



SAMInside

Скачать v2.6.3.0
Купить лицензию

Архив обновлен: 06.01.2009

PasswordsPro

Скачать v2.5.1.1
Купить лицензию

Архив обновлен: 29.04.2009

Extreme GPU Bruteforcer

Скачать v1.5
Купить лицензию

Архив обновлен: 18.05.2009

11.05.2009

Вышла EGB v1.5

[Обсудить](#) (комментариев: 53)

29.04.2009

Вышла PasswordsPro v2.5.1.1

[Обсудить](#) (комментариев: 2)

01.04.2009

Вышла PasswordsPro v2.5.1.0

[Обсудить](#) (комментариев: 2)

Словари

Файлов: 73

Библиотека

Документов: 404

Генератор хэшей

Алгоритмов: 109

Подписка на новости

Подписчиков: 877

Дополнительные сервисы

Всего запросов: 19230434

Всего посещений: 3127204

Посетителей сегодня: 1098

Форум

Сообщений: 18829

Последнее: 03.06.2009, 09:18

База хэшей

Найдено паролей: 1579814

Всего хэшей в базе: 35634161

[На главную](#)

[Контакты](#)



Файловая система NTFS извне и изнутри

Автор: (с) [Крис Касперски](#) aka [мышцх](#)

"Покажи мне свои структуры данных, и я скажу, кто ты!"

Народная программистская мудрость

В этой статье мы рассмотрим основные структуры данных файловой системы NTFS, определяющие ее суть - главную файловую запись MFT, файловые записи FILE Record, последовательности обновления (update sequence или fix-ups), атрибуты (также называемые потоками - streams) и списки отрезков (run-lists). Без этих знаний осмысленная работа с редактором диска и ручное или полуавтоматическое восстановление данных просто невозможны!

Введение

Файловую систему NTFS принято описывать как сложную реляционную базу данных, обескураживающую грандиозностью своего архитектурного замысла не одно поколение начинающих исследователей. NTFS похожа на огромный, окутанный мраком лабиринт, в котором очень легко заблудиться. Хакеры давно разобрались с основными структурами данных, осветив магистральные коридоры лабиринта светом множества факелов. Боковые ответвления разведаны намного хуже и все еще находятся по власти Тьмы, хранящей множество смертоносных ловушек, ждущих своих исследователей. В общем, если NTFS-"читалку" можно запрограммировать буквально за один вечер (с отладкой!), писать на NTFS-тома еще никто не рисковал.

К счастью, никто не требует от нас написания полноценного NTFS-драйвера! Наша задача значительно скромнее - вернуть разрушенный том в рабочее состояние, пригодное для восприятия операционной системой (задача максимум) или извлечь из него все ценные файлы (задача минимум). Вникать в структуру журналов транзакций, дескрипторов безопасности, двоичных деревьев индексаций и т.д. для этого совершенно необязательно! Реально нам потребуется разобраться лишь с устройством главной файловой записи - MFT и нескольких дочерних подструктур.

Основным источником данных по NTFS служат:

а) Книга Хелен Кастер (Helen Custer, часто сокращаемая до просто "Helen") **"Inside the Windows NT file system"** (в русском издании она входит в состав **"Основы Windows NT и NTFS"**), подробно описывающая концепции файловой системы и дающая о ней общее представление. К сожалению, все объяснения ведутся на абстрактном уровне без указания конкретных числовых значений, смещений и структур. К тому же, в операционных системах Windows 2000 и Windows XP с файловой системой произошли значительные изменения, никак не отраженные в книге. Если не найдете эту книгу в магазинах - ищите ее в файлообменных сетях. В них есть все!

б) Хакерская документация от коллектива "Linux-NTFS Project" (<http://linux-ntfs.sourceforge.net/>), чьим хобби долгое время была разработка независимого NTFS-драйвера для OS Linux, однако сейчас энтузиазм команды начал стремительно угасать. Это выдающееся творение, подробно описывающее все ключевые структуры файловой системы (естественно, на английском языке), отнюдь не заменяет книгу Хелен, а лишь расширяет ее! Разобраться в NTFS-project'e без знаний NTFS очень и очень

непросто!

в) Документация от Active Data Recovery Software на утилиту Active Uneraser, бесплатную копию которой можно найти на сайте www.NTFS.com. Это своеобразный синтез книги Хелен и Linux-NTFS Project, описывающий важнейшие структуры данных и обходящий стороной все вопросы, которые только можно обойти. Здесь же можно найти до предела выхолощенное изложение методики восстановления данных. В общем, если не найдете Хелен, скачайте демонстрационную версию Active Uneraser и воспользуйтесь прилагаемой к ней документацией. Внимание! Active Uneraser поставляется в двух вариантах - образе FDD и образе CD, документация присутствует только на последнем из них.

г) Контекстная помощь на Disk Explorer также содержит достаточно подробное описание файловой системы, однако на редкость бестолково организованное. Для упрощения навигации по тексту рекомендуется декомпилировать chm-файл в обычный текст, вручную перегнав его в MS Word, pdf или любой другой симпатичный вам формат.

Наконец, вы можете воспользоваться этой статьей. Однако наличие документации Linux-NTFS Project все же очень желательно, поскольку мы будем часто ссылаться на нее.

Версии NTFS

Служебные структуры файловой системы NTFS не остаются постоянными, а слегка меняются от одной версии Windows NT к другой (см. таблицу 1). Этот факт следует принять во внимание при использовании автоматизированных "докторов". Попав на более свежую версию NTFS, "доктор", не оснащенный мощным AI, может запутаться в измененных структурах и разрушить вполне здоровый том.

Версия NTFS	Операционная система	Условное обозначение
1.2	Windows NT	NT
3.0	Windows 2000	W2K
3.1	Windows XP	XP

Таблица 1. Определение версии NTFS по операционной системе.

Полезный совет

Как быстро узнать тип текущего раздела - FAT или NTFS? Да очень просто - достаточно попробовать создать в его корневом каталоге файл \$mft - если он будет создан успешно, то это FAT и, соответственно, наоборот. Если файл \$Extend будет создан успешно, то версия файловой системы 3.0 или выше.

Обзор NTFS с высоты птичьего полета

Основным структурным элементом всякой файловой системы является **том (volume)**, в случае с FAT совпадающий с **разделом (partition)**, о котором мы говорили в прошлой статье. NTFS поддерживает тома, состоящие из нескольких разделов. Подробнее схему отображения томов на разделы мы обсудим в следующей статье этого цикла, а пока же будем для простоты считать, что том представляет собой отформатированный раздел (т.е. раздел, содержащий служебные структуры файловой системы).

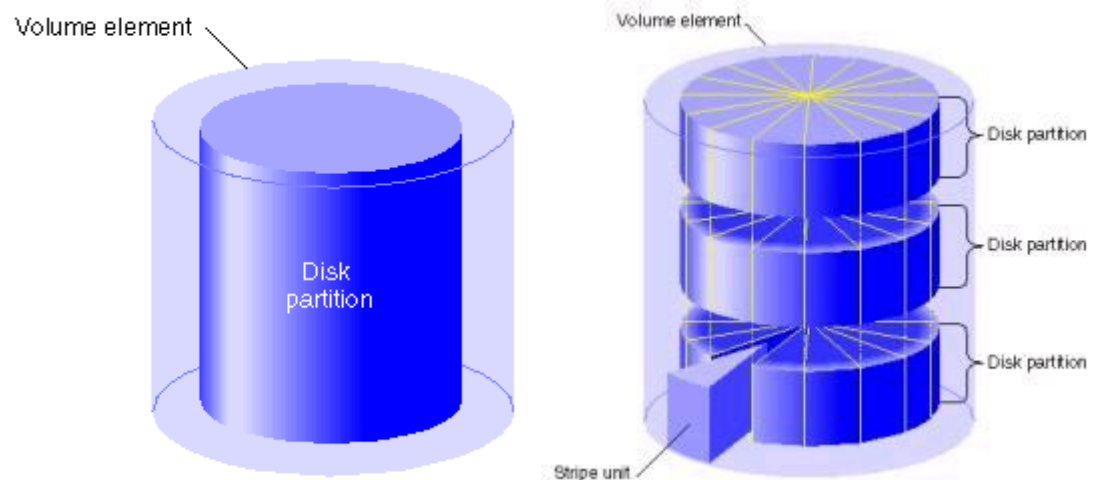


Рисунок 1. Обычный (слева) и разряженный (справа) тома.

