Автономная некоммерческая организация высшего образования
«Университет Иннополис»
(АНО ВО «Университет Иннополис»)

АННОТАЦИЯ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
(БАКАЛАВРСКУЮ РАБОТУ)
ПО НАПРАВЛЕНИЮ ПОДГОТОВКИ
09.03.01 – «ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА»

НАПРАВЛЕННОСТЬ (ПРОФИЛЬ) ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ «ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА»

Тема **UEFI драйвер для анализа активности диска**

Выполнил **Лямец Михаил Андреевич** подпись

Иннополис, Innopolis, 2021

Оглавление

Аннотация

Предыдущие исследования о **внешнем логировании** [1] были сосредоточены на методах виртуализации: мониторинг гостевых машин из гипервизора [2]. Данное исследование попыталось расширить эту тему, показав, что такое логирование может происходить из UEFI [3] путем создания драйвера, который отслеживает дисковую активность UEFI. Это актуально, поскольку UEFI обеспечивает лучшую производительность и масштабируемость.

С другой стороны, мы также затронули тему **обнаружение основного наборов файлов** [4]. Этот набор существует, что было доказано исследователями в прошлом. Однако вычисление такого набора на машинах конечных пользователей никогда не было удобным. Мы показали, что вычисление этого набора может быть быстрым и масштабируемым, реализовав его в UEFI в меньшем масштабе: набор необходимых файлов для фазы загрузки ОС.

Целью данного исследования была проверка гипотезы о том, что мониторинг дисковой активности UEFI возможен посредством простых **хуков протоколов UEFI**. Кроме того, мы хотели помочь Acronis Active

[1] Внешнее логировании - мониторинг операционной системы извне

[2] Гипервизор - программное решение, которое запускает виртуальные машины

[3] Extensible Firmware Interface – предоставляет программные интерфейсы для программного обеспечения и служит гибкой средой программирования

[4] Обнаружение набора основного файлов - поиск файлов, которые абсолютно необходимы для загрузки ОС

Restore [5] в генерации необходимого набора файлов - коллекции файлов, которая необходима для загрузки системы.

Мы выбрали UEFI в качестве базовой среды, поскольку она более гибкая и производительная, чем BIOS, а также ее драйверы легко развертываются и поддерживаются. Из всех протоколов UEFI мы решили использовать Block IO, поскольку он наиболее близок к микропрограмме. Мы подключили хуки в Block IO протокол, собрали данные о доступе к диску, сбросили их в файловую систему, а затем перевели в человекочитаемый формат в пользовательском пространстве операционной системы.

Таким образом, мы доказали, что мониторинг дисковой активности в UEFI возможен с помощью хуков. Наше решение не требовало серьезных накладных расходов и было легко развертываемым. Мы собрали необходимый набор файлов для Active Restore и проложили путь в область внешнего логирования операционной системы из UEFI.

Мониторинг ОС из UEFI - перспективная тема. Мы создали пригодный для использования прототип и показали, что он может быть широко распространен. Во многом благодаря среде UEFI и ее поддержке среди OEM[6] (Original Equipment Manufacturer). Однако не стоит останавливаться на достигнутом, ведь это прекрасный шанс продвинуться дальше в области внешнего логирования и создать драйвер, который сможет исследовать любую ОС из среды UEFI.

[5] Активное восстановление - решение, ускоряющее загрузку системы за счет "ленивой"загрузки файлов

[6] OEM – производитель компонентов

Глава 1

Введение

Существует область криминалистики, которая требует особой тщательности при работе с данными. Она требует от аналитиков уверенности в том, что источник заслуживает доверия. Например, если операционная система (ОС) скомпрометирована вредоносным ПО, можно ли верить данным, полученным из ее журналов событий? Скорее всего, нет. По-видимому, эту проблему можно решить только путем переноса логики логирования событий за пределы ОС. Это сделает ее недоступной для вредоносных программ.

Ведение логов за пределами операционных систем долгое время оставалось сложной темой. Ситуация изменилась для многих после появления мощных средств виртуализации. Несмотря на удобство виртуальных машин, некоторые обстоятельства не способствуют их использованию, например, домашний компьютер конечного пользователя. Именно здесь в игру вступает UEFI[1].

