Об анализе алгоритмов приложений с использованием драйверов в операционных системах Windows 7-10

В.Е. Копейцев
kopeicev@kasperskyclub.com

А.Б. Веретенников
alexander@veretennikov.ru

Кафедра вычислительной математики и компьютерных наук УрФУ (Екатеринбург)

Аннотация

В данной статье идёт речь о создании системы для мониторинга активности приложений, в частности, отслеживания операций работы с файловой системой, системным реестром и сетевыми устройствами. При этом описывается реализация данной системы с применением современной платформы Windows Driver Foundation, позволяющей создаваемым драйверам режима ядра работать в ОС Windows 7-10. Был исследован вопрос, позволяют ли новые подходы к проектированию и разработке драйверов ОС Windows создать универсальную систему мониторинга, решающую широкий круг задач, начиная от тестирования и анализа модели ввода-вывода приложения, заканчивая анализом вредоносного программного обеспечения. В документе приводится описание архитектуры системы, применяемых методов решения поставленных задач. Приведено описание возникающих проблем, в том числе не описанных ранее, а также алгоритмов их решения, некоторые из которых были модифицированы и усовершенствованы для применения в данной задаче.

1 Введение

1.1 Об исследовании поведения приложений

Задача анализа поведения приложения и, соответственно, алгоритма, по которому оно работает, является актуальной в разных областях компьютерных наук: автоматическая классификация моделей ввода-вывода, а также анализ и последующая оптимизация ввода-вывода [1, 2, 3], системы предотвращения вторжений [4], обнаружение и анализ вредоносного программного обеспечения [5]. Зачастую решать данную задачу приходится в условиях отсутствия исходного кода программы. Конечно же, чаще всего данная задача решается в рамках обеспечения информационной безопасности электронных систем, когда требуется проанализировать действия, которые производятся программой, чтобы определить, имеет ли приложение деструктивную активность. Также, без решения данной задачи практически невозможно определить, какой ущерб был нанесён системе вредоносным программным обеспечением, ведь не всегда результаты деструктивной активности хорошо заметны, а анализ алгоритма приложения в большинстве случаев позволяет установить весь функционал исследуемой программы.

In: A.A. Makhnev, S.F. Pravdin (eds.): Proceedings of the International Youth School-conference «SoProMat-2017», Yekaterinburg, Russia, 06-Feb-2017, published at http://ceur-ws.org

Эту же задачу необходимо решить для оценки нагрузки на систему [1], которая создаётся в результате работы приложения (или группы приложений). В частности, решение данной задачи полезно при использовании так называемых твердотельных накопителей SSD (solid-state drive) для прогнозирования выхода оборудования из строя. Другим примером использования анализа алгоритмов приложений может служить вычисление нагрузки, создаваемой на каналы связи, в частности, на канал подключения к сети интернет. Генерируя различные наборы данных для используемых приложений, мы можем наблюдать, какую нагрузку на канал связи они создают. Таким образом, становится возможно вычислять, какая пропускная способность сетевого канала необходима для решения поставленных задач.

Наконец, с помощью отслеживания действий конкретного приложения, можно определить модель ввода-вывода этого приложения. Нет ли периодов, на протяжении которых ввод-вывод существенно отличается от других периодов. Соответствует ли ввод-вывод программы тем алгоритмам, которые в ней заложены (аудит системы на наличие ошибок в реализации). Можно сравнивать разные программы, решающие одну и ту же задачу. В некоторых случаях, даже сами разработчики приложений бывают заинтересованы в определении модели ввода-вывода [1], так как при создании программы могут быть использованы сторонние средства ввода-вывода, реализованные во вспомогательных библиотеках, либо механизмы, предоставляемые операционной системой, таким образом, у разработчиков приложения нет доступа к исходному коду продуктов, которые они используют.

1.2 Методы анализа поведения приложения

Как известно, существует несколько подходов к анализу поведения приложения. Традиционно их подразделяют на статические и динамические [6]. Статические методы, например, дизассемблирование, декомпиляция, эмуляция, имеют ряд недостатков. Существуют различные методы затруднения проведения статического анализа: упаковка исполняемых файлов, шифрование, динамическое получение адресов импортируемых функций, обфускация кода и т.д. Использование перечисленных методов приводит к тому, что даже опытному специалисту очень тяжело изучать ассемблерный код программы и это требует значительного количества времени, а декомпиляторы не могут восстановить оригинальный исходный код. Обфускация программы, приводящая к появлению большого количества инструкций, и использование недокументированных команд или нестандартных параметров функций API операционной системы затрудняют эмуляцию.

Именно эти проблемы и заставляют исследователей применять как статический, так и динамический анализ, отладку и мониторинг [7]. Но и для проведения отладки приложения требуется большое количество времени и усилий специалиста. Поскольку мониторинг позволяет запоминать последовательности действий, выполненных программой в системе, он лишён перечисленных недостатков и сводит задачу анализа алгоритма приложения к задаче анализа собранных данных (логов).

