

3. Настройка их взаимодействия:

3.1. Для Windows 7, WinDbg поддерживает отладку виртуальной машины через виртуальный COM-порт. Проблема состоит в том, что эти виртуальные машины находятся не на одной физической машине (поскольку производительности ни одного, ни другого ноутбука, использовавшегося в этой работе, не хватило бы для поддержания комфортной скорости работы двух виртуальных машин с ОС Windows). Поэтому на машине для разработки, через Parallels Desktop, был создан виртуальный COM-порт в режиме сервера, который затем при помощи утилиты socat был присоединен к TCP сокету [11], слушающему входящие подключения по локальной сети.

3.2. На тестовой машине настройка оказалась несколько проще, поскольку QEMU имеет возможность сразу расшаривать (от англ. share – делить, использовать совместно) COM-порт по сети через протокол TCP.

3.3. Теперь для отладки тестовой виртуальной машины через WinDbg, нужно запустить сначала виртуальную машину с ОС Windows на машине для разработки, затем запустить утилиту socat и, наконец, можно включать тестовую виртуальную машину, которая на этапе загрузки подсоединится через свой TCP сокет к сокету компьютера для разработки и, таким образом, «создаст» виртуальный кабель, соединяющий два виртуальных COM-порта.

5.3 Proof of concept

На основе примера от Microsoft [23], был написан драйвер, который вызывает функции KeInitializeCrashDumpHeader и KeCapturePersistentThreadState и сохраняет возвращаемую ими информацию в файлы на диске гостя: KeInitializeCrashDumpHeader.dmp и KeCapturePersistentThreadState.dmp соответственно.

1. KeInitializeCrashDumpHeader:

Возвращает хороший заголовок от полного дампа:

```
 1 00000000: 5041 4745 4455 3634 0f00 0000 b11d 0000  PAGEDU64........
 2 00000010: 00f0 d82e 0000 0000 0000 0000 80fa ffff  ................
 3 00000020: 907e 8902 00f8 ffff 909b 8702 00f8 ffff  .~..............
 4 00000030: 6486 0000 0200 0000 0000 0000 5041 4745  d...........PAGE
 5 00000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
 6 00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
 7 00000060: 5041 4745 5041 4745 5041 4745 5041 4745  PAGEPAGEPAGEPAGE
 8 00000070: 5041 4745 5041 4745 5041 4745 5041 4745  PAGEPAGEPAGEPAGE
 9 00000080: a030 8402 00f8 ffff 0200 0000 5041 4745  .0..........PAGE
10 00000090: 7eff 0300 0000 0000 0100 0000 0000 0000  ~...............
11 000000a0: 9e00 0000 0000 0000 0001 0000 0000 0000  ................
12 000000b0: e0fe 0300 0000 0000 5041 4745 5041 4745  ........PAGEPAGE
```

Рисунок 5.3.1. Начало заголовка дампа от KeInitializeCrashDumpHeader

Но, не содержит KdDebuggerDataBlock. Все это видно в WinDbg:

```
Command

Microsoft (R) Windows Debugger Version 10.0.14321.1024 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.


Loading Dump File [Y:\Documents\Programming\Virtuozzo\dumps\final\for screer
Kernel Complete Dump File: Full address space is available

************************************************************
WARNING: Dump file has been truncated.  Data may be missing.
************************************************************
Error: Change all symbol paths attempts to access 'c:\Programming\Drivers\sy
Error: Change all symbol paths attempts to access 'c:\Programming\Symbols\lc

************* Symbol Path validation summary **************
Response                         Time (ms)     Location
Error                            15            c:\Programming\Symbols\local_
Error                            15            c:\Programming\Drivers\symbol
Deferred                                       SRV*c:\Programming\Symbols\we
Symbol search path is: c:\Programming\Symbols\local_symbols;c:\Programming\I
Executable search path is:
***************************************************************************
THIS DUMP FILE IS PARTIALLY CORRUPT.
KdDebuggerDataBlock is not present or unreadable.
***************************************************************************
```