Среда UEFI была создана как средство преодоления некоторых недо-

[1]Unified Extensible Firmware Interface - спецификация, определяющая среду до операционной системы

статков своего предшественника, устаревшей системы BIOS[2]. BIOS предоставлял возможности предварительной настройки системы в течение многих лет, прежде чем сообщество осознало нужно в более мощном решении. Разработчики хотели исследовать внутреннее устройство системы до запуска ОС, во время её загрузки и во время выполнения. Они также предпочитали писать на языке высокого уровня, а не на ассемблере BIOS. UEFI решил эту проблему, сделав язык C частью своего стандарта и улучшив интеграцию между до-ОС средой и самой ОС [1].

1.1 Acronis Active Restore

Проблема обнаружения необходимого набора файлов возникла несколько лет назад в компании Acronis[3] в рамках проекта под названием Active Restore[4] [2]. Решение этой задачи позволит ответить на следующий вопрос. Как эффективно вычислить те файлы, которые абсолютно необходимы для успешного запуска системы? Частично он был решен Кимом (Kim)[3], который доказал, что такой набор существует. К сожалению, их работа не отвечала критериям удобства и эффективности, оставаясь в основном теоретическим подходом к проблеме.

1.2 Цели и вклад

Наша цель - улучшить результаты работы Кима и предоставить удобную структуру, которая будет столь же точной в получении набора файлов,

[2]Basic I/O System

[3]Acronis International GmbH разрабатывает локальное и облачное программное обеспечение для резервного копирования и аварийного восстановления

[4]Acronis Active Restore - это запатентованная технология, которая возвращает систему в рабочее состояние сразу после начала восстановления части диска

но при этом быстрой и простой в обращении. Это поможет команде Acronis создавать частичные резервные копии, т.е. резервные копии, сохраняющие только необходимые файлы, что позволит сэкономить значительные накладные расходы. В частности, мы хотим, чтобы наше решение отвечало следующим требованиям:

1. Решение должно быть развертываемо на машине конечного пользователя как обычный UEFI драйвер.

2. Работа драйвера не должна замедлять процесс загрузки машины конечного пользователя.

Кроме того, мы хотим доказать, что ведение логов из UEFI возможно, создав прототип драйвера. Мы рассмотрим только логирование среды UEFI. Однако, созданная кодовая база послужит средством для перехода к более широкой теме **логирования ОС извне** (раздел 1.3). Поэтому мониторинг ОС не входит в рамки данной работы из-за своей сложности и невозможности рассмотрения в рамках одной диссертации.

1.3 Получение криминалистических доказательств вне ОС

Почему внешнее логирование привлекательно для криминалистов аналитиков? Это устраняет еще один уровень доверия. Вместо того чтобы хранить журналы в файловой системе, которая может быть легко подделана вредоносным ПО, внешнее логирование хранит их в недоступном для злоумышленников месте (например, на защищенном паролем сетевом

томе). UEFI представляет собой подходящую среду для создания инструментов, которые помогут судебным аналитикам получить цифровые доказательства независимо от взломанной ОС. Это позволяет уменьшить площадь атаки и получить надежные данные.

1.4 Новизна

1.4.1 Предыдущие работы, связанные с обнаружением основного набора файлов

Хотя Ким нашел способ вычислить основной набор файлов, его подход не был предназначен для того, чтобы стать готовым к производству решением. Его разработка занимает несколько часов и требует развертывания тяжелого окружения для выполнения кода. Наш подход использует UEFI для достижения того же уровня точности, но с меньшими затратами. А именно, вся функциональность содержится в отдельном драйвере, что означает простоту развертывания и возможность повторного использования несколькими агентами, работающими в операционной системе. Выбор UEFI обусловлен его растущей популярностью и независимостью от ОС.

1.4.2 Microsoft Prefetcher

На первый взгляд, можно подумать, что Prefetcher занимается тем же, чем и это исследование. Это не так, Prefetcher решает другую проблему аналогичными средствами: он отслеживает обращения к диску и затем использует эту информацию для оптимизации [4]. Предлагаемый драйвер имеет другую цель и использует другой подход:

1. Основан на UEFI, а не на ОС
2. Запускается намного раньше (раньше, чем сама ОС)
3. Использует протоколы UEFI, а не примитивы ядра ОС

Автономность от ОС дает огромное преимущество в плане возможности мониторинга ОС от самой загрузки до выгрузки. Если ОС скомпрометирована вредоносным ПО, то данные, собранные независимыми источниками, считаются надежными и могут быть уверенно использованы криминалистами.