Существуют различные способы реализации мониторинга. Первый способ заключается в сравнении двух снимков состояния системы «до» и «после» запуска программы. Такой подход имеет ряд плюсов, в частности сравнительную простоту реализации приложения. Однако имеются и недостатки, а именно: невозможно определить хронологию изменений, если вредоносная программа не производит операций записи, а выполняет только операции чтения, например, с целью похищения данных. Тогда такие, безусловно важные, части функционала приложения зафиксированы не будут. Другим достаточно очевидным подходом является реализация аппаратного монитора, например, SCSI или IDE аппаратные анализаторы [2], позволяющие перехватывать операции файлового ввода-вывода. Третий подход заключается в использовании утилит, позволяющих смонтировать каталоги, находящиеся на удалённом сервере, как локальные. При обращении к данным каталогам все запросы исследуемого приложения будут перенаправлены на другое устройство, что сводит задачу к перехвату и анализу сетевого трафика [3, 8]. Но, к сожалению, для использования такой модели необходимо заранее знать список файлов и каталогов, к которым обращается программа, что не укладывается в условия решаемых задач. Ещё один недостаток методов анализа с помощью перехвата с использованием аппаратных анализаторов и перехвата сетевой активности состоит в том, что не ясно, как отделить модель ввода-вывода приложения от работы кэша операционной системы.

Отметим, что в ОС семейства Linux предусмотрены механизмы перехвата вызовов API операционной системы, которые позволяют решить поставленные задачи [9]. По умолчанию в Windows подобные механизмы отсутствуют, однако их можно реализовать, например, использовав возможность создания специальных компонентов ядра ОС – драйверов, которые перехватывают и логируют запросы к оборудованию, сделанные приложением, в момент их обработки операционной системой. Именно эта методика позволяет

обрабатывать как операции чтения, так и записи, при этом возможно наиболее точно фиксировать время обработки события и устанавливать хронологическую последовательность действий приложения. Реализация драйверов, которые будут описаны далее, работает до кэша, что позволяет исключить влияние работы кэша на результаты экспериментов. Исходя из вышесказанного, можно заключить, что метод мониторинга с реализацией в виде драйверов позволяет решить все описанные задачи. Именно этот подход используется при практической реализации систем обнаружения и предотвращения вторжений [4].

2 Архитектура системы

2.1 Мониторинг файловой активности

Первой задачей, которую требуется решить, является задача мониторинга дисковых операций, так как работа исследуемого приложения с файлами и директориями важна для решения задач информационной безопасности, тестирования, изучения нагрузки на систему и т.д. С введением технологии Windows Driver Foundation между драйвером файловой системы и менеджером ввода-вывода появилось ещё одно промежуточное звено – менеджер фильтров (Filter Manager). Также появился новый тип драйвера – минифильтр файловой системы, который подключается к стеку драйверов через менеджер фильтров.

Важно заметить, что некоторым фильтрующим драйверам важно их положение в стеке драйверов относительно других минифильтров. Например, если драйвер антивируса будет размещён «ниже» по стеку, т.е. ближе к физическому устройству, чем драйвер, осуществляющий резервное копирование, может произойти ситуация, когда драйвер антивируса детектирует и удаляет вредоносное программное обеспечение, а копия вредоносного файла уже попала в резервную копию, так как до драйвера резервного копирования запрос дошёл раньше. Именно минифильтр файловой системы был выбран для реализации мониторинга файлового ввода-вывода.

2.2 Мониторинг сетевой активности

Второй задачей, которая имеет не меньшую важность, чем мониторинг дисковых операций, является мониторинг сетевой активности. Для реализации была выбрана технология Windows Filtering Platform (WFP), которая, как и File System Minifilter, появилась в составе фреймворка Windows Driver Foundation. WFP позволяет создавать драйверы, работающие с подключениями и пересылаемыми пакетами на транспортом уровне модели OSI. Основное назначение таких драйверов – выполнение функций сетевого экрана (Firewall), т.е. по различным признакам (или правилам) разрешать или запрещать сетевую активность. WFP, как и File System Minifilter, не встраивает создаваемые на этой платформе драйвера непосредственно в стек устройства. Вместо этого в стек драйверов встраивается модуль, поставляемый вместе с ОС и называемый Filter Engine. Задачей Filter Engine, в свою очередь, является вызов функций обработки сетевой активности (callouts), которые и выполняют фильтрацию.

2.3 Мониторинг реестра

На данном этапе двух имеющихся компонентов уже достаточно для сбора информации при решении задач анализа нагрузки, создаваемой приложениями на дисковые устройства и сетевые каналы связи. Однако, исследование показало, что этих модулей недостаточно для анализа вредоносного программного обеспечения. Известно, что значительные изменения в работу системы Windows могут быть внесены через записи системного реестра, которые хранят настройки системы, а также сторонних приложений. Системный реестр хранится на диске в виде нескольких файлов, поэтому логично предположить, что работа с реестром влечёт за собой запись изменений в файлы и видна монитору файловой активности. Однако, данная активность не видна драйверу File System Minifilter. Решение описанной проблемы было найдено в виде включения в систему третьего драйвера, отвечающего за мониторинг операций с системным реестром по средствам взаимодействия с компонентом ядра ОС, называемым Configuration Manager. Данный компонент даёт возможность регистрировать функции обработки операций с системным реестром, что позволяет решить поставленную задачу минимальным набором функций.