Большинство файловых систем трактуют том, как совокупность файлов, свободного дискового пространства и служебных структур файловой системы, но в NTFS **все** служебные структуры представлены файлами, которые (как и это и положено файлу) могут находиться в **любом** месте тома, при необходимости фрагментируя себя на несколько частей.

Самым главным служебным файлом является **\$MFT - Master File Table** (Главная Файловая Таблица) - своеобразная база данных, хранящая информацию обо всех файлах тома - их именах, атрибутах, способе и порядке размещения на диске (каталог также является файлом особого типа со списком принадлежащих ему файлов и подкаталогов внутри). Важно подчеркнуть, что в MFT присутствуют **все** файлы, находящиеся во **всех** подкаталогах тома, поэтому для восстановления диска наличия \$MFT-файла будет вполне достаточно.

Остальные служебные файлы (кстати говоря, называемые **метафайлами** или **метаданными** - metafile/metadate, соответственно, и всегда предваряются знаком доллара '\$') носят сугубо вспомогательный характер, интересный только самой файловой системе. К ним в первую очередь относятся: \$LogFile - файл транзакций, \$Bitmap - карта свободного/занятого пространства, \$BadClust - перечень плохих кластеров и т.д. Подробнее см. "Назначение некоторых служебных файлов". Текущие версии Windows блокируют доступ к служебным файлам с прикладного уровня (даже с правами администратора!) и всякая попытка открытия/создания такого файла в корневом каталоге обречена на неудачу.

Классическое определение, данное в учебниках информатики, отождествляет файл с именованной записью на диске. Большинство файловых систем добавляет к этому понятие **атрибута (attribute)** - некоторой вспомогательной характеристики, описывающей время создания, права доступа и т.д. В NTFS имя файла, данные файла и его атрибуты полностью уравниваются в правах. Можно сказать, что всякий NTFS-файл представляет собой совокупность атрибутов, каждый из которых хранится как отдельный **поток (stream)** байтов, поэтому во избежание путаницы атрибуты, хранящие данные файла, часто называют потоками.

Каждый атрибут состоит из **тела (body)** и **заголовка (header)**. Атрибуты делятся на **резидентные (resident)** и **нерезидентные (non-resident)**. Резидентные атрибуты хранятся непосредственно в \$MFT, что существенно уменьшает грануляцию дискового пространства и сокращает время доступа. Нерезидентные хранят в \$MFT лишь свой заголовок, описывающий порядок

размещения атрибута на диске (см. "Списки отрезков").

Назначение атрибута определяется его **типом (type)** - четырехбайтовым шестнадцатеричным значением. При желании атрибуту можно дать еще и **имя (name)**, состоящее из символов, входящих в соответствующее пространство имен (см. "Пространства имен"). Подавляющее большинство файлов имеет, по меньшей мере, три атрибута: стандартная информация о файле (время создания, модификации, последнего доступа, права доступа и т.д.) хранится в атрибуте типа 10h, условно обозначаемом **\$STANDARD_INFORMATION**. Ранние версии Windows NT позволяли обращаться к атрибутам по их условным обозначениям, однако Windows 2000 и Windows XP лишены этой возможности. Полное имя файла (не путать с путем!) хранится в атрибуте типа 30h (**\$FILE_NAME**). Если у файла есть одно или более альтернативных имен (например, MS-DOS имя), таких атрибутов может быть и несколько (см. "Типы атрибутов"). Здесь же хранится **ссылка (file reference)** на материнский каталог, позволяющая разобраться - к какому каталогу данный файл/подкаталог принадлежит. Данные файла по умолчанию хранятся в безымянном атрибуте типа 80h (**\$DATA**), однако при желании прикладные приложения могут создавать дополнительные потоки данных, отделяя имя атрибута от имени файла знаком двоеточия (например, "ECHO xxx > file:attr1; ECHO yyy > file:attr2; more < file:attr1; more < file:attr2").

Изначально в NTFS была заложена способность индексации любых атрибутов, значительно сокращающая время поиска файла по заданному списку критериев (например, времени последнего доступа). Внутренне индексы хранятся в виде двоичных деревьев, поэтому среднее время выполнения запроса оценивается как $O(\lg n)$. Однако в текущих NTFS-драйверах реализована индексация лишь по одному атрибуту - имени файла. Как уже говорилось выше, каталог представляет собой особый файл - файл **индексов (INDEX)**. В отличие от FAT, где файл каталога представляет единственный источник данных об организации файлов, в NTFS файл каталога используется лишь для ускорения доступа к содержимому директории и не является обязательным, поскольку ссылка на материнский каталог всякого файла в обязательном порядке присутствует в атрибуте его имени (\$FILE_NAME).

Каждый атрибут может быть зашифрован, разряжен или сжат. Однако техника работы с такими атрибутами далеко выходит за рамки первичного знакомства с организацией файловой системы и будет рассмотрена позднее. А пока же мы углубимся в изучение фундамента файловой системы - структуры \$MFT.

Главная файловая запись (master file table)

В процессе форматирования логического раздела, в его начале создается так называемая **MTF-зона (MFT-zone)**, по умолчанию занимающая 12.5% от емкости тома (а вовсе не 12%, как утверждается во многих публикациях), хотя в зависимости от значения от параметра NtfsMftZoneReservation она может составлять 25%, 37% или 50%.

В этой области расположен \$MFT-файл, изначально занимающий порядка 64 секторов и растущий от начала MFT-зоны к ее концу по мере создания новых пользовательских файлов/подкаталогов. Таким образом, чем больше файлов содержится на дисковом томе, тем больше размер MFT. Приблизительный размер MFT-файла можно оценить по следующей формуле: **sizeof (FILE Record) * N Files**, где sizeof(FILE Record) обычно равен 1 Кбайт, а N Files - полное количество файлов/подканалов раздела, включая недавно удаленные.

Для предотвращения фрагментации \$MFT-файла MFT-зона держится зарезервированной, вплоть до полного исчерпания свободного пространства тома, затем незадействованный "хвост" MFT-зоны усекается в два раза, освобождая место для пользовательских файлов. Этот процесс может повторяться многократно, вплоть до полной отдачи всего зарезервированного пространства. Решение красивое, хотя и не новое. Многие из файловых систем восьмидесятых позволяли резервировать заданное дисковое пространство в хвосте активных файлов, тем самым сокращая их фрагментацию (причем любых файлов, а не только служебных). В частности, такая способность была у DOS 3.0, разработанной для персональных компьютеров типа Агат. Может, кто помнит такую машину?

Когда \$MFT-файл достигает границ MFT-зоны, в ходе своего последующего роста он неизбежно фрагментируется, вызывая обвальное падение производительности файловой системы, причем подавляющее большинство дефрагментаторов \$MFT-файл не обрабатывают! А ведь API дефрагментации, встроенное в штатный NTFS-драйвер, это в принципе позволяет! Подробности (вместе с самой утилитой дефрагментации) можно найти на сайте Марка Руссиновича. Но, как бы там ни было, **заполнять дисковый том более чем на 88% его емкости категорически не рекомендуется!**

При необходимости, \$MFT-файл может быть перемещен в любую часть диска и тогда в начале тома его уже не окажется. Стартовый адрес \$MFT-файла хранится в Boot-секторе по смещению 30h байт от его начала (см. "boot-сектор", описанный в предыдущей статье данного цикла) и в подавляющем большинстве случаев этот адрес ссылается на 4-й кластер.



Рисунок 2. Структура дискового тома под NTFS.

\$MFT-файл представляет собой массив записей типа **FILE Record** (или в терминологии UNIX - **inodes**), каждая из которых описывает соответствующий ей файл или подкаталог (подробнее см. "Файловые записи"). В подавляющем большинстве случаев один файл/подкаталог полностью описывается одной-единственной записью FILE Record, хотя, теоретически, этих записей может потребоваться и несколько.

Для ссылки на одну файловую запись из другой используется ее порядковый номер (он же индекс) в \$MFT файле, отсчитываемый от нуля. Файловая ссылка (file reference) состоит из двух частей (см. таблицу 2) - 48-битного индекса и 16-битного номера последовательности (sequence number).