Рисунок 5.3.2. Заголовок от KeInitializeCrashDumpHeader загружен в WinDbg

2. KeCapturePersistentThreadState:

Возвращает заголовок от минидампа, поэтому его нельзя взять полностью для создания полного дампа, но содержит расшифрованный KdDebuggerDataBlock. Это видно и в WinDbg (поскольку отсутствует ошибка, изображенная на рисунке 5.3.2):

```
Command

Microsoft (R) Windows Debugger Version 10.0.14321.1024 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.


Loading Dump File [Y:\Documents\Programming\Virtuozzo\dumps\final\for screens\wi
Mini Kernel Dump File: Only registers and stack trace are available

Error: Change all symbol paths attempts to access 'c:\Programming\Drivers\symbols
Error: Change all symbol paths attempts to access 'c:\Programming\Symbols\local_s

************* Symbol Path validation summary **************
Response                         Time (ms)     Location
Error                                          c:\Programming\Symbols\local_symbo
Error                                          c:\Programming\Drivers\symbols
Deferred                                       SRV*c:\Programming\Symbols\websyml
Symbol search path is: c:\Programming\Symbols\local_symbols;c:\Programming\Drive
Executable search path is:
Windows 7 Kernel Version 7601 (Service Pack 1) MP (2 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 7601.17514.amd64fre.win7sp1_rtm.101119-1850
Machine Name:
Kernel base = 0xfffff800`02652000 PsLoadedModuleList = 0xfffff800`02897e90
Debug session time: Sat Jun 24 21:15:11.052 2017 (UTC + 3:00)
System Uptime: 0 days 0:21:27.922
Loading Kernel Symbols
...............................................................
...............................................................
........
Loading User Symbols
Mini Kernel Dump does not contain unloaded driver list
```

Рисунок 5.3.3. Дамп от KeCapturePersistentThreadState загружен в WinDbg

Также это видно в самом файле по сигнатуре KDBG:

```
521 00002080: 70f4 8502 00f8 ffff 70f4 8502 00f8 ffff  p.......p.......
522 00002090: 4b44 4247 4003 0000 0020 6502 00f8 ffff  KDBG@.... e.....
523 000020a0: 90a4 6c02 00f8 ffff 0000 0000 0000 0000  ..l.............
524 000020b0: e801 d800 0000 0100 109c 6c02 00f8 ffff  ..........l.....
525 000020c0: 0000 0000 0000 0000 907e 8902 00f8 ffff  .........~......
526 000020d0: 909b 8702 00f8 ffff c89b 8702 00f8 ffff  ................
527 000020e0: d0f7 8602 00f8 ffff c0fd 8602 00f8 ffff  ................
528 000020f0: 7030 9002 00f8 ffff 8430 9002 00f8 ffff  p0.......0......
529 00002100: 80ab 8c02 00f8 ffff 004a 8c02 00f8 ffff  .........J......
530 00002110: 90f1 8c02 00f8 ffff d0a7 8702 00f8 ffff  ................
531 00002120: e8a7 8702 00f8 ffff 0000 0000 0000 0000  ................
532 00002130: 0000 0000 0000 0000 c06d 8502 00f8 ffff  .........m......
533 00002140: 2832 9002 00f8 ffff 0000 0000 0000 0000  (2..............
```

Рисунок 5.3.4. Начало KdDebuggerDataBlock из дампа KeCapturePersistentThreadState