Следующей задачей, которую требовалось решить, является обработка информации, собранной драйверами. Поскольку требуется обеспечить взаимодействие создаваемой системы с произвольными модулями анализа собранных данных, было принято решение записывать всю собранную информацию в виде текстовых лог-файлов в формате CSV. Таким образом, задача анализа результатов работы системы сводится

к задаче анализа текстовых логов заданного формата. Также, для обработки команд необходимо взаимодействовать с пользователем. Очевидно, что для выполнения всех эти требований необходимо создание приложения пользовательского режима. Конечно же, при таком решении возникает задача коммуникации между драйверами и приложением, её решение будет рассмотрено позже. Общая модель архитектуры создаваемой системы представлена на рис. 1.

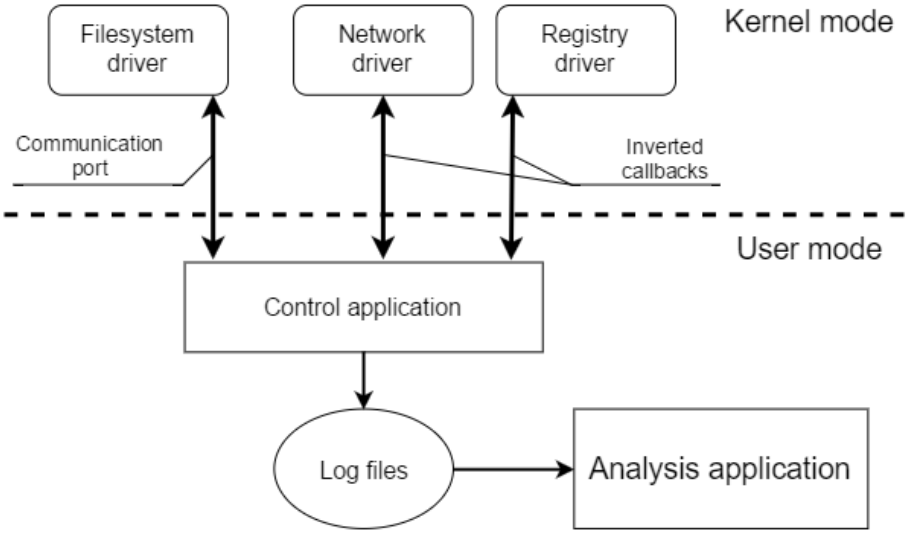

Рис. 1: Архитектура создаваемого решения

3 Реализация драйвера мониторинга файловой системы

Реализация драйверов класса File System Minifilter уже была описана в различной литературе и документации компании Microsoft, поэтому остановимся лишь на особенностях применения данного класса драйверов к задаче сбора данных о работе приложений.

Первой задачей, которую требовалось решить, является сбор всех необходимых данных, а именно: ID процесса-инициатора операции и ID его текущего потока, объект файловой системы, к которому производится обращение и информация о производимой операции. С информацией о производимой операции дело обстоит проще всего, все необходимые данные передаются в обработчик IRP драйвера менеджером фильтров в структуре PFLT_CALLBACK_DATA. Получение ID процесса и потока тоже производится достаточно тривиально, в уже упомянутой структуре обработчику передаётся поле Thread – объект потока-инициатора операции. Последовательное использование системных вызовов IoThreadToProcess, PsGetProcessId и PsGetThreadId позволяет получить объект процесса-инициатора операции, его ID и ID нужного потока, соответственно.

Сложнее обстоит дело с получением информации об объекте файловой системы, к которому обращена операция. Применение системного вызова FltGetFileNameInformation позволяет получить путь к файлу относительно корневого каталога устройства, на котором он хранится, но не определить, к какому из устройств хранения информации, подключённых к системе, этот путь относится. В обработчик IRP передаётся имя устройства, к которому обращена операция, однако данное имя имеет вид "HardDriveX", где X – номер носителя. Конечно же, такое имя устройства не подходит для отображения в результатах работы системы и необходимо сопоставить каждому такому имени букву носителя, которая используется для обозначения устройства в графическом интерфейсе ОС. Единственным путём решения данной задачи, который был найден во время исследования, является получение GUID устройства из кода драйвера режима ядра при помощи системного вызова FltGetVolumeGuidName. Полученный GUID может быть передан в управляющее приложение пользовательского режима, после чего, при помощи вызова WIN32 API GetVolumePathNamesForVolumeNameW, преобразован к нужному виду. Заметим, что функция GetVolumePathNamesForVolumeNameW может возвращать несколько имён-алиасов, однако, поскольку все они указывают на одно устройство, может быть использовано любое из них.

Важно отметить, что прямая реализация данного метода значительно замедляет работу драйвера режима ядра, поэтому следует подвергнуть её некоторой оптимизации. Тестирование показало, что значи-

тельную часть времени занимает работа функции FltGetVolumeGuidName, поэтому реализация с вызовом данной функции при обработке каждого IRP снижает производительность системы. Было решено воспользоваться одной из особенностей архитектуры драйверов класса File System Minifilter, а именно вынести указанный функционал в функцию-обработчик, вызываемую единожды, при установке фильтра в стек устройства. Данная функция позволяет определить собственную структуру так называемого контекста устройства, в этой структуре и сохраняется полученный GUID. При помощи системного вызова FltAllocateContext производится выделение памяти для хранения контекста устройства, а его сохранение производится при помощи функции FltSetInstanceContext. Затем, при обработке IRP, понадобится лишь вызвать функцию FltGetInstanceContext, которая вернёт сохранённый контекст устройства, содержащий его GUID.