При удалении файла/каталога соответствующая ему файловая последовательность помечается как неиспользуемая. При создании новых файлов записи, помеченные как неиспользуемые, могут задействоваться вновь, при этом счетчик номера последовательности, хранящийся внутри файловой записи, увеличивается на единицу. Этот механизм позволяет отслеживать "мертвые" ссылки на уже удаленные файлы - очевидно, sequence number внутри file reference будет отличаться от номера последовательности соответствующей файловой записи (этой проверкой занимается утилита chkdsk и автоматически, насколько мне известно, она не выполняется).

Смещение	Размер	Описание
00h	6	Индекс файловой записи (FILE record number), отсчитываемый от нуля
06h	2	Номер последовательности (sequence number)

Таблица 2. Структура файловой ссылки (file reference).

Первые 12 записей в MFT всегда занимают служебные метафайлы: \$MFT (собственно, сам \$MFT), \$MFTMirr (зеркало \$MFT), \$LogFile (файл транзакций), \$Volume (сведения о дисковом томе), \$AttrDef (определенные атрибуты), '.' (корневой каталог), \$Bitmap (карта свободного пространства), \$Boot (системный загрузчик), \$BadClus (перечень плохих кластеров) и т.д. Подробнее см. таблицу 11.

Первые четыре записи настолько важны, что продублированы в специальном \$MFTMirr-файле, находящимся приблизительно посередине диска (точное расположение хранится в boot-секторе по смещению 38h байт от его начала). Вопреки своему названию, \$MFTMirr это отнюдь не зеркало всего \$MFT-файла, это всего лишь копия первых четырех элементов.

Записи с 12 по 15 помечены как используемые, но в действительности же они пусты (как нетрудно догадаться, это задел на будущее). Записи с 16 по 23 не задействованы и честно помечены как неиспользуемые.

Начиная с 24-й записи, располагаются пользовательские файлы и каталоги. Четыре метафайла, появившихся в W2K - \$ObjId, \$Quota, \$Reparse и \$UsnJrnl могут располагаться в любой записи, номер которой равен 24 или больше (как мы помним, номера файловых записей отсчитываются, начиная с нуля).

Для знакомства с MFT запустим DiskExplorer от Runtime Software, не забывая о том, что он требует прав администратора, в меню "File" найдем пункт "Drive" и в открывшемся диалоговом окне выберем логический диск, который мы хотим редактировать. Затем в меню "Goto" выберем пункт "Mft", заставляя DiskExplorer перейти к MFT, автоматически меняя режим отображения на наиболее естественный (см. рис. 3). Как вариант, можно нажать <F6> (View as File Entry) и промотать несколько первых секторов клавишей <Page Down>.

Для каждого из файлов DiskExplorer сообщает:

1) Номер сектора, к которому данная файловая запись принадлежит (обратите внимание, что номера секторов монотонно увеличиваются на 2, подтверждая тот факт, что размер одной файловой записи равен 1 Кбайту, однако вы можете столкнуться и с другими значениями). Для удобства информация отображается сразу в двух системах исчисления - шестнадцатеричной и десятичной;

2) Основное имя файла/каталога (т.е. имя файла из заголовка файловой записи; некоторые файлы имеют несколько альтернативных имен, подробнее см. "Атрибуты"). Если имя файла/каталога зачеркнуто, значит он был удален, но соответствующая ему файловая запись все еще цела. Чтобы извлечь файл с диска (неважно, удаленный или нет) подведите к нему курсор и нажмите <Ctrl-T> для просмотра его содержимого в шестнадцатеричном виде или <Ctrl-S> для сохранения файла на диск. То же самое можно сделать и через контекстное меню (раздел "recovery"). При нажатии на <Ctrl-C> в буфер обмена копируется последовательность кластеров, занятых файлом (например, "DISKEXPL:K:1034240-1034240").

3) Тип файловой записи - файл это или каталог?

4) Атрибуты файла/каталога - a = архивный файл, r = только на чтение, h = скрытый, s = системный, l = метка тома, d = каталог, c = сжатый файл;

5) Размер файла в байтах в десятичной системе исчисления (не для каталогов!);

6) Дату и время модификации файла/каталога;

7) Номер первого кластера файла/каталога (или "resident" для полностью резидентных файлов/каталогов);

8) Перечень типов NTFS-атрибутов, имеющих у файла/каталога, записанных в шестнадцатеричной нотации (обычно эта строка имеет вид 10 30 80 - атрибут стандартной информации, атрибут имени и атрибут данных файла, подробнее см. "Типы атрибутов");

9) Индекс файловой записи в MFT, выраженный в шестнадцатеричной и десятичной системах исчисления и следующий за словом "No:" (сокращение от Number - номер);

10) Индекс файловой записи материнского каталога, выраженный в шестнадцатеричной и десятичной системах исчисления (5h - если файл принадлежит к корневому каталогу). Для быстрого перемещения по файловым записям выберите в меню "Goto" пункт "Mft no" и введите требуемый индекс в шестнадцатеричной или десятичной форме;

11) Для нерезидентных файлов/каталогов - перечень кластеров, занятых файлом в не декодированном виде (а зря - могли бы и декодировать!). Схема кодирования кластеров подробно описана в главе "Списки отрезков".

Прежде чем продолжать чтение статьи, попробуйте поэкспериментировать в MFT файлами (особенно фрагментированными). Посмотрите, как создаются и удаляются записи из MFT. Лучше всего это делать на диске, содержащем небольшое количество файлов/каталогов. Чтобы не форматировать логический диск, создайте виртуальный (благо количество оперативной памяти современных компьютеров это позволяет).

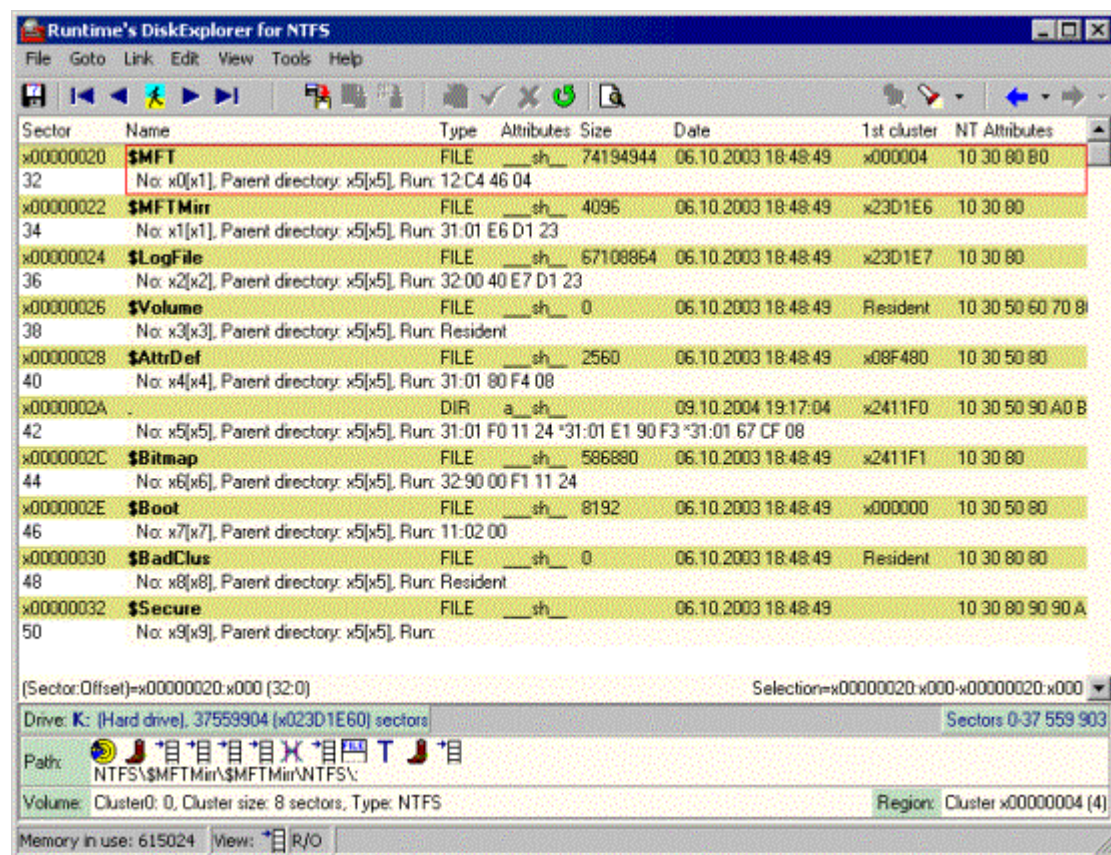


Рисунок 3. DiskExplorer отображает главную файловую запись в естественном виде.