и совпадающим с заголовком дампа значениям PsLoadedModuleList (0xfffff800`02897e90) и PsActiveProcessHead (0xfffff800`02879b90):

```
1 00000000: 5041 4745 4455 3634 0f00 0000 b11d 0000  PAGEDU64........
2 00000010: 00f0 d82e 0000 0000 0000 0000 80fa ffff  ................
3 00000020: 907e 8902 00f8 ffff 909b 8702 00f8 ffff  .~..............
4 00000030: 6486 0000 0200 0000 e200 0000 5041 4745  d...........PAGE
5 00000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
6 00000050: 0000 0000 0000 0000 0000 0000 80f8 ffff  ................
7 00000060: 5041 4745 5041 4745 5041 4745 5041 4745  PAGEPAGEPAGEPAGE
8 00000070: 5041 4745 5041 4745 5041 4745 5041 4745  PAGEPAGEPAGEPAGE
9 00000080: a030 8402 00f8 ffff 5041 4745 5041 4745  .0......PAGEPAGE
```

Рисунок 5.3.5. Начало заголовка дампа от KeCapturePersistentThreadState

3. Теперь, подключив к тестовой машине в режиме отладки ядра WinDbg, исполняется команда .crash, которая вызывает аварийное завершение системы [24]. После перезагрузки, на диске тестовой системы сохраняется полный дамп памяти MEMORY.DMP (для этого нужна предварительная настройка [25]). Который без ошибок загружается в WinDbg:

```
Command

Microsoft (R) Windows Debugger Version 10.0.14321.1024 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.


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

Error: Change all symbol paths attempts to access 'c:\Programming\Drivers\symbol
Error: Change all symbol paths attempts to access 'c:\Programming\Symbols\local_

************* Symbol Path validation summary **************
Response                         Time (ms)     Location
Error                                          c:\Programming\Symbols\local_symt
Error                                          c:\Programming\Drivers\symbols
Deferred                                       SRV*c:\Programming\Symbols\websym
Symbol search path is: c:\Programming\Symbols\local_symbols;c:\Programming\Drive
Executable search path is:
Windows 7 Kernel Version 7601 (Service Pack 1) MP (2 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 7601.17514.amd64fre.win7sp1_rtm.101119-1850
Machine Name:
Kernel base = 0xfffff800`02652000 PsLoadedModuleList = 0xfffff800`02897e90
Debug session time: Sat Jun 24 21:15:40.612 2017 (UTC + 3:00)
System Uptime: 0 days 0:21:57.482
Loading Kernel Symbols
...............................................................
................................................................
........
Loading User Symbols

Loading unloaded module list
.....
```

Рисунок 5.3.6. Полный дамп, созданный Windows в результате аварийного завершения

4. В качестве proof of concept, при помощи консольной утилиты dd родной заголовок MEMORY.DMP заменяется на заголовок от KeInitializeCrashDumpHeader (в нем было исправлено значение поля DirectoryTableBase на системное, которое было в оригинальном MEMORY.DMP; эта ручная правка не является проблемой, поскольку гипервизору доступно значение регистра cr3 (в нем хранится адрес первой таблицы страничного преобразования текущего процесса, т.е. значение DirectoryTableBase) всех процессов гостевой системы и оно может быть скопировано в заголовок либо таким образом, либо внутри драйвера можно использовать функцию KeStackAttachProcess для того, чтобы подсоединится к любому процессу, после чего можно считать его значение cr3), а KdDebuggerDataBlock на взятый из дампа, возвращаемого KeCapturePersistentThreadState. В результате получившийся дамп MEMORY_my.DMP загружается в WinDbg и показывает результат без ошибок, как и оригинальный MEMORY.DMP:

```
Command

Microsoft (R) Windows Debugger Version 10.0.14321.1024 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.


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

Error: Change all symbol paths attempts to access 'c:\Programming\Symbols\local_
Error: Change all symbol paths attempts to access 'c:\Programming\Drivers\symbol

************* Symbol Path validation summary **************
Response                         Time (ms)     Location
Error                                          c:\Programming\Symbols\local_symb
Error                                          c:\Programming\Drivers\symbols
Deferred                                       SRV*c:\Programming\Symbols\websym
Symbol search path is: c:\Programming\Symbols\local_symbols;c:\Programming\Drive
Executable search path is:
Windows 7 Kernel Version 7601 (Service Pack 1) MP (2 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 7601.17514.amd64fre.win7sp1_rtm.101119-1850
Machine Name:
Kernel base = 0xfffff800`02652000 PsLoadedModuleList = 0xfffff800`02897e90
Debug session time: Sat Jun 24 21:15:11.052 2017 (UTC + 3:00)
System Uptime: 0 days 0:21:27.922
Loading Kernel Symbols
...............................................................
................................................................
........
Loading User Symbols

Loading unloaded module list
.....
```