Во время создания драйвера мониторинга файловой активности был реализован функционал как обработки операций одного выбранного процесса, так и всех процессов, запущенных в системе, при этом была выявлена проблема, до этого не описанная в публично доступной литературе. Каждая попытка произвести мониторинг всех запущенных процессов приводила к полному зависанию всей системы и необходимости перезапуска компьютера. Рассмотрим следующую ситуацию: некий процесс делает операцию ввода-вывода, драйвер перехватывает её и передаёт информацию приложению, приложение пытается записать информацию в лог-файл и этим действием порождает ещё одну операцию ввода-вывода, её, как и предыдущую, перехватывает драйвер, он снова пытается передать информацию приложению, однако, приложение ожидает завершения своей операции ввода-вывода и не может принять сообщение от драйвера. Таким образом, между драйвером и приложением возникает косвенная рекурсия (indirect recursion), которая может привести к взаимоблокировке обоих компонентов системы, так как драйвер блокируется в ожидании завершения операции передачи данных через коммуникационный порт. Очевидно, что блокировка приложения пользовательского режима не угрожает общей функциональности системы, но блокировка драйвера режима ядра ожидаемо приводит к зависанию всей операционной системы. Схема возникновения подобной ситуации изображена на рис. 2.

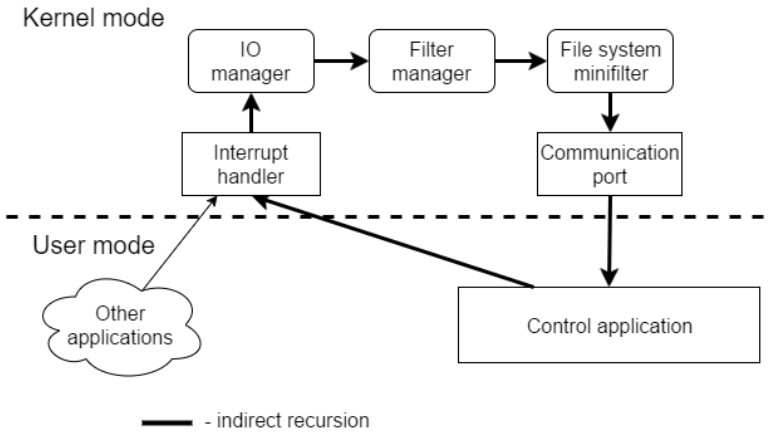

Рис. 2: Схематичное изображение возникновения косвенной рекурсии в случае мониторинга всех процессов

Создание на стороне приложения нескольких потоков-производителей, которые будут принимать сообщения из драйвера и ставить их в некую очередь, и одного потока-потребителя, который будет осуществлять запись в лог, улучшит ситуацию, так как на несколько операций ввода-вывода будет создаваться всего одна новая операция записи в лог. Однако, многопоточность полностью не исключит проблему, а лишь уменьшит вероятность её возникновения, ведь может возникнуть ситуация, когда очередь сообщений в приложении полностью заполнена, а поток-потребитель заблокирован в операции вывода в файл. Единственным корректным способом решения проблемы является добавление в исключения процесса приложения, работающего с драйвером, тем более, что это не вредит функциональности продукта: вряд ли пользователю интересно видеть в логе операции записи в этот лог. Остаётся лишь определить, какой из процессов, работающих в системе, добавлять в исключение. Как уже было упомянуто, для коммуникации между описываемым драйвером режима ядра и приложением пользовательского режима используется коммуникационный порт. При создании коммуникационного порта и вызове функции FltCreateCommunicationPort ей передаётся функция-обработчик подключений к создаваемому порту. Известно, что во время обработ-

ки системных вызовов драйвер хоть и работает в режиме ядра, однако одновременно с этим он работает в контексте процесса-инициатора вызова. Вторым полезным фактом является знание того, что подключение к коммуникационному порту приложением пользовательского режима производится через вызов одной из функций WIN32 API. Таким образом, функция-обработчик подключений, заданная драйвером, вызывается в контексте процесса, который подключается к порту. Это позволяет, используя системные вызовы PsGetCurrentProcess и PsGetProcessId, получить ID процесса утилиты, который требуется добавить в исключения.

4 Реализация драйверов мониторинга сетевой активности и работы с реестром

Как уже было сказано ранее, мониторинг сетевой активности реализуется в виде драйвера режима ядра, построенного согласно технологии WFP, входящей в платформу разработки драйверов WDF. Согласно архитектуре WFP, фильтрафия может осуществляться на различных уровнях и этапах сетевого обмена данными, например, на этапах установки подключения или передачи данных. Как и в случае с мониторингом файловой активности, по каждой интересующей нас операции требуется собрать различные данные, которые мы можем получить, осуществляя фильтрацию на транспортном уровне согласно модели OSI: IP адреса отправителя и получателя, порты отправителя и получателя, протокол транспортного уровня и направление трафика (входящее, либо исходящее), ID процесса и потока, инициировавших операцию. Конечно же, в большинстве случаев такие данные необходимы далеко не по всем событиям, проходящим через фильтр, ведь мы можем исследовать активность одного процесса из многих, работающих в системе. Поэтому, если мы будем собирать данные по всем событиям, это существенно замедлит работу системы.