Файловые записи (FILE Record)

Благодаря наличию Disk Explorer'a от Runtime Software с файловыми записями практически никогда не приходится работать вручную, тем не менее знание их структуры нам не помешает.

Структурно файловая запись состоит из **заголовка (header)** и одного или нескольких **атрибутов (attribute)** произвольной длины, завершаемых **маркером конца (end marker)** - четырехбайтовым шестнадцатеричным значением FFFFFFFFh (см. листинг 1). Несмотря на то, что количество и длина атрибутов меняется от одной файловой записи к другой, размер самой структуры **FILE Record** строго фиксирован и в большинстве случаев равен 1 Кбайту (это значение хранится в \$boot-файле, не путать с boot-сектором!). Причем, первый байт файловой записи всегда совпадает с началом сектора.

Если реальная длина атрибутов меньше размеров файловой записи, ее хвост просто не используется. Если же атрибуты не вмещаются в отведенное им пространство, создается дополнительная файловая запись (**extra FILE Record**), ссылающаяся на свою предшественницу.


```

FILE Record
    Header                ; заголовок
    Attribute 1           ; атрибут 1
    Attribute 2           ; атрибут 2
    ...                   ; ...
    Attribute N           ; атрибут N
End Marker (FFFFFFFFh)   ; маркер конца

```

Листинг 1. Структура файловой записи.

Первые четыре байта заголовка оккупированы магической последовательностью "FILE", сигнализирующей о том, что мы имеем дело с файловой записью типа FILE Record. При восстановлении сильно фрагментированного \$MFT файла это обстоятельство играет решающую роль, поскольку позволяет отличить сектора, принадлежащие MFT, от всех остальных секторов.

Следом за сигнатурой идет 16-разрядный указатель, содержащий смещение **последовательности обновления (update sequence)**, см. "Последовательности обновления". Под "указателем" здесь и до конца раздела подразумевается смещение от начала сектора, отсчитываемое от нуля и выраженное в байтах. В NT и W2K это поле всегда равно 002Ah, поэтому для поиска файловых записей можно использовать сигнатуру "FILE*\x00", что уменьшает вероятность ложных срабатываний. Правда, в XP и более старших системах последовательность обновления хранится по смещению 002Dh и поэтому сигнатура приобретает следующий вид "FILE-\x00".

Размер заголовка (кстати сказать, варьирующийся от одной операционной системы к другой) в явном виде нигде не хранится, вместо этого в заголовке присутствует указатель на первый атрибут, содержащий его смещение в байтах относительно начала FILE Record, расположенный по смещению 14h байт от начала сектора. Смещения последующих атрибутов (если они есть) определяются путем сложения размеров всех предыдущих атрибутов (размер каждого из атрибутов содержится в его заголовке) со смещением первого атрибута. За концом последнего атрибута находится маркер конца - FFFFFFFFh.

Вдобавок к этому, длина файловой записи хранится в двух полях - 32-разрядное поле **реального размера (real size)**, находящееся по смещению 18h байт от начала сектора и содержащее совокупный размер заголовка, всех его атрибутов и маркера конца, округленный по 8 байтной границе и 32-разрядное поле **выделенного размера (allocated size)**, находящееся по смещению 1Ch байт от начала сектора, содержащее действительный размер файловой записи в байтах, округленный по размеру сектора. Документация Linux-NTFS Project (версия 0.4) утверждает, что allocated size должен быть кратен размеру кластера, однако в действительности это не так. В частности, на моей машине, allocated size равен четвертинке кластера.

16-разрядное поле флагов, находящееся по смещению 16h байт от начала сектора, в подавляющем большинстве случаев принимает одно из трех следующих значений: 00h - данная файловая запись не используется или ассоциированный с ней файл/каталог удален, 01h - файловая запись используется и описывает файл, 02h - файловая запись используется и описывает каталог.

64-разрядное поле, находящееся по смещению 20h байт от начала сектора содержит индекс базовой файловой записи. Для первой файловой записи это поле всегда равно нулю, а для всех последующих, расширенных (extra) записей - индексу первой файловой записи. Расширенные файловые записи могут находиться в любых частях MFT, не обязательно рядом с основной записью. А коль скоро так, необходим какой-то механизм, обеспечивающий быстрый поиск расширенных файловых записей, принадлежащих данному файлу (просматривать весь MFT целиком не предлагать!). И этот механизм основан на ведении списков атрибутов \$ATTRIBUTE_LIST. Список атрибутов представляет собой специальный атрибут, добавляемый к первой файловой записи и содержащий индексы расширенных записей. Формат списка атрибутов приведен в разделе "типы атрибутов".

Остальные поля заголовка файловой записи не столь важны и поэтому здесь не рассматриваются. При необходимости обращайтесь к документации "Linux-NTFS Project".

Смещение	Размер	ОС	Описание
00h	4	любая	сигнатура (magic number) 'FILE'
04h	2	любая	смещение номера последовательности обновления (update sequence number)
06h	2	любая	размер в словах номера последовательности обновления и массива обновления (Update Sequence Number & Array), условно (S)
08h	8	любая	номер последовательности файла транзакций (\$LogFile Sequence Number или сокращенно LSN)
10h	2	любая	номер последовательности (sequence number)
12h	2	любая	счетчик жестких ссылок (hard link)
14h	2	любая	смещение первого атрибута (attribute)
16h	2	любая	Значение Описание 0x00 файловая запись не используется 0x01 файловая запись используется и описывает файл (file) 0x02 файловая запись используется и описывает каталог (directory) 0x04 только Билл Гейтс знает 0x08 только Билл Гейтс знает
18h	4	любая	реальный размер (real size) файловой записи
1Ch	4	любая	выделенный размер (allocated size) файловой записи
20h	8	любая	ссылка (file reference) на базовую файловую запись (base FILE record) или ноль, если данная файловая запись базовая
28h	2	любая	идентификатор следующего атрибута (next attribute ID)
2Ah	2	XP	для выравнивания
2Ch	4	XP	индекс данной файловой записи (number of this MFT record)
	2	любая	номер последовательности обновления (update sequence number)
	2S-2	любая	массив последовательности обновления (update sequence array)

Таблица 3. Структура заголовка файловой записи (FILE Record).

Последовательности обновления (update sequence)

Будучи очень важными компонентами файловой системы, \$MFT, INDEX и \$LogFile нуждаются в механизме контроля целостности своего содержимого. Традиционно для этого используется ECC/EDC-коды, однако во времена проектирования NTFS процессоры

были не настолько быстрыми, как теперь и расчет корректирующих кодов занимал значительное время, существенно снижающее производительность файловой системы. Поэтому от них пришлось отказаться. Вместо этого, разработчики NTFS применили так называемые **последовательности обновления (update sequence)**, также называемые **fix-up**ами.

В конец каждого из секторов, слагающих файловую запись (INDEX Record, RCRD Record или RSTR Record) записывается специальный 16-байтовый **номер последовательности обновления (update sequence number)**, дублируемый в заголовке файловой записи. При каждой операции чтения два последних байта сектора сверяется с соответствующим полем заголовка и если NTFS-драйвер обнаруживает расхождение, данная файловая запись считается недействительной.

Основное назначение последовательностей обновления - защита от "обрыва записи". Если в процессе записи сектора на диск, исчезнет питающее напряжение, может случиться так, что половина файловой записи будет успешно записана, а половина - сохранит прежнее содержимое (файловая запись, как мы помним, обычно состоит из двух секторов). После восстановления питания, драйвер файловой системы не может уверенно сказать - была ли файловая запись записана целиком или нет. Вот тут-то последовательности обновления и выручают! При каждой перезаписи сектора update sequence увеличивается на единицу и потому, если произошел обрыв записи, значение последовательности обновления, находящейся в заголовке файловой записи не будет совпадать с последовательностью обновления, расположенной в конце сектора.