Рисунок 5.3.7. Полный дамп, с заголовком от KeInitializeCrashDumpHeader и KdDebuggerDataBlock от KeCapturePersistentThreadState

Также можно заметить, что относительно оригинального MEMORY.DMP с рисунка 5.3.6, поменялось значение Debug System time (было: Sat Jun 24 21:15:40.612 2017 (UTC + 3:00), стало: Sat Jun 24 21:15:11.052 2017 (UTC + 3:00)) и System Uptime (было: 0 days 0:21:57.482, стало: 0 days 0:21:27.922) на более раннее, такое же, как у заголовка дампа от KeInitializeCrashDumpHeader:

```
*************************************************************************
THIS DUMP FILE IS PARTIALLY CORRUPT.
KdDebuggerDataBlock is not present or unreadable.
*************************************************************************
KdDebuggerData.KernBase < SystemRangeStart
Windows 7 Kernel Version 7601 MP (2 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Machine Name:
Kernel base = 0x00000000`00000000 PsLoadedModuleList = 0xfffff800`02897e90
Debug session time: Sat Jun 24 21:15:11.052 2017 (UTC + 3:00)
System Uptime: 0 days 0:21:27.922
*************************************************************************
```

Рисунок 5.3.8. Заголовок дампа от KeCapturePersistentThreadState в WinDbg

5. Таким образом, идея полностью подтверждена на практике.
6. Поиск адреса KdDebuggerDataBlock в файле дампа:

6.1. Поиск адреса в файле полного дампа, куда нужно вставить KdDebuggerDataBlock в случае, рассматриваемом в этой работе (в полном дампе системы KdDebuggerDataBlock расшифрован), происходил путем взятия значений PsLoadedModuleList и PsActiveProcessHead из заголовка дампа и их поиска по файлу дампа, поскольку эти два поля в таком же порядке содержатся в KdDebuggerDataBlock (рисунок 5.3.4).

6.2. Но, в готовом программном решении предполагается брать память живой гостевой системы, в которой этот блок может быть зашифрован и тогда предыдущий способ не пройдет. Нужно воспользоваться тем, что в заголовке от KeInitializeCrashDumpHeader есть поле DirectoryTableBase, которое содержит значение регистра cr3, в котором хранится адрес первой таблицы страничного преобразования. Пример преобразования виртуальных (в нашем случае тоже самое, что и линейных) адресов в физические приведен на рисунке 5.3.6.2.1. Весь адрес разбивается

на части, где каждая часть служит смещением в соответствующей таблице страничного преобразования. По этому смещению лежит адрес следующей по иерархии таблицы. В самой последней таблице лежит адрес физической страницы, а последняя часть линейного адреса является смещением в этой странице.