1.5 Частичное резервное копирование

Частичные резервные копии играют важную роль в Active Restore. Прежде чем приступить к активной подгрузке данных для поддержания работоспособности системы, Active Restore должен загрузить ОС. Для этого нам необходимо знать вышеупомянутый набор файлов, а именно частичную резервную копию. Проще говоря, храня его в надежном месте, мы всегда можем скопировать его на диск и сразу же загрузиться с него.

Даже если основной набор файлов слишком велик для частичного резервного копирования, он может помочь упорядочить файлы резервного копирования в соответствии с их приоритетами.

1.6 Масштабирование

Помимо частичного резервного копирования, есть и другие возможные применения этой схемы. Одно из них относится к области форензики

и называется внешним логированием. Исследователи уже предлагали способы выполнения такого типа протоколирования [5]. Однако никто еще не описал метод сбора цифровых доказательств (например, хронологии файловой системы) именно из UEFI, что очень жаль, поскольку это позволяет избежать использования виртуализации. Более того, реализация логирующего UEFI-драйвера может стать хорошим ориентиром для других разработчиков систем.

Чтобы собрать основной набор файлов, нам нужно отслеживать весь процесс загрузки системы, от UEFI до ОС и далее. В данном исследовании мы рассматриваем только UEFI, но дальнейшие работы должны быть способны расширить эту область на внешнее логирование.

Применение результатов данной работы возможно в ходе реализации Программы деятельности лидирующего исследовательского центра АНО ВО "Университет Иннополис"в области систем распределенного реестра после дополнительных исследований.

Глава 2

Методология

Данное исследование направлено на решение проблемы регистрации дисковой активности в UEFI и построение структуры для вычисления существенных наборов файлов при соблюдении двух важных критериев:

1. Система должна быть легко развертываемой на ПК с поддержкой UEFI

2. Система не должна вызывать существенных накладных расходов во время ведения журнала

В разделе 2.1 будут представлены соответствующие компоненты внутренних компонентов UEFI, включая протоколы и типы драйверов. Подраздел 2.1.3 ответит на вопрос, почему мы выбрали Block I/O в качестве протокола для мониторинга. В подразделе 2.1.4 будут рассмотрены общие структуры данных в памяти UEFI - важное архитектурное решение, которое делает наш подход возможным.

В разделе 2.2 мы представим наши соображения о том, какой подход лучше: регистрация имен файлов или секторов диска (также известных

как блоки). В подразделе 2.3.2 мы объясним, почему преобразование LBA в имена файлов не подходит для среды UEFI.

2.1 Компоненты UEFI

2.1.1 Типы драйверов UEFI

Существует только два типа драйверов UEFI: драйверы службы загрузки и драйверы времени выполнения [6]. Драйверы времени выполнения отличаются только тем, что они сохраняются на этапе загрузки системы и продолжают работать после того, как операционная система переходит под управление машины. Сначала мы начнем с драйверов службы загрузки, а затем перейдем к драйверам времени выполнения. Это связано главным образом с тем, что первые в определенный момент удаляются из памяти и, таким образом, не могут напрямую связываться с приложениями уровня ОС. За исключением использования файловой системы, конечно.

2.1.2 Протоколы UEFI

Независимо от типа, любой драйвер UEFI существует с единственной целью - инициализировать устройства, создавать и потреблять полезные протоколы для других драйверов. В то время как первое не имело значения в текущем контексте, второе стало фокусом нашего эксперимента.

Протоколы, созданные драйверами, могут затем безопасно использоваться другими пользователями для любых целей. Таким образом, протокол Block I/O предоставляет услуги по чтению и записи блоков с блочных устройств. Он имеет простой интерфейс, состоящий из двух основных

```
// BlockIo.h excerpt
struct _EFI_BLOCK_IO_PROTOCOL {
    UINT64 Revision;
    EFI_BLOCK_IO_MEDIA* Media;
    EFI_BLOCK_RESET Reset;
    EFI_BLOCK_READ ReadBlocks;
    EFI_BLOCK_WRITE WriteBlocks;
    EFI_BLOCK_FLUSH FlushBlocks;
};
```

Рис. 2.1: Интерфейс протокола Block I/O

функций: ReadBlocks и WriteBlocks (рис. 2.1).