Для решения описанной проблемы в обработчиках фильтра производится лишь начальная проверка, подходит ли операция под установленные параметры фильтрации. Если подходит, она изымается из потока сетевых данных и помещается в очередь, где ожидает обработки. Если не подходит, то драйвер больше её не задерживает и пропускает дальше по стеку драйверов. Использование очереди событий позволяет организовать отложенную обработку операций в отдельном потоке, также работающем в режиме ядра. Поэтому обработка выбранных событий не оказывает существенного влияния на производительность всей системы.

Для реализации драйвера мониторинга операций с системным реестром были использованы системные вызовы, работающие с компонентом ядра ОС, называемым Configuration Manager. Данные системные вызовы позволяют установить функцию-обработчик, которая будет перехватывать операции работы с системным реестром. Поскольку для получения необходимой информации были использованы методы, уже описанные в данной статье применительно к другим драйверам, детали их реализации мы опустим.

5 Решение задачи коммуникации драйверов режима ядра и приложения

Изначально Windows Filtering Platform задумывалась в качестве набора средств для фильтрации сетевых данных согласно заданным правилам, поэтому данная платформа не подразумевает наличия готового API для взаимодействия с приложениями пользовательского режима, такого как, например, Communication Port в File System Minifilter. Аналогичную ситуацию мы можем наблюдать и при работе с Configuration Manager. Поэтому в ходе разработки системы возникла необходимость создания способа взаимодействия данных драйверов с приложением пользовательского режима. Очевидно, что передать данные из драйвера режима ядра можно многими способами, ведь его код обладает неограниченными привилегиями. Можно записывать данные в файлы, системный реестр, отправлять их по сети, но все эти способы были отвергнуты на этапе исследования, так как не обеспечивали достаточного быстродействия. Другим способом является запись данных напрямую в память процесса приложения пользовательского режима. Конечно, такой способ позволяет достичь максимальной производительности, однако является недостаточно безопасным: любые ошибки при записи в память другого процесса могут привести к аварийному завершению работы приложения. Для решения описанной проблемы необходимо посмотреть на задачу коммуникации драйвера и приложения под другим углом, ведь приложение само может инициировать передачу данных через интерфейс отправки команд устройству, при помощи функции WIN32 API DeviceIOControl. Известно, что DeviceIOControl позволяет отправлять три типа команд, описание которых приведено в таблице 1.

Если выделить участок памяти в приложении пользовательского режима и указать его в качестве буфера команды, имеющей тип METHOD_BUFFERED, можно передать адрес этого участка памяти в драйвер режима ядра. Затем драйвер сможет записать данные в указанный буфер и пометить команду устройству

Таблица 1: Типы методов, предоставляемые функцией DeviceIOControl

Тип команд (методов)	Возможность доступа к общей памяти
METHOD_BUFFERED	Из драйвера и из приложения
METHOD_IN_DIRECT METHOD_OUT_DIRECT	Только из драйвера или только из приложения
METHOD_NEITHER	Отсутствует

как выполненную, после чего приложение сможет работать с этими данными. Таким образом, появляется ещё один способ передать сообщение из драйвера к приложению. Однако, если использовать эту схему синхронно, мы получим очень низкую производительность, так как приложение будет бездействовать во время ожидания каждого ответа от драйвера. Заметим, что открытие устройства по символической ссылке происходит так же, как и открытие обычного файла, через функцию WIN32API CreateFile, которая позволяет использовать асинхронный ввод-вывод. Асинхронный ввод-вывод позволяет существенно увеличить быстродействие, поскольку, отправив команду устройству, приложение сразу же может отправлять следующую команду, не ожидая завершения обработки предыдущей. Когда команда устройству будет выполнена, будет вызвана функция обратного вызова (callback), находящаяся в коде приложения. Описанный метод известен среди специалистов как inverted callbacks (метод отражённых функций обратного вызова) и применяется для решения задач коммуникации драйверов и приложений достаточно давно. Как уже было сказано, применение inverted callbacks позволяет теоретически решить проблему недостаточного быстродействия системы, однако при реализации данной техники возникает ряд вопросов. Должно ли приложение постоянно генерировать команды устройству с максимально возможной скоростью? Если да, то в случае малого количества событий, перехватываемых драйвером, приложение будет создавать избыточное потребление памяти. Если нет, то приложение должно останавливать генерацию команд устройству, тогда возникают другие вопросы, связанные с тем, успеет ли приложение снова начать отправку событий в нужный момент и обеспечить должную производительность.

Очевидно, что нагрузка на систему и, соответственно, количество операций в единицу времени может почти мгновенно возрастать или снижаться в разы и, поскольку это зависит от действий пользователя, такие изменения невозможно предсказать. Примером такой ситуации является момент начала работы пользователя после длительного простоя системы или запуск приложения, требующего загрузки значительного объёма данных. Это означает, что для эффективной работы приложению необходимо реагировать на изменяющуюся нагрузку и генерировать различное количество команд устройству в разные моменты времени.