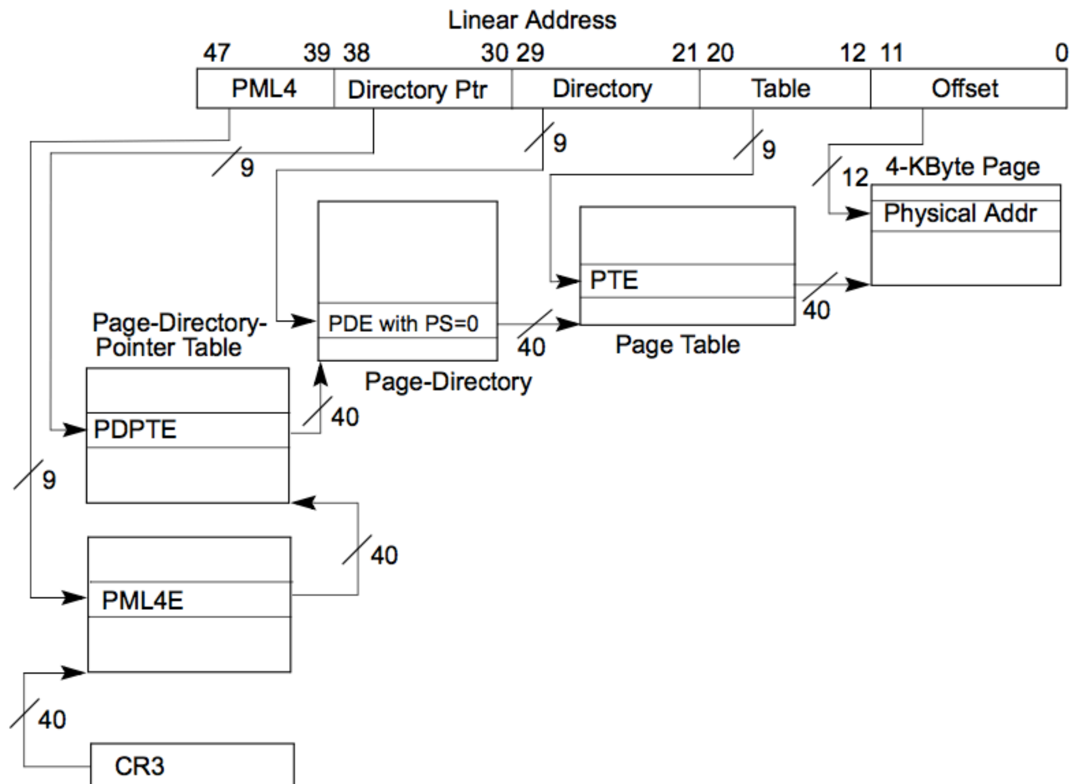

Рисунок 5.3.6.2.1. Преобразование линейных адресов в физические [26]

Мы получили из виртуального адреса физический. Теперь необходимо понять, как из физического адреса в памяти гостя получить смещение в файле дампа. Обратившись, например к [18], становится ясно, что нужно обратить внимание на поле PhysicalMemoryBlock из заголовка, возвращаемого функцией KeInitializeCrashDumpHeader, которое является структурой PHYSICAL_MEMORY_DESCRIPTOR64 (рассматриваем 64-битную версию ОС):

```
typedef struct _PHYSICAL_MEMORY_DESCRIPTOR64 {
    ULONG64 NumberOfRuns;                    // 0x88
    ULONG64 NumberOfPages;                   // 0x90
    PHYSICAL_MEMORY_RUN64 Run[2];            // 0x98
} PHYSICAL_MEMORY_DESCRIPTOR64,
*PPHYSICAL_MEMORY_DESCRIPTOR64;

typedef struct _PHYSICAL_MEMORY_RUN64 {
    ULONG64 BasePage;            // 0x98
    ULONG64 PageCount;           // 0xa0, …
} PHYSICAL_MEMORY_RUN64,
*PPHYSICAL_MEMORY_RUN64;
```

В заголовке дампа из KeInitializeCrashDumpHeader эти структуры отвечают полям начиная с адреса 0x88:

```
 9 00000080: a030 8402 00f8 ffff 0200 0000 5041 4745  .0..........PAGE
10 00000090: 7eff 0300 0000 0000 0100 0000 0000 0000  ~...............
11 000000a0: 9e00 0000 0000 0000 0001 0000 0000 0000  ................
12 000000b0: e0fe 0300 0000 0000 5041 4745 5041 4745  ........PAGEPAGE
```

Рисунок 5.3.6.2.2. Заголовок дампа от KeCapturePersistentThreadState

Также помогает утилита dumpchk из набора Microsoft Debugging Tools, проанализировав которой дамп можно получить следующую информацию:

```
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`3fee0000
Last Page: 00000000`3ff7e000     00000000`3ffdf000
```

Рисунок 5.3.6.2.3. Анализ заголовка от KeCapturePersistentThreadState в dumpchk

NumberOfRuns ограничено 2, NumberOfPages 0x3ff7e, а поскольку каждая страница выровнена на 4КБ (0x1000), то адрес последней страницы 0x3ff7e000.