Оригинальное содержимое, расположенное "под" последовательностью обновления, хранится в специальном **массиве обновления (update sequence array)**, расположенном в заголовке файловой записи непосредственно за концом update sequence number. Для восстановления файловой записи в исходный вид мы должны извлечь из заголовка указатель на update sequence number (он хранится по смещению 04h байт от начала заголовка) и сверить лежащее по этому адресу 16-байтное значение с последним словом каждого из секторов, слагающих файловую запись (INDEX Record, RCRD Record или RSTR Record). Если они не совпадут, значит соответствующая структура данных повреждена и использовать ее следует с очень большой осторожностью (а на первых порах лучше не использовать вообще).

По смещению 006h от начала сектора находится 16-разрядное поле, хранящее совокупный размер номера последовательности обновления вместе с массивом последовательности обновления (sizeof(update sequence number) + sizeof(update sequence array)), выраженный в словах (не в байтах!). Поскольку размер номера последовательности обновления всегда равен одному слову, то размер массива последовательности обновления, выпавший в байтах, равен:
 $(\text{update sequence number} \& \text{update sequence array} - 1) * 2$. Соответственно, смещение массива оригинального содержимого равно $(\text{offset to update sequence number}) + 2$. В NT и W2K update sequence number всегда располагается по смещению 2Ah от начала FILE Record Header/INDEX Header, а update sequence array - по смещению 2Ch. В XP же и более старших системах - по смещениям 2Dh и 2Fh соответственно.

Первое слово массива последовательности обновления соответствует последнему слову первого сектора FILE Record/INDEX. Второе - последнему слову второго сектора, и т.д. Для восстановления сектора в исходный вид мы должны вернуть все элементы массива последовательности обновления на их законные места (естественно, модифицируется не сам сектор, а его копия в памяти).

Продemonстрируем это на следующем примере:

```
--> начало первого сектора FILE Record
00000000: 46 49 4C 45-2A 00 03 00-7C 77 1A 04-02 00 00 00  FILE*.....
00000010: 01 00 02 00-30 00 01 00-28 02 00 00-00 04 00 00  .....
00000020: 00 00 00 00-00 00 00 00-06 00 06 00-00 00 47 11  .....
...
000001F0: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 06 00  .....
<-- конец первого сектора FILE Record
```

```

...
000003F0:  07 CC E1 0D-00 09 00 00-FF FF FF FF-82 79 06 00  .....
<-- конец второго сектора FILE Record

```

Листинг 2. Оригинальная файловая запись до восстановления.

Сигнатура "FILE" указывает на начало файловой записи. А раз так, то по смещению 04h байт будет расположен 16-разрядный указатель на номер последовательности обновления. В данном случае он равен 002Ah. Ок, переходим по смещению 002Ah и видим, что здесь лежит слово 0006h. Перемещаемся в конец сектора и сверяем его с последними двумя байтами. Как и предполагалось, они совпадают. Повторяем ту же самую операцию со следующим сектором. Собственно говоря, количество секторов может и не равняться двум. Чтобы не гадать на кофейной гуще, необходимо извлечь 16-разрядное значение, расположенное по смещению 06h от начала файловой записи (в данном случае оно равно 0003h) и вычесть из него единицу. Действительно, получается два (сектора).

Теперь нам необходимо найти массив последовательности обновления, хранящий оригинальное значение последнего слова каждого из секторов. Смещение массива обновления равно значению указателя на последовательность обновления, увеличенной на два, т.е. в данном случае 002Ah + 02h = 002Ch. Извлекаем первое слово (в данном случае равное 00h 00h) и записываем его в конец первого сектора. Извлекаем следующее слово (47h 11h) и записываем его в конец второго сектора.

В результате чего восстановленный сектор будет выглядеть так:

```

--> начало первого сектора FILE Record
00000000:  46 49 4C 45-2A 00 03 00-7C 77 1A 04-02 00 00 00  FILE*.....
00000010:  01 00 02 00-30 00 01 00-28 02 00 00-00 04 00 00  .....
00000020:  00 00 00 00-00 00 00 00-06 00 06 00-00 00 47 11  .....
...
000001F0:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00  .....
<-- конец первого сектора FILE Record
...
000003F0:  07 CC E1 0D-00 09 00 00-FF FF FF FF-82 79 47 11  .....
<-- конец второго сектора FILE Record

```

Листинг 3. Восстановленная файловая запись.

Внимание! FILE Record, INDEX Record, RCRD Record или RSTR Record искажены последовательностями обновления и в обязательном порядке должны быть восстановлены перед их использованием, в противном случае вместо актуальных данных вы получите мусор!

Атрибуты (attribute)

Структурно всякий атрибут стоит из **атрибутного заголовка (attribute header)** и **тела атрибута (attribute body)**. Заголовок атрибута всегда хранится в файловой записи, расположенной внутри MFT (см. "Файловые записи"). Тела резидентных атрибутов хранятся там же. Нерезидентные атрибуты хранят свое тело вне MFT в одном или нескольких кластерах, перечисленных в заголовке данного атрибута в специальном списке (см. "Списки отрезков"). Если 8-разрядное поле, расположенное по смещению 08h байт от начала атрибутного заголовка, равно нулю - атрибут считается резидентным, а если единице, то - нет. Любые другие значения недопустимы.

Первые четыре байта атрибутного заголовка определяют его тип. Тип атрибута в свою очередь определяет формат представления тела атрибута. В частности, тело атрибута данных (тип: 80h - \$DATA) представляет собой "сырую" последовательность байт. Тело атрибута стандартной информации (тип: 10h - \$STANDARD_INFORMATION) описывает время его создания, права доступа и т.д. Подробнее см. "Типы атрибутов".

Следующие четыре байта заголовка содержат длину атрибута, выражаемую в байтах. Длина нерезидентного атрибута равна сумме длин его тела и заголовка, а длина резидентного атрибута равна длине его заголовка. Короче говоря, если к смещению атрибута добавить его длину, мы получим указатель на следующий атрибут (или маркер конца, если текущий атрибут - последний в цепочке).

Длина тела резидентных атрибутов, выраженная в байтах, хранится в 32-разрядном поле, расположенном по смещению 10h байт от начала атрибутного заголовка. 16-разрядное поле, следующее за его концом, хранит смещение резидентного тела, отсчитываемое от начала атрибутного заголовка. С нерезидентными атрибутами в этом плане все намного сложнее и для хранения длины их тела используется множество полей. **Реальный размер тела атрибута (real size of attribute)**, выраженный в байтах, хранится в 64-разрядном поле, находящимся по смещению 30h байт от начала атрибутного заголовка. Следующее за ним 64-разрядное поле хранит **инициализированный размер потока (initialized data size of the stream)**, выраженный в байтах и, судя по всему, всегда равный реальному размеру тела атрибута. 64-разрядное поле, расположенное по смещению 28h байт от начала атрибутного заголовка, хранит **выделенный размер (allocated size of attribute)**, выраженный в байтах и равный реальному размеру тела атрибута округленному до размера кластера (в большую сторону).

Два 64-разрядных поля, расположенные по смещению 10h и 18h байт от начала атрибутного заголовка задают первый (**starting VCN**) и последний (**last VCN**) номер виртуального кластера, принадлежащего телу нерезидентного атрибута. Виртуальные кластеры представляют собой логические номера кластеров, не зависящие от своего физического расположения на диске. В подавляющем большинстве случаев номер первого кластера тела нерезидентного атрибута равен нулю, а последний - количеству кластеров занятых телом атрибута, уменьшенном на единицу. 16-разрядное поле, расположенное по смещению 20h от начала атрибутного заголовка содержит указатель на массив **Data Runs**, расположенный внутри этого заголовка и описывающий логический порядок размещения нерезидентного тела атрибута на диске (подробнее см. "Списки отрезков").

Каждый атрибут имеет свой собственный **идентификатор (attribute ID)**, уникальный для данной файловой записи и хранящийся в 16-разрядном поле, расположенном по смещению 0Eh от начала атрибутного заголовка.

Если атрибут имеет **имя (attribute Name)**, то 16-разрядное поле, расположенное по смещению 0Ah байт от атрибутного заголовка, содержит указатель на него. Для безымянных атрибутов оно равно нулю (а большинство атрибутов безымянны!). Имя атрибута хранится в атрибутном заголовке в формате UNICODE, а его длина определяется 8-разрядным полем, расположенным по смещению 09h байт от начала атрибутного заголовка.