В ходе исследования возможных путей повышения производительности и оптимизации работы системы было замечено, что выполнение команды устройству может быть приостановлено драйвером на неопределённый срок, а затем возобновлено в нужный момент. Это означает, что приложение может заранее сгенерировать значительное количество команд, а драйвер может обработать их асинхронно тогда, когда ему это будет необходимо и появится достаточное количество событий. Для повышения производительности системы был применён достаточно простой, но эффективный алгоритм. Для предотвращения чрезмерного потребления памяти приложению устанавливается число, ограничивающее максимальное количество одновременно существующих команд устройству, назовём его max_requests. После запуска приложение генерирует команды устройству в количестве, равном max_requests, после чего переходит в режим ожидания. Драйвер сохраняет команды в очередь и обрабатывает их, когда появляются события, требующие отправки информации приложению. Приложение подсчитывает количество запросов, обработка которых завершена. Когда их количество достигает значения, кратного (max_requests)/2, приложение генерирует команды, пока их число снова не достигнет max_requests. Заметим, что данная задача и применённый метод являются частным случаем решения задачи производителя и потребителя. Таким образом, метод inverted callbacks был доработан при помощи создания очереди запросов в ядре и управления количеством команд в приложении. Для ясности будем называть второй метод inverted callbacks with queue. Применение данных изменений позволило получить прирост производительности, что выражается в исчезновении проблемы исчерпания памяти из-за накопления событий, ожидающих обработки. Общая схема работы метода inverted callbacks with queue приведена на рис. 3.

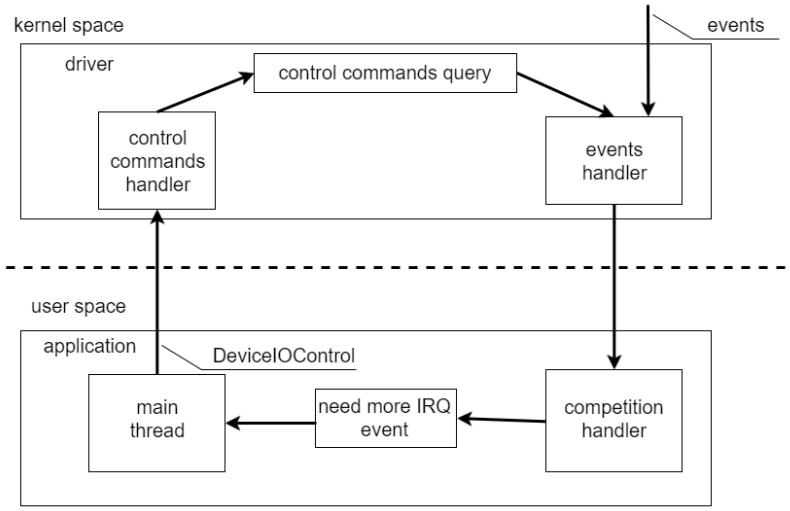

Рис. 3: Общая схема работы метода inverted callbacks with queue

6 Реализация приложения для взаимодействия с драйверами

Остановимся на нескольких моментах реализации приложения пользовательского режима, осуществляющего управление системой, взаимодействие с пользователем и запись результатов в лог файлы. Поскольку по умолчанию процесс не может получить доступ к процессам других приложений, так как это обусловлено необходимостью обеспечения безопасности, необходимо выполнить ряд действий, чтобы получить информацию о процессах, работающих в системе. Частично решить проблему может запуск программы с правами администратора, однако системные процессы всё равно окажутся недоступными. Функция WIN32API AdjustTokenPrivileges позволяет приложению получить отладочные привилегии (SeDebugPrivilege), которые позволяют работать с большинством других процессов. Также нужно помнить, что приложения, скомпилированные под x86, в системах архитектуры x64 работают под виртуальной машиной, называемой WOW64 (Windows On Windows 64), и не могут получить доступ к процессам приложений x64, запущенных на реальной системе. Для решения этой проблемы необходимо компилировать приложение под платформу x64, если предполагается использование программы в системах с такой разрядностью. И даже с учётом всех этих моментов, существует два исключения. Первое – это процесс с ID 0, бездействие системы (Idle), второе – экземпляры процесса csrss.exe, подсистемы пользовательского режима Win32. Данные процессы, формально, видны из пользовательского режима, но на самом деле работают в режиме ядра, о чём явно говорят разработчики ОС, поэтому получить к ним доступ из приложения пользовательского режима невозможно.

Отметим также, что приложение позволяет запускать произвольный исполняемый файл, указанный в настройках. Запуск производится при помощи системного вызова CreateProcess с установленным флагом CREATE_SUSPENDED, что позволяет инициализировать систему и получить корректные результаты наблюдений. Дело в том, что по умолчанию запущенный процесс немедленно начинает исполнение, в то же время необходимо передать настройки мониторинга драйвера, что занимает некоторое время. Передать настройки заранее не представляется возможным, так как до запуска исследуемого приложения не известен ID, который будет присвоен его процессу. Возникает ситуация, когда часть результатов может быть потеряна. Именно поэтому сначала приложение запускается в приостановленном (SUSPENDED) режиме, затем настройки передаются драйверам, после чего производится запуск исполнения исследуемой программы.