Далее: BasePage из первого Run это 0x1, значит ее адрес 0x1000, PageCount = 0x9e, значит длина страниц 0x9e000, а следующее смещение в файле:

FileOffset2 = FileOffset1 + Length1 = 0x1000 + 0x9e000 = 0x 9f000.

Далее аналогично для всех Run.

6.3. Таким образом, понятно, как посчитать смещение в файле полного дампа, куда нужно записать KdDebuggerDataBlock.

5.4 Архитектура ПО

На основе проведенных исследований и вышеизложенных фактов, предлагается следующая модель программного решения для достижения поставленной цели:

1. На основе файла:

   1.1. Гостевой драйвер делает видимым в пространстве пользователя гостя файл, при доступе к которому происходит исполнение описанных ранее функций KeInitializeCrashDumpHeader и KeCapturePersistentThreadState.

   1.2. Хост при помощи виртуализационного ПО читает из гостя этот файл, т.е. получает части заголовка дампа.

   1.3. Преимущество этого метода в том, что уже есть рабочий прототип этого решения. Недостаток: для работы этого прототипа предполагается расширение функциональности виртуализационного ПО (об этом далее).

2. На основе дополнительной виртуальной очереди:

   2.1. В реализации гостевого драйвера добавить virtio очередь.

   2.2. При старте системы, во время загрузки драйвера, вызывать функции KeInitializeCrashDumpHeader и KeCapturePersistentThreadState и передавать при помощи этой очереди информацию для заголовка в хост.

   2.3. Преимущество этого метода в том, что информация для заголовка будет получена на раннем этапе загрузки системы, что крайне полезно, поскольку иногда ОС аварийно завершается раньше инициализации пространства пользователя и, в таком случае, формирование дампа первым способом является невозможным. Недостаток в том, что придется писать/изменять как front-end, так и back-end virtio драйвер, что может занять большее количество времени, чем реализация первого способа.

3. Затем, когда понадобится создать дамп, происходит копирование оперативной памяти гостя и формирование верного заголовка из информации полученной в шаге 1 или 2.

Теперь, когда ясна высокоуровневая структура решения, нужно обратить внимание на конкретные детали реализации:

1. Существует драйвер PVpanic [12], который реализует часть функциональности описанной в выше в шаге 1. Но он использует только функцию KeInitializeCrashDumpHeader, а, следовательно, получает информацию не про весь заголовок. Т.е. его можно взять за основу, но нужно добавить использование функции KeCapturePersistentThreadState.
2. Автор драйвера реализовал доступ к файлу через операцию ввода-вывода со специальным кодом IOCTL_GET_CRASH_DUMP_HEADER. Но, шаг 2 предлагается делать при помощи QEMU Guest Agent, который на текущий момент (середина 2017 года) не поддерживает операции ввода-вывода с произвольными кодами. Автор PVpanic пытался добавить эту функциональность в исходный код QEMU Guest Agent [13], но сообщество не приняло этот патч.
   Далее цитата из письма с отказом одно из мейнтейнеров (от англ. maintain – обслуживать, содержать в исправности) QEMU [22]:
   … you need to know precise operating system details before you can use the API. I think that's really flawed.

   It would be possible to design a ioctl API that is more usable if you didn't try to do generic passthrough of arbitrary ioctl commands. Instead ... provide strongly typed arguments for the ioctl commands you care about using. Or completely separate ... commands instead of multiplexing everything into an ioctl command.