Протокол UEFI - это интерфейс к модулю. Например, протокол Block I/O позволяет разработчикам управлять блочными интерфейсами ввода-вывода. Другими словами, он предоставляет средства для работы с блочными устройствами, чтения и записи данных.

2.1.3 Block I/O, Disk I/O или Block I/O Crypto?

Существует три низкоуровневых протокола UEFI, используемых для записи на диск: Block I/O, Disk I/O и Block I/O Crypto. Поскольку Disk I/O является оберткой вокруг Block I/O, в нашей работе мы придерживаемся протокола Block I/O. Криптографический аналог Block I/O редко используется в системах [6].

Мы рассматривали возможность использования протокола Simple File System. Однако он подходит только для работы с FAT-совместимыми разделами [6].

Важно заметить, что Block I/O очень близок к аппаратному обеспечению, самый близкий из всех протоколов. Это означает, что драйвер либо

использует Block I/O для записи на диск, либо любой другой протокол, который в конечном итоге использует Block I/O. Это означает, что возможность отслеживать использование протокола Block I/O дает возможность регистрировать все обращения к диску. **Вопрос в том, как это отследить?**

2.1.4 Общие структуры данных

Чтобы регистрировать каждый доступ к блочному вводу/выводу, нам нужно подключить функции этого протокола. Это возможно только благодаря тому, что потребители протокола не создают новых экземпляров, они потребляют один и тот же экземпляр, что означает, что все они обращаются к функциям по одним и тем же указателям.

Хотя есть и исключение. UEFI генерирует отдельный экземпляр для каждого контроллера устройства. Это означает, что мы должны уникально идентифицировать каждый экземпляр и хранить его в отдельной структуре данных в памяти. Нет необходимости сохранять эту информацию между запусками, поскольку она теряет свое значение от одного запуска к другому.

Всякий раз, когда драйвер хочет применить существующий протокол, ему необходимо обратиться к определенной структуре данных, называемой "EFI Boot Services Table которая, по сути, является контейнером сервисов. Она включает в себя указатели на функции, которые позволяют находить протоколы и создавать новые.

Как только протокол найден, вы получаете доступ к его общим интерфейсам. Это значит, что можно аккуратно переписать существующие указатели своими собственными функциями, чтобы расширить функцио-

нальность и, таким образом, реализовать хуки.

2.2 Логирование блоков или логирование имён файлов

В отличие от логирования на уровне файлов, логирование на уровне блоков позволяет добиться более высокой детализации. Вы можете различать разные части файла. Когда мы храним в логе информацию на уровне файлов (например, имя файла и абсолютный путь), нет возможности узнать, какие именно части файла были прочитаны. Например, хранение файла размером 5 ГБ значительно увеличивает размер частичной резервной копии (Partial Backup, PB), хотя, возможно, при загрузке требуется лишь небольшая его часть. В этом случае много места и пропускной способности тратится впустую. Хорошим примером может служить реестр Windows, возможно, программа обращается только к нескольким килобайтам из него. Хотя из-за гранулярности вам придется хранить весь реестр в PB. Превращение одного блока в атомарную единицу дает нам огромное преимущество.

Таким образом, логирование на уровне блоков превосходит логирование на уровне файлов в двух аспектах

1. Высокая гранулярность помогает сохранить небольшой размер частичной резервной копии

2. Отсутствие преобразования LBA в имя файла [1] делает драйвер более простым в создании

[1]Logical Block Address (LBA) - это адрес блока в виртуализированном пространстве диска, который принадлежит файлу

С другой стороны, у него есть существенный недостаток, поскольку он не может быть прочитан человеком. Более того, данное исследование существует в контексте Acronis Active Restore (AR) и является частью компонента частичной резервной копии (Partial Backup, PB). Поэтому нам необходимо учитывать и требования AR. Его драйвер фильтра работает на уровне файлов, а не блоков. Это делает целесообразным добавление функции перевода LBA в имя файла в драйвер UEFI или, по крайней мере, в отдельный компонент: перевод может быть выполнен в пространстве пользователя, чтобы уменьшить сложность драйвера и убрать ненужные накладные расходы.