Что касается общей архитектуры описываемого приложения, основной поток программы производит запуск трёх потоков, каждый из которых работает с одним из драйверов, так достигается независимость обработки событий разных типов. Если при работе с драйвером используется метод inverted callbacks with queue, также в отдельном потоке вызывается функция-обработчик ответов на асинхронные запросы приложения. Общая схема архитектуры приложения для взаимодействия с драйверами изображена на рис. 4.

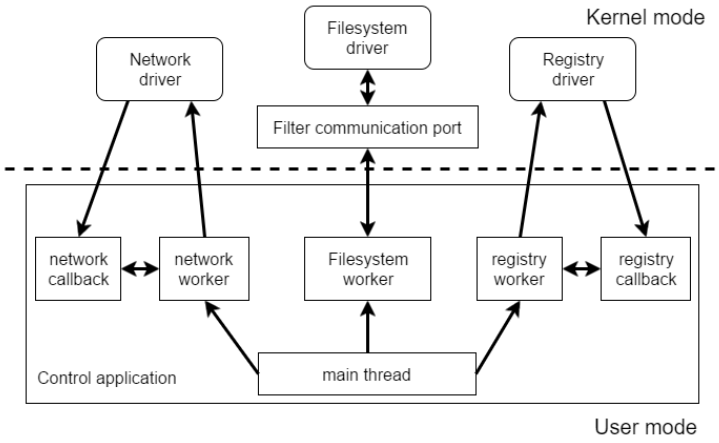

Рис. 4: Общая схема архитектуры приложения для взаимодействия с драйверами

7 Результаты проведённых экспериментов

После создания описанной системы было проведено несколько экспериментов. Рассмотрим полученные результаты. Часть экспериментов связана с анализом вредоносного программного обеспечения. Следует заметить, что, если программа имеет деструктивный функционал, запуск её на реальной системе может повлечь негативные последствия. Однако, если исследуемая программа уже работает в системе, подобный мониторинг ситуацию не ухудшит, а только поможет её прояснить. В других случаях можно использовать специально подготовленную виртуальную машину, но она может быть обнаружена исследуемым программным обеспечением. В данной работе мы не будем касаться вопросов защиты от обнаружения виртуальных машин, так как это слишком опосредованно относится к рассматриваемому материалу.

7.1 Анализ вредоносного программного обеспечения Trojan-PSW.Win32.Tepfer.mqgh

В первом эксперименте проводился анализ образца вредоносной программы, которая по классификации «Лаборатории Касперского» детектируется как «Trojan-PSW.Win32.Tepfer.mqgh». Данная вредоносная программа представляет собой исполняемый файл Windows (PE-EXE файл). Файл имеет размер 115712 байт и контрольную сумму, посчитанную по алгоритму MD5 A36E46551F0ECE7D8808D1BBF1CC8314. Мониторинг активности с файловой системой сразу позволил выявить главный функционал данной вредоносной программы – кражу аутентификационных данных, сохранённых в различных приложениях, например, FTP-клиентах, браузерах, почтовых клиентах. На рис. 5 изображён фрагмент результатов анализа лога мониторинга файловой активности, на котором видно, что вредоносная программа производит попытки открытия стандартных директорий установки различных FTP-клиентов. Таким образом выполняется задача сбора информации о существующих директориях и, соответственно, установленном программном обеспечении. Аналогичную активность можно наблюдать и на мониторинге системного реестра, например, по попыткам открытия ключей различных приложений.

Кроме того, мониторинг сетевой активности зафиксировал множество подключений (на что указывает количество задействованных портов отправителя) к ресурсу, имеющему IP адрес 50.63.202.32. Согласно имеющимся данным, в частности, опубликованным анализам данного вредоносного программного обеспечения и информации сервисов reverse DNS, указанный ресурс содержит вредоносное ПО, а также принимает украденные данные. Полученные данные приведены в таблице 2.

Таблица 2: Фрагмент результатов мониторинга сетевых операций эксперимента 1

Период времени	Отправитель	Получатель	Порты отправителя	Порты получателя	Протокол
18.1.2017 18:36:44:797 18.1.2017 19:7:9:438	10.0.2.15	50.63.202.32	49177, 49178, 49179, 49180, 49181, 49182, 49183, 49184, 49185	80	TCP

Рис. 5: Фрагмент результатов мониторинга файлового ввода-вывода эксперимента 1

7.2 Сравнительный анализ алгоритмов работы приложений Far и Hiew

Перейдём к рассмотрению второго эксперимента. В нём был проведён сравнительный анализ двух приложений, позволяющих выполнять поиск фрагмента текста в текстовом файле, Far 3.0.4545 x64 и Hiew 8.45. Обоим приложениям ставилась одна и та же задача: найти первое вхождение заданной последовательности символов в заданном тексте. Поскольку абсолютные значения экспериментов не важны, так как проводился сравнительный анализ результатов, выполненных на одной и той же машине, её конфигурацию можно опустить. Используемый текстовый файл имел размер 1 092 494 851 байт, кодировку UTF-8, выполнялся поиск первого вхождения последовательности символов «FIND PATTERN», которое находилось по смещению 546 933 089 байт от начала файла.