вольный перевод:

... перед тем, как использовать предложенное вами API (API – Application Programming Interface – программный интерфейс приложения, прим. перев.) нужно в точности узнать характеристики [гостевой] операционной системы. На мой взгляд это существенный недостаток.

Можно спроектировать гораздо более удобное API для операций ввода-вывода (ioctl), если не пытаться делать обобщенную передачу произвольной ioctl операции. Вместо этого лучше предоставить четко типизированные аргументы для той ioctl операции, которую вам надо использовать. Или вообще разделить операции [QEMU Guest Agent'а] вместо того, чтобы объединять все в одну ioctl операцию.

Поэтому нужно либо дописать драйвер таким образом, чтобы к файлу, который он делает видимым, можно было обращаться при помощи обычных операций чтения и записи, доступных через QEMU Guest Agent, либо прислушаться к замечаниям мейнтейнеров QEMU и предложить расширение функциональности QEMU Guest Agent, которое бы удовлетворило обе стороны.

3. Получить оперативную память гостя можно, при помощи QEMU Guest Agent, а именно функции под названием dump-guest-memory. Но проблема в том, что она возвращает память не в том формате, который понятен WinDbg. Сконвертировать в нужный формат можно либо при помощи утилиты из фреймворка Volatility [14], либо можно написать программу конвертации самостоятельно взяв Volatility за основу, так как у нее открыт исходный код.

4. Также необходимо помнить, что заголовок дампа нужно создавать заново при любом изменении размера оперативной памяти гостя. Это можно сделать при помощи механизма асинхронных уведомлений

драйвера (Asynchronous Driver Notification, [31]). Для этого необходимо в драйвере, при помощи функции IoRegisterPlugPlayNotification [33] с параметром GUID_DEVICE_MEMORY, зарегистрировать функцию-callback определенного вида, которая будет вызвана после того, как драйвер получит уведомление от операционной системы, о том, что был изменен объем оперативной памяти. Этот callback будет заново запускать процесс создания заголовка.

6. Результаты

В результате этой работы была исследована и доказана осуществимость идеи создания отладочных дампов гостевой ОС Windows, в процессе которой:

- Изучен стандартный формат дампа ОС Windows
- Найдены функции, предоставляющие поля заголовка дампа
- Подготовлено окружение для написания и отладки ПО
- Разработана архитектура готового программного решения для создания дампов в автоматическом режиме
- Написаны свои и также найдены примеры программ в открытом доступе, подтверждающие реализуемость упомянутых подходов.

7. Дальнейшее развитие

На основе разработанной архитектуры и написанных proof of concept программ написать готовое решение, позволяющее в произвольный момент времени создавать отладочные дампы памяти гостевой ОС Windows в автоматическом режиме.

Также одним из следующих шагов может быть внедрение написанного ПО в существующее виртуализационное программное обеспечение QEMU/KVM.

8. Список литературы и ссылки на источники

1. A step by step guide for linux kvm virtualization on embedded systems [электронный ресурс]: http://www.virtualopensystems.com/en/solutions/guides/kvm-on-arm/
2. Virtio: An I/O virtualization framework for Linux [электронный ресурс]: https://www.ibm.com/developerworks/library/l-virtio/index.html
3. Varieties of Kernel-Mode Dump Files [электронный ресурс]: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/varieties-of-kernel-mode-dump-files
4. Small Memory Dump [электронный ресурс]: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/small-memory-dump
5. Описание формата Dmp [электронный ресурс]: https://forum.antichat.ru/threads/48533/
6. KeCapturePersistentThreadState() и crash dump'ы [электронный ресурс]: https://vxlab.info/wasm/print.php-article=kcpts.htm
7. Пишем свой первый Windows-драйвер [электронный ресурс]: https://habrahabr.ru/post/40466/
8. Windows driver debugging with WinDbg and VMWare [электронный ресурс]: https://briolidz.wordpress.com/2012/03/28/windows-driver-debugging-with-windbg-and-vmware/
9. Getting started with virtualization [электронный ресурс]: https://fedoraproject.org/wiki/Getting_started_with_virtualization
10. Serial Communication between VMs on different Computers [электронный ресурс]: http://web.fe.up.pt/~pfs/aulas/lcom2015/proj/ser_multi_pcs.html
11. Building a Very, Very Long Serial Cable [электронный ресурс]: http://www.digitalbond.com/blog/2011/12/28/building-a-very-very-long-serial-cable/