2.3 Недостатки хуков Block I/O

В этом разделе будут рассмотрены недостатки Block I/O с точки зрения логирования. Подраздел 2.3.1 объясняет, почему нам нужно отслеживать контекст хуков. В подразделе 2.3.2 будет приведена мотивация передачи преобразования LBA в файл скриптам пользовательского уровня [2]. Значение битов и байтов в наших двоичных лог-файлах будет раскрыто в разделе 3.2.

2.3.1 Почему мы должны отслеживать контекст?

Логирование чисто LBA[3] не имело смысла, поскольку LBA имел значение только для конкретного диска или раздела. К сожалению, Block I/O не передавал никакой другой информации в подключаемые функции. По-

[2]Пространство пользователя - среда программирования на уровне ОС

[3]Logical Block Address (LBA) - это адрес внутри линейного виртуального пространства, который в конечном итоге отображается на диск

этому мы использовали структуру HookingContext и hashmap для хранения информации о дисках и разделах.

Мы поместили эту информацию в двоичный лог (раздел 3.2.1) для последующей обработки другими пользовательскими сценариями.

2.3.2 Конвертация LBA в имя файла внутри UEFI

Учитывая, что все метаданные известны, перевод этих данных в расположение файлов все еще громоздок. Это связано со сложностью компоновки файловой системы, которая требует низкоуровневого разбора. Мы отказались от этого подхода в пользу обработки в пространстве пользователя, чтобы избежать раздувания и, что самое главное, устранить лишние накладные расходы на логирование.

2.4 Обзор Архитектуры

Для решения проблемы мониторинга чтения с дисков в UEFI и вычисления основного набора файлов, мы предложили следующую архитектуру системы:

1. Перехватывающий драйвер
2. Тестирующий драйвер
3. Скрипт перевода с привязкой к The Sleuth Kit (TSK)

Перехватывающий драйвер не производил никаких протоколов. Однако он потреблял все существующие экземпляры Block I/O и заменял указатели на WriteBlocks и ReadBlocks своими собственными функциями.

Мы выбрали именно Block I/O, потому что это самый близкий драйвер UEFI к чтению и записи на уровне блоков.

Тестовый драйвер использовался на этапе разработки для тестирования перехватывающего драйвера. Он также не производил никаких протоколов. Он использовал только блочный ввод-вывод для проверки того, что в логе появляются только действительные LBA.

Скрипт перевода работал в пространстве пользователя и был фактически оберткой вокруг TSK. После сохранения лога в файловую систему перехватывающим драйвером и загрузки ОС, он строил индекс сопоставлений LBA с именами файлов и выводил человекочитаемый лог чтения с файлов с диска. Мы решили перенести трансляцию лога в пространство пользователя, потому что в противном случае это привело бы к большим накладным расходам в перехватывающем драйвере.

Глава 3

Имплементация и эксперимент

Для того чтобы решить проблему сбора необходимого набора файлов и попытаться исследовать внешнее протоколирование из UEFI, мы должны создать драйвер. Поскольку методология определена, мы должны перейти к реализации. В разделе 3.1 будет подробно рассказано о том, как функции протокола по умолчанию могут быть настроены на новое поведение. В разделе 3.2 мы раскроем бинарную структуру наших логов и то, как она потом понимается.

3.1 Осуществляем перехват Block I/O

После того, как мы нашли буфер с хэндлами протокола Block I/O, хукинг потребовал два дополнительных шага для выполнения на каждом экземпляре протокола:

```
// HookingDriver.c excerpt
EFI_STATUS
EFIAPI
ReadBlocksHook(
    IN EFI_BLOCK_IO_PROTOCOL* This,
    IN UINT32                 MediaId,
    IN EFI_LBA                Lba,
    IN UINTN                  BufferSize,
    OUT VOID*                 Buffer
)
{
    DEBUG((EFI_D_INFO, "ReadBlocksHook: argument list: %x %x %x\r\n",
        MediaId, Lba, BufferSize));
    AppendToLog(This, MediaId, Lba, BufferSize, Buffer, TRUE);
    return ReadBlocksOrigAddress(This, MediaId, Lba, BufferSize, Buffer);
}
```

Рис. 3.1: Интерфейс протокола Block I/O

1. Сохранение контекстного кортежа[1] в структуру данных hashmap

2. Замена ReadBlocks и WriteBlocks на крючки

Оба хука следуют простой схеме: печатают отладочное сообщение, добавляют в лог-файл и вызывают базовую функцию ввода-вывода блоков (рис. 3.1).