Если тело атрибута сжато, зашифровано или разряжено, 16-разрядное поле флагов, расположенное по смещению 0Ch байт от начала атрибутного заголовка не равно нулю.

Остальные поля не играют сколь-нибудь существенной роли и потому здесь не рассматриваются.

Смещение	Размер	Значение	Описание
00h	4		тип (type) атрибута (например, 0x10, 0x60, 0xB0)
04h	4		длина атрибута, включая этот заголовок
08h	1	00h	нерезидентный флаг (non-resident flag)

09h	1	N	длина имени атрибута (ноль, если атрибут безымянный)
0Ah	2	18h	смещение имени (ноль, если атрибут безымянный)
0Ch	2	00h	Значение Описание 0001h сжатый атрибут (compressed) 4000h зашифрованный атрибут (encrypted) 8000h разряженный атрибут (sparse)
0Eh	2		идентификатор атрибута (attribute ID)
10h	4	L	длина тела атрибута, без заголовка
14h	2	2N+18h	смещение тела атрибута
16h	1		индексный флаг
17h	1	00h	для выравнивания
18h	2N	UNICODE	имя атрибута (если есть)
2N+18h	L		тело атрибута

Таблица 4. Структура резидентного атрибута.

Смещение	Размер	Значение	Описание
00h	4		тип (type) атрибута (например, 0x20, 0x40)
04h	4		длина атрибута, включая этот заголовок
08h	1	01h	нерезидентный флаг (non-resident flag)
09h	1	N	длина имени атрибута (ноль, если атрибут безымянный)
0Ah	2	40h	смещение имени (ноль, если атрибут безымянный)
0Ch	2		Значение Описание 0001h сжатый атрибут (compressed) 4000h зашифрованный атрибут (encrypted) 8000h разряженный атрибут (sparse)
0Eh	2		идентификатор атрибута (attribute ID)
10h	8		начальный виртуальный кластер (starting VCN)
18h	8		конечный виртуальный кластер (last VCN)
20h	2	2N+40h	смещение списка отрезков (data runs)

22h	2		размер блока сжатия (compression unit size), округленный до 4 байт вверх
24h	4	00h	для выравнивания
28h	8		выделенный размер (allocated size), округленный до размера кластера
30h	8		реальный размер (real size)
38h	8		инициализированный размер потока (initialized data size of the stream)
40h	2N	UNICODE	имя атрибута, если есть
2N+40h	...		список отрезков (data runs)

Таблица 5. Структура нерезидентного атрибута.

Типы атрибутов

NTFS поддерживает больше количество предопределенных типов атрибутов, перечисленных в таблице 8. Тип атрибута определяет его назначение и формат представления тела. Полное описание всех атрибутов заняло бы очень много места и поэтому здесь приводятся лишь наиболее "ходовые" из них, а за информацией об остальных обращайтесь к Linux-NTFS Project.

Значение	ОС	Условное обозначение	Описание
010h	любая	\$STANDARD_INFORMATION	стандартная информация о файле (время, права доступа)
020h	любая	\$ATTRIBUTE_LIST	список атрибутов
030h	любая	\$FILE_NAME	полное имя файла
040h	NT	\$VOLUME_VERSION	версия тома
040h	2K	\$OBJECT_ID	уникальный GUID и прочие ID
050h	любая	\$SECURITY_DESCRIPTOR	дескриптор безопасности и списки прав доступа (ACL)
060h	любая	\$VOLUME_NAME	имя тома
070h	любая	\$VOLUME_INFORMATION	информация о томе
080h	любая	\$DATA	основные данные файла
090h	любая	\$INDEX_ROOT	корень индексов
0A0h	любая	\$INDEX_ALLOCATION	ветви (sub-nodes) индекса
0B0h	любая	\$BITMAP	карта свободного пространства
0C0h	NT	\$SYMBOLIC_LINK	символическая связь
0C0h	2K	\$REPARSE_POINT	для сторонних производителей
0D0h	любая	\$EA_INFORMATION	расширенные атрибуты для HPFS

0E0h	любая	\$EA	расширенные атрибуты для HPFS
0F0h	NT	\$PROPERTY_SET	устарело и ныне не используется
100h	2K	\$LOGGED_UTILITY_STREAM	используется шифрованной файловой системой (EFS)

Таблица 6. Основные типы атрибутов.

Атрибут стандартной информации \$STANDARD_INFORMATION

Атрибут стандартной информации описывает время создания/изменения/последнего доступа к файлу и права доступа, а также некоторую другую вспомогательную информацию (например, квоты):

Смещение	Размер	ОС	Описание
~ ~		любая	стандартный атрибутный заголовок (standard attribute header)
00h	8	любая	С время создания (creation) файла
08h	8	любая	А время изменения (altered) файла
10h	8	любая	М время изменения файловой записи (MFT changed)
18h	8	любая	Р время последнего чтения (read) файла
20h	4	любая	права доступа MS-DOS (MS-DOS file permissions) Значение Описание 0001h только на чтение (read-only) 0002h скрытый (hidden) 0004h системный (system) 0020h архивный (archive) 0040h устройство (device) 0080h обычный (normal) 0100h временный (temporary) 0200h разряженный (sparse) файл 0400h reparse point 0800h сжатый (compressed) 1000h оффлайновый (offline) 2000h не индексируемый (not content indexed) 4000h зашифрованный (encrypted)
24h	4	любая	старшее двойное слово номера версии (maximum number of versions)
28h	4	любая	младшее двойное слово номера версии (version number)
2Ch	4	любая	идентификатор класса (class ID)

30h	4	2K	идентификатор владельца (owner ID)
34h	4	2K	идентификатор безопасности (security ID)
38h	8	2K	количество котируемых байт (quota charged)
40h	8	2K	номер последней последовательности обновления (update sequence number USN)

Таблица 7. Структура атрибута \$STANDARD_INFORMATION.

Атрибут списка атрибутов \$ATTRIBUTE_LIST

Атрибут списка атрибутов (прямо каламбур) используется в тех случаях, когда все атрибуты файла не умещаются в базовой файловой записи и файловая система вынуждена располагать их в расширенных. Индексы расширенных файловых записей содержатся в атрибуте списка атрибутов, помещаемом в базовую файловую запись.

При каких обстоятельствах атрибуты не умещаются в одной файловой записи? Это может произойти когда: а) файл содержит много альтернативных имен или жестких ссылок; б) файл очень-очень сильно фрагментирован; в) файл содержит очень сложный дескриптор безопасности; г) файл имеет очень много потоков данных (т.е. атрибутов типа \$DATA).

Структура атрибута списка атрибутов приведена ниже:

Смещение	Размер	Описание
~ ~		стандартный атрибутный заголовок (standard attribute header)
00h	4	тип (type) атрибута (см. таблицу 6)
04h	2	длина записи (record length)
06h	1	длина имени (name length), или ноль, если нет; условно - N
07h	1	смещение имени (offset to name), или ноль если нет
08h	8	начальный виртуальный кластер (starting VCN)
10h	8	ссылка на базовую/расширенную файловую запись
18h	2	идентификатор атрибута (attribute ID)
1Ah	2N	if N > 0, то имя в формате UNICODE

Таблица 8. Структура атрибута \$ATTRIBUTE_LIST.

Атрибут полного имени \$FILE_NAME

Атрибут полного имени файла хранит имя файла в соответствующем пространстве имен. Таких атрибутов у файла можно быть и несколько (например, win32-имя и MS-DOS имя). Здесь же хранятся и жесткие ссылки (hard link), если они есть.

Структура атрибута полного имени приведена ниже:

Смещение	Размер	Описание
~ ~		стандартный атрибутный заголовок (standard attribute header)
00h	8	ссылка (file reference) на материнский каталог
08h	8	C - время создания (creation) файла
10h	8	A - время последнего изменения (altered) файла
18h	8	M - время последнего изменения файловой записи (MFT changed)
20h	8	R - время последнего чтения (read) файла
28h	8	выделенный размер (allocated size) файла
30h	8	реальный размер (real size) файла
38h	4	флаг (см. таблицу 9)
3Ch	4	используется HPFS
40h	1	длина имени в символах - L
41h	1	пространство имен файла (filename namespace)
42h	2L	имя файла в формате UNICODE без завершающего нуля

Таблица 9. Структура атрибута \$FILE_NAME.