До проведения эксперимента мы ожидали увидеть последовательное чтение участков файла для проведения поиска подстроки, однако результаты эксперимента показали неочевидные подробности работы алгоритмов исследуемых программ. В частности, чтение участков файла выполняется с некоторым «перекрытием», т.е. следующий участок файла начинается не с конца предыдущего, а с некоторой точки, находящейся внутри предыдущего участка. Иначе говоря, если мы обозначим за n размер считываемого блока, а за s обозначим смещение начала некоторого блока, то, согласно результатам эксперимента, для обоих исследуемых приложений будет выполняться следующее соотношение:

$$s_i - s_{i-1} < n.$$

Более подробное рассмотрение полученных результатов показало, что для приложения Far размер считываемого блока константен и $n = 1280000$ байт, а разница в стартовых смещениях двух соседних блоков всегда составляет 640000 байт, т.е. $(1/2)n$. Для приложения Hiew размер считываемого блока также является постоянным, $n = 262144$, а разница в стартовых смещениях двух соседних блоков всегда составляет 262038 байт. Таким образом, в Hiew «перекрытие» составляет всего 106 байт, что значительно меньше, чем 640000 байт у Far.

Точно ответить на вопрос, для чего используется такая техника, конечно же, могут лишь разработчики приложений, но в ходе исследования была высказана гипотеза, что «перекрытие» используется для обработки случаев, когда искомый фрагмент текста оказался на границе прочитанного участка. Обратите внимание: полученные данные показывают, что, поскольку Far читает большими блоками и с большим

«перекрытием», в большинстве случаев для выполнения поставленной задачи данному приложению потребуется прочитать и просканировать больший объём данных. Напротив, Hiew читает меньшими блоками, соответственно, как правило, ему требуется проанализировать меньший объём данных, но сделать больше операций чтения. Можно заключить, что эти приложения используют разную стратегию обработки файлов: Far минимизирует количество операций чтения, но замедляет обработку данных в памяти, а Hiew использует меньший объём памяти, но делает больше операций чтения.

Данный эксперимент позволяет ответить на вопрос, какая стратегия выгоднее в данном случае, т.е. какой способ позволяет найти первое вхождение заданного фрагмента текста быстрее. По результатам эксперимента видно, что Hiew справился с задачей быстрее, выполнив поиск за 7 секунд, против 11 секунд у Far. Можно сказать, что во всех случаях, когда не важно количество обращения к носителю информации, эффективнее показало себя приложение Hiew. Также отметим, что само наличие чтения с «перекрытием» неоптимально, поскольку обработку нахождения искомого фрагмента на границе буфера можно осуществлять более эффективно.

8 Выводы

Разработана система для мониторинга активности приложений в части ввода-вывода, сетевой активности, обращений к реестру. Дан положительный ответ на вопрос, возможна ли реализация подобной системы с использованием технологии Windows Driver Foundation. Разработаны эффективные алгоритмы взаимодействия управляющего клиентского приложения с драйверами. Представлены результаты экспериментов: анализ активности вредносного приложения, сравнительный анализ алгоритмов поиска в тексте для Far и Hiew, давший неожиданный результат. Разработанные методы могут применяться для анализа активности вредоносных программ, в системах обнаружения и предотвращения вторжений, при анализе алгоритма работы приложений в целях оптимизации и проверки корректности.

Список литературы

[1] T. M. Madhyastha. Automatic classification of input/output access patterns. Doctoral Dissertation. *University of Illinois*, 153, 1997.

[2] A. Riska, E. Riedel. Disk drive level workload characterization. *In USENIX Annual Conference*, 9–9, 2006.

[3] D. J. Ellard. Trace-based analyses and optimizations for network storage servers. PhD thesis. *Harvard University, Cambridge*, 218, 2004.

[4] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 222–232, 1987.

[5] I. Santos, J. Devesa, F. Brezo, J. Nieves, P. G. Bringas. OPEM A Static-Dynamic Approach for Mailware Detection. *International Joint Conference CISIS'12-ICEUTE'12-SOCO'12 Special Sessions*, 271–280, 2013.

[6] IEEE Standard Glossary of Software Engineering Terminology. *ANSI/IEEE Std.*, 610, 1990.

[7] E. Skoudis, L. Zeltser. Malware: Fighting Malicious Code. *Prentice Hall PTR.*, 672, 2003.

[8] D. J. Ellard, J. Ledlie, P. Malkani, M. Seltzer. Passive NFS Tracing of Email and Research Workloads. *FAST '03 Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 203–216, 2003.

[9] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. E. Long, T. T. McLarty. File System Workload Analysis for Large Scale Scientific Computing Applications. *In proceedings of the 21st IEEE / 12th NASA Goddard conference on mass storage systems and technologies*, 139–152, 2004.

On the analysis of application algorithms using drivers in Windows 7-10 operating systems

Vyacheslav E. Kopeytsev, Alexander B. Veretennikov

Ural Federal University (Yekaterinburg, Russia)

Keywords: I/O traces, file systems, application analysis, operating systems, drivers, computer science

This article about creation of system for application monitoring using Windows Driver Foundation. It describes tracing of operations with file system, network and registry, that can help to solve a wide range of tasks, from testing and analysis of input-output models to malware analysis. The document describes the architecture of this system, the methods used to solve the tasks. Also, given examples of problems, including those not previously described, as well as their solutions, some of which have been modified and improved for application in this task.