3.2 Разбор данных лога

Двоичный лог содержал в себе последовательность записей, где каждая запись предоставляла информацию о диске, разделе и LBA, где данные были записаны или откуда они были прочитаны.

[1] В основном идентификаторы файловой системы и старые указатели

3.2.1 Формат двоичного журнала

1. красный обозначает байт чтения/записи

2. зелёный обозначает размер записываемого буфера

3. синий обозначает LBA (также известный как сектор диска), который в настоящее время читается или записывается

4. циан обозначает идентификатор физического диска - GUID

0000	01	00	00	00	00	00	00	00	40	a8	03	00	00	00	00	00
0010	00	08	00	00	00	00	00	00	ff	ff	ff	ff	ff	ff	ff	ff
0020	87	50	a3	f8	00	00	00	00	ff	ff	ff	ff	ff	ff	ff	ff
0030	01	00	00	00	00	00	00	00	00	a8	03	00	00	00	00	00
0040	00	08	00	00	00	00	00	00	ff	ff	ff	ff	ff	ff	ff	ff
0050	87	50	a3	f8	00	00	00	00	ff	ff	ff	ff	ff	ff	ff	ff

Глава 4

Результаты и оценка

В этой главе мы расскажем о результатах нашего эксперимента. В разделе 4.1 мы обсудим то, что мы собрали на разных этапах эксперимента и исследования в целом. В разделе 4.2 будут интерпретированы факты и их влияние на данное исследование и будущее внешнего протоколирования.

4.1 Результаты

4.1.1 Сбор лога

Мы развернули код подключения в существующую среду UEFI и смогли регистрировать обращения к диску на уровне блоков. В демонстрационных целях и для удобства команды Active Restore (AR) мы также создали модуль перевода на Python с помощью The Sleuth Kit (TSK) API. Модуль перевода будет анализировать двоичный журнал и выводить список приоритетных файлов.

4.1.2 Перевод данных журнала

Модуль перевода включал единственный скрипт, написанный на Python и использующий TSK API. Этот модуль переводил двоичный лог в список файлов, отсортированных по приоритету.

4.2 Интерпретация результатов

Сбор логов через хукинг увенчался полным успехом. Мы смогли отследить все обращения к диску и собрать необходимый набор файлов для ESP.

Идея хукинга показала себя как идеальный инструмент для такого рода сценариев. Это означает, что мы сможем применить ее в будущем, когда начнем разрабатывать компонент мониторинга ОС.

Перевод лога стал возможен только благодаря низкоуровневой природе блочного ввода-вывода - LBA и сектора диска представляют собой одно и то же. Исследователи могут применить это и в будущих работах.

Глава 5

Обсуждение

Мы поставили перед собой три основные задачи при создании прототипа водителя:

1. Отслеживать все чтения блоков в UEFI до загрузки ОС
2. Управлять, чтобы сохранить низкие накладные расходы
3. Сделать систему простой для конечного пользователя в плане установки и обслуживания

Мониторинг протокола UEFI Block I/O помог собрать необходимый набор файлов. Несмотря на то, что этот набор касался только активности UEFI до его выгрузки и запуска ОС, это было именно то, на что мы рассчитывали: мониторинг внутренних компонентов UEFI, а не самой ОС, с которой предполагалось разобраться в будущих работах. Нагрузка на систему не сильно изменилась, и поэтому мы полагаем, что накладные расходы удовлетворяют целям данной работы. Цель сохранения низких накладных расходов также была достигнута.

5.1 Формальная верификация

Предложенный прототип драйвера хранил записи журнала в очереди, которая по сути представляла собой связный список. Кроме того, мы использовали хэшмап для связывания указателя на объект протокола Block I/O в его контекст.

Производительность драйвера зависела исключительно от этих двух структур данных. Поэтому было важно покрыть их реализацию тестами, а еще лучше - формально верифицировать их. К сожалению, в рамках одной диссертации этого достичь не удалось.

В будущих работах исследователи могут формально верифицировать эти компоненты на основе следующих работ: Nguyen [7], который предлагает механизм верификации для красно-черного дерева, и Tuch [8], который предлагает способ формальной верификации управления памятью на языке C.