Списки отрезков (data runs)

Тела нерезидентных атрибутов хранятся на диске в одной или нескольких кластерных цепочках, называемых **отрезками (runs)**. Отрезком называется последовательность смежных кластеров, характеризующаяся номером начального кластера и длиной. Совокупность отрезков называется списком, **run-list'ом** или **data run'ом**.

Внутренний формат представления списков не то, чтобы сложен, но явно не прост, за что получил прозвище brain damage format'a (формата, срывающего крышу и обламывающего кайф). Для экономии места длина отрезка и номер начального кластера хранятся в полях переменной длины. То есть, если размер отрезка умещается в байт (т.е. его значение не превышает 255), он и хранится в байте. Соответственно, если размер отрезка требует для своего представления двойного слова, он и хранится в двойном слове.

Сами же поля размеров хранятся в 4-байтовых ячейках, называемых **нибблами (nibble)** или **полубайтами**. Шестнадцатеричная система исчисления позволяет легко переводить байты в нибблы и наоборот. Младший ниббл равен (X & 15), а старший - (X / 16). Легко видеть, что младший ниббл соответствует младшему шестнадцатеричному разряду байта, а старший - старшему. Например, 69h состоит из двух нибблов - младший равен 9h, а старший - 6h.

Список отрезков представляет собой массив структур, каждая из которых описывает характеристики "своего" отрезка, а в конце списка находится завершающий нуль. Первый байт структуры состоит из двух полубайтов: младший задает длину поля

начального кластера отрезка (условно обозначаемого буквой F), старший - количество кластеров в отрезке (L). Поле длины отрезка идет следом. В зависимости от значения L оно может занимать от одного до восьми байт (более длинные поля недопустимы). Первый байт поля стартового кластера файла расположен по смещению $1 + L$ байт от начала структуры (что соответствует $2 + 2 * L$ нибблам). Кстати говоря, в документации Linux-NTFS Project (версия 0.4) поля размеров начального кластера и количества кластеров в отрезке перепутаны местами.

Смещение в нибблах	Размер в нибблах	Описание
0	1	размер поля длины (L)
1	1	размер поля начального кластера (S)
2	2*L	количество кластеров в отрезке
2+2*L	2*S	номер начального кластера отрезка

Таблица 10. Структура одного элемента списка отрезков.

Покажем, как с этим работать на практике. Допустим, мы имеем следующий run-list, соответствующий нормальному нефрагментированному файлу (что может быть проще!): **"21 18 34 56 00"**. Попробуем его декодировать?

Начнем с первого байта - 21h. Младший полубайт (01h) описывает размер поля длины отрезка, старший (02h) - размер поля начального кластера. Следующие несколько байт представляют поле длины отрезка, размер которого в данном случае равен одному байту - 18h. Два других байта (34h 56h) задают номер начального кластера отрезка. Нулевой байт на конце сигнализирует о том, что это последний отрезок в файле. Итак, наш файл состоит из одного-единственного отрезка, начинающегося с кластера 5634h и заканчивающегося кластером $5634h + 18h = 564Ch$.

Рассмотрим более сложный пример фрагментированного файла со следующим списком отрезков: **"31 38 73 25 34 32 14 01 E5 11 02 31 42 AA 00 03 00"**. Извлекаем первый байт - 31h. Один байт приходится на поле длины и три байта на поле начального кластера. Таким образом, первый отрезок (run 1) начинается с кластера 342573h и продолжается вплоть до кластера $342573h + 38 = 3425ABh$. Чтобы найти смещение следующего отрезка в списке, мы складываем размер обоих полей с их начальным смещением: $3 + 1 = 4$. Отсчитываем четыре байта от начала run-list'a и переходим к декодированию следующего отрезка: 32h - два байта на поле длины отрезка (равное в данном случае 0114h) и три байта на поле номера начального кластера (0211E5h). Следовательно, второй отрезок (run 2) начинается с кластера 0211E5h и продолжается вплоть до кластера $0211E5h + 114h = 212F9h$. Третий отрезок (run 3): 31h - один байт на поле длины и три байта на поле начального кластера, равные 42h и 0300AAh соответственно. Поэтому третий отрезок (run 3) начинается с кластера 0300AAh и продолжается вплоть до кластера $0300AAh + 42h = 300ECh$. Завершающий нуль на конце run-list'a сигнализирует о том, что это последний отрезок в файле.

Таким образом, подопытный файл состоит из трех отрезков, разбросанных по диску в следующем живописном порядке: 342573h - 3425ABh; 0211E5h - 212F9h; 0300AAh - 300ECh. Остается только прочитать его с диска!

Начиная с версии 3.0, NTFS поддерживает разряженные (sparse) атрибуты, т.е. такие, которые не записывают на диск кластеры, содержащие одни нули. При этом поле номера начального кластера отрезка может быть равным нулю, что означает, что данному отрезку не выделен никакой кластер. Поле длины содержит количество кластеров, заполненных нулями. Их не нужно считывать с диска. Вы должны самостоятельно изготовить их в памяти. Кстати говоря, далеко не все дисковые доктора знают о существовании разряженных атрибутов (если атрибут разряжен, его флаг равен 8000h), и интерпретируют нулевую длину поля номера начального кластера весьма странным образом. Последствия такого "лечения" обычно оказываются очень печальными.

Пространства имен (name spaces)

NTFS изначально проектировалась как системно-независимая файловая система, способная работать со множеством различных подсистем: win32, MS-DOS, POSIX и т.д. Поскольку каждая из них налагает свои собственные ограничения на набор символов, допустимых для использования в имени файла, NTFS вынуждена поддерживать несколько независимых пространств имен (name space).

POSIX

Допустимы все UNICODE-символы (с учетом регистра), за исключением символа нуля (NULL), обратного слеша ('/') и знака двоеточия (':'). Последнее, кстати говоря, не ограничение POSIX, а ограничение NTFS, использующей этот символ для доступа к именованным атрибутам. Максимально допустимая длина имени составляет 255 символов.

Win32

Доступны все UNICOE-символы (без учета регистра), за исключением следующего набора: '"' (кавычки), '*' (звездочка), '/' (прямой слеш), ':' (двоеточие), '<' (знак меньше), '>' (знак больше), '?' (вопросительный знак), '\' (обратный слеш), '|' (символ конвейера). К тому же, имя файла не может заканчиваться на точку или пробел. Максимально допустимая длина имени составляет 255 символов.

MS-DOS

Доступны все символы пространства имен win32 (без учета регистра), за исключением: '+' (знак плюс), ',' (знак запятая), '.' (знак точка), ';' (точна с запятой), '=' (знак равно). Имя файла не должно превышать восьми символов, за которыми следует необязательное расширение с длиной от одного до трех символов.

Назначение некоторых служебных файлов

NTFS содержит большое количество служебных файлов (метафайлов) строго определенного формата, важнейший из которых - \$MFT мы только что рассмотрели. Остальные метафайлы играют вспомогательную роль и для восстановления данных знать их структуру, в общем-то, и необязательно. Тем не менее, если они окажутся искажены, штатный драйвер файловой системы не сможет работать с таким томом, поэтому иметь некоторые представления о назначении каждого из них все же необходимо.

У нас нет возможности рассказать о структуре всех метафайлов (да и незачем дублировать Linux-NTFS Project), поэтому эта информация здесь не приводится.

inode	Имя файла	ОС	Описание
0	\$MFT	любая	главная файловая таблица (Master File Table, MFT)
1	\$MFTMirr	любая	резервная копия первых четырех элементов 4 MFT
2	\$LogFile	любая	журнал транзакций (transactional logging file)
3	\$Volume	любая	серийный номер, время создания, dirty flag (флаг не сброшенного кэша) тома

4	\$AttrDef	любая	определение атрибутов
5	. (точка)	любая	корневой каталог (root directory) тома
6	\$Bitmap	любая	карта свободного/занятого пространства
7	\$Boot	любая	загрузочная записи (boot record) тома
8	\$BadClus	любая	список плохих кластеров (bad clusters) тома
9	\$Quota	NT	информация о квотах (quota information)
9	\$Secure	2K	использованные дескрипторы безопасности (security descriptors)
10	\$UpCase	любая	таблица заглавных символов (uppercase characters) для трансляции имен
11	\$Extend	2K	каталоги: \$ObjId, \$Quota, \$Reparse, \$UsnJrnl
12-15	не используется	любая	помечены как использованные, но в действительности - пустые
16-23	не используется	любая	помечены как неиспользуемые
любой	ObjId	2K	уникальные идентификаторы каждого файла
любой	\$Quota	2K	информация о квотах (quota information)
любой	\$Reparse	2K	информация типа reparse point
любой	\$UsnJrnl	2K	журнал шифрованной файловой системы (journaling of encryption)
> 24	польз. файл	любая	обычные файлы
> 24	польз. каталог	любая	обычные каталоги

Таблица 11. Назначение основных стандартных файлов.