12. Github: kvm-guest-drivers-windows/pvpanic/ [электронный ресурс]: https://github.com/virtio-win/kvm-guest-drivers-windows/tree/master/pvpanic

13. [Qemu-devel] [PATCH] qga: implement guest-file-ioctl [электронный ресурс]: http://lists.gnu.org/archive/html/qemu-devel/2017-02/msg00035.html

14. Extracting Windows VM crash dumps [электронный ресурс]: https://ladipro.wordpress.com/2017/01/06/extracting-windows-vm-crash-dumps/

15. Виртуализация [электронный ресурс]: https://ru.wikipedia.org/wiki/Виртуализация

16. Паравиртуализация [электронный ресурс]: https://ru.wikipedia.org/wiki/Паравиртуализация

17. virtio: Towards a De-Facto Standard For Virtual I/O Devices [электронный ресурс]: https://ozlabs.org/~rusty/virtio-spec/virtio-paper.pdf

18. DMP File Structure [электронный ресурс]: http://computer.forensikblog.de/en/2006/03/dmp-file-structure.html

19. 64bit Crash Dumps [электронный ресурс]: http://computer.forensikblog.de/en/2008/02/64bit-crash-dumps.html

20. Microsoft Research Singularity [электронный ресурс]: https://singularity.svn.codeplex.com/svn/base/Windows/Inc/Dump.h

21. The Secret to 64-bit Windows 8 and 2012 Raw Memory Dump Forensics [электронный ресурс]: https://volatility-labs.blogspot.ru/2014/01/the-secret-to-64-bit-windows-8-and-2012.html

22. Re: [Qemu-devel] [PATCH] qga: implement guest-file-ioctl [электронный ресурс]: https://lists.gnu.org/archive/html/qemu-devel/2017-02/msg00046.html

23. Non-PnP Driver Sample [электронный ресурс]: https://github.com/Microsoft/Windows-driver-samples/tree/master/general/ioctl/kmdf

24. Forcing a System Crash from the Debugger [электронный ресурс]: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/forcing-a-system-crash-from-the-debugger

25. How to generate a kernel or a complete memory dump file [электронный ресурс]: https://support.microsoft.com/en-us/help/969028/how-to-generate-a-kernel-or-a-complete-memory-dump-file-in-windows-server-2008-and-windows-server-2008-r2

26. Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 3A.

27. Server Virtualization and OS Trends [электронный ресурс]: https://community.spiceworks.com/networking/articles/2462-server-virtualization-and-os-trends

28. KVM [электронный ресурс]: https://ru.wikipedia.org/wiki/KVM

29. Сравнение виртуальных машин [электронный ресурс]: https://ru.wikipedia.org/wiki/Сравнение_виртуальных_машин

30. Paravirtualized drivers for kvm/Linux [электронный ресурс]: http://www.linux-kvm.org/page/Virtio

31. Introduction to Driver Notification [электронный ресурс]: https://msdn.microsoft.com/en-us/library/windows/hardware/ff548029(v=vs.85).aspx

32. Registering for Asynchronous Driver Notification [электронный ресурс]: https://msdn.microsoft.com/en-us/library/windows/hardware/ff560880(v=vs.85).aspx

33. IoRegisterPlugPlayNotification [электронный ресурс]: https://msdn.microsoft.com/en-us/library/windows/hardware/ff549526(v=vs.85).aspx