Глава 6

Заключение

Цель данного исследования - расширить тему внешнего логирования путем переноса возможностей логирования из гипервизора в UEFI. Предыдущие исследования о **внешнем логировании** [1] были сосредоточены на методах виртуализации: мониторинг гостевых машин из гипервизора [2].

С другой стороны, мы также затронули тему **обнаружения набора основных файлов** [3]. Этот набор существует, что было доказано исследователями в прошлом. Однако вычисление такого набора на машинах конечных пользователей никогда не было удобным. Мы показали, что вычисление этого набора может быть быстрым и масштабируемым, реализовав его в UEFI в меньшем масштабе: набор необходимых файлов для фазы загрузки ОС.

Чтобы проверить гипотезу о возможности мониторинга дисковой активности UEFI, мы решили реализовать хуки протокола Block I/O [4]. Мы

[1] Внешнее логирование - мониторинг извне операционной системы

[2] Гипервизор - микропрограммное решение, которое запускает виртуальные машины

[3] Обнаружение набора необходимых файлов - поиск файлов, которые абсолютно необходимы для загрузки ОС

[4] Block I/O - низкоуровневый протокол UEFI, предоставляющий интерфейсы для блочных устройств

выбрали протокол Block I/O для хуков, потому что он наиболее близок к микропрограммной оболчке и не может быть обойден компонентами UEFI (например, драйверами или приложениями) при записи на диск.

Чтобы избежать дорогостоящей записи в файл, мы сбрасывали журнал на диск только один раз - при выгрузке UEFI.

В качестве последнего шага мы перевели сектора в имена файлов, чтобы журнал соответствовал другим компонентам Active Restore (например, драйверу фильтра Windows) и был читабельным.

Мы реализовали два основных компонента: драйвер подцепления и скрипт перевода. Драйвер перехвата использовал базовые вызовы UEFI API для перечисления всех экземпляров Block I/O и замены целевых функций на перехватывающие.

Скрипт трансляции был, по сути, оберткой вокруг The Sleuth Kit[5] (TSK). Обращаясь к TSK, он разбирал EFI System Partition[6] (ESP) и извлекал сопоставления имен файлов с секторами.

Результаты этого исследования показали, что логирование в UEFI с помощью простых хуков протоколов возможно, в то время как Ким [3] только доказал существование необходимого набора файлов. Несмотря на то, что у этого подхода есть свои ограничения: мониторинг не идет дальше UEFI - можно смело сказать, что он проложил новый путь в будущее внешнего логирования.

[5]TSK - инструментарий криминалиста для анализа метаданных файловой системы

[6]ESP - раздел, в котором находятся EFI и системные загрузчики

Список литературы

[1] M. Krau, «Clarifying the Ten Most Common Misconceptions About UEFI,» с. 1—21, 2014.

[2] Acronis, *Glossary*, 2021. url: https://www.acronis.com/en-eu/support/documentation/ABR11.5/546.html#o543 (дата обр. 25.01.2021).

[3] R. Kim, «Acronis Active Restore on Windows: Files classification for restoration by priority,» 2020. url: https://e.lanbook.com/vkr/48166?category=71.

[4] R. Mark, *Microsoft Windows internals : Microsoft Windows server 2003, Windows XP, and Windows 2000*, 4. ed. Redmond, Wash.: Redmond, WA : Microsoft Press, 2003, с. 458. url: https://archive.org/details/isbn_9780735619173/page/458/mode/2up.

[5] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai и P. M. Chen, «ReVirt: Enabling intrusion analysis through virtual-machine logging and replay,» *Operating Systems Review (ACM)*, т. 36, № Special Issue, с. 211—224, 2002, ISSN: 01635980. DOI: 10.1145/844128.844148.

[6] Unified EFI Inc., «Unified Extensible Firmware Interface Specification 2.8,» № March, с. 101—112, 2019. url: https://uefi.org/sites/default/files/resources/UEFI_Spec_2_8_final.pdf.

[7] H. Nguyen, *Formal verification of a red-black tree data structure*, март 2019. url: http://essay.utwente.nl/77569/.

[8] H. Tuch, *Formal Verification of C Systems Code*, апр. 2009. url: https://link.springer.com/article/10.1007/s10817-009-9120-2#citeas.