Путешествие по NTFS

Рассказ о NTFS был бы неполным без практической иллюстрации техники разбора файловой записи "руками". До сих пор мы витали в облаках теоретической абстракции. Пора спускаться на грешную землю.

Воспользовавшись любым дисковым редактором, например Disk Probe, попробуем декодировать одну файловую запись вручную. Найдем сектор, содержащий сигнатуру "FILE" в его начале (не обязательно брать первый встретившийся сектор). Он может выглядеть, например, так:

	:	00	01	02	03	04	05	06	07		08	09	0A	0B	0C	0D	0E	0F	
00000000:	46	49	4C	45	2A	00	03	00	00		60	79	1A	04	02	00	00	00	FILE* `y> O
00000010:	01	00	01	00	30	00	01	00	00		50	01	00	00	00	04	00	00
00000020:	00	00	00	00	00	00	00	00	00		04	00	03	00	00	00	00	00
00000030:	10	00	00	00	60	00	00	00	00		00	00	00	00	00	00	00	00
00000040:	48	00	00	00	18	00	00	00	00		B0	D5	C9	2F	C6	0B	C4	01
00000050:	E0	5A	B3	7B	A9	FA	C3	01	00		90	90	F1	2F	C6	0B	C4	01

```

00000060: 50 7F BC FE C8 0B C4 01 | 20 00 00 00 00 00 00 00 .....
00000070: 00 00 00 00 00 00 00 00 | 00 00 00 00 05 01 00 00 .....
00000080: 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 .....
00000090: 30 00 00 00 70 00 00 00 | 00 00 00 00 00 00 02 00 .....
000000A0: 54 00 00 00 18 00 01 00 | DB 1A 01 00 00 00 01 00 .....
000000B0: B0 D5 C9 2F C6 0B C4 01 | B0 D5 C9 2F C6 0B C4 01 .....
000000C0: B0 D5 C9 2F C6 0B C4 01 | B0 D5 C9 2F C6 0B C4 01 .....
000000D0: 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 .....
000000E0: 20 00 00 00 00 00 00 00 | 09 03 49 00 6C 00 66 00 .....
000000F0: 61 00 6B 00 2E 00 64 00 | 62 00 78 00 00 00 00 00 .....
00000100: 80 00 00 00 48 00 00 00 | 01 00 00 00 00 00 03 00 .....
00000110: 00 00 00 00 00 00 00 00 | ED 04 00 00 00 00 00 00 .....
00000120: 40 00 00 00 00 00 00 00 | 00 E0 4E 00 00 00 00 00 .....
00000130: F0 D1 4E 00 00 00 00 00 | F0 D1 4E 00 00 00 00 00 .....
00000140: 32 EE 04 D9 91 00 00 81 | FF FF FF FF 82 79 47 11 .....
000001F0: 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 03 00 .....
      : 00 01 02 03 04 05 06 07 | 08 09 0A 0B 0C 0D 0E 0F .....

```

Листинг 4. Ручное декодирование файловой записи (разные атрибуты выделены разным цветом).

Первым делом необходимо восстановить оригинальное содержимое последовательности обновления. По смещению 04h от начала сектора лежит 16-разрядный указатель на нее, равный в данном случае 2Ah (значит, это NTFS 3.0 или младше). А что у нас лежит по смещению 2Ah? Ага, пара байт 03 00. Это - номер последовательности обновления. Сверяем его с содержимым двух последних байт этого и следующего секторов (смещения 1FEh и 3FEh соответственно). Они равны! Значит, данная файловая запись цела (по крайней мере, внешне) и можно переходить к операции спасения. По смещению 2Ch расположен массив, содержащий оригинальные значения последовательности обновления. Количество элементов в нем равно содержимому 16-разрядного поля, расположенному по смещению 06h от начала сектора и уменьшенного на единицу (т.е. в данном случае 03h - 01h = 02h). Извлекаем два слова, начиная со смещения 2Ch (в данном случае они равны 00 00 и 00 00), и записываем их в конец первого и последнего секторов.

Теперь нам необходимо выяснить - используется ли данная файловая запись или ассоциированный с ней файл/каталог был удален. 16-разрядное поле, расположенное по смещению 16h, содержит значение 01h. Следовательно, перед нами файл, а не каталог и этот файл еще не удален. Но является ли данная файловая запись базовой для данного файла или мы имеем дело с ее продолжением? 64-разрядное поле, расположенное по смещению 20h, равно нулю, следовательно данная файловая запись - базовая.

Ок, переходим к исследованию атрибутов. 16-разрядное поле, находящееся по смещению 14h равно 30h, следовательно заголовок первого атрибута начинается со смещения 30h от начала сектора.

Первое двойное слово атрибута равно 10h, значит перед нами атрибут типа \$STANDARD_INFORMATION. 32-разрядное поле длины атрибута, находящееся по смещению 04h и равное в данном случае 60h байт, позволяет нам вычислить смещение следующего атрибута в списке - 30h (смещение нашего атрибута) + 60h (его длина) = 90h (смещение следующего атрибута). Первое двойное слово следующего атрибута равно 30h, значит, это атрибут типа \$NAME и следующее 32-разрядное поле хранит его длину, равную в данном случае 70h. Сложив длину атрибута с его смещением, мы получим смещение следующего атрибута - 90h + 70h = 160h. Первое двойное слово третьего атрибута равно 80h, следовательно это атрибут типа \$DATA, хранящий основные данные файла. Складываем его смещение с длиной - 160h + 32h = 192h. Упс! Мы наткнулись на частотол FFFFFFFh, сигнализирующий о том, что атрибут \$DATA последний в списке.

Теперь, разбив файловую запись на атрибуты, как мясник рассекает телячью тушу (try the veil, как говаривал старина Шрек), не грех будет приступить к исследованию каждого из атрибутов в отдельности. Начнем с имени. 8-разрядное поле, находящееся по смещению 08h от начала атрибутного заголовка (и по смещению 98h от начала сектора), содержит флаг нерезидентности, который в данном случае равен нулю (т.е. атрибут резидентный и его тело хранится непосредственно в самой файловой записи, что есть гуд). 16-разрядное поле, расположенное по смещению 0Ch от начала атрибутного заголовка (и по смещению 9Ch от начала сектора) равно нулю, следовательно тело атрибута не сжато и не зашифровано. Хорошо! Тогда подать это тело на стол! 32-разрядное поле, расположенное по смещению 10h от начала атрибутного заголовка и по смещению A0h от начала сектора, содержит длину атрибутного тела, равную в данном случае 54h байт, а 16-разрядное поле, расположенное по смещению 14h от начала атрибутного заголовка и по смещению A4h от начала сектора, хранит смещение атрибутного тела, равное в данном случае 18h, следовательно тело атрибута \$NAME располагается по смещению A8h от начала сектора.

Формат атрибута типа \$NAME описан выше в таблице. Первые восемь байт содержат ссылку на материнский каталог данного файла, равную в данном случае 11ADBh:01 (индекс - 11ADBh, номер последовательности - 01h). Следующие 32-байта содержат штампы времени создания, изменения и времени последнего доступа к файлу. По смещению 28h от начала тела атрибута и D0h от начала сектора лежит 64-разрядное поле выделенного размера, а за ним - 64-разрядное поле реального размера. Оба равны нулю, что означает, что за размером файла следует обращаться к атрибутам типа \$DATA.

Длина имени файла содержится в 8-разрядном поле, находящемся по смещению 40h байт от начала тела атрибута и по смещению E8h от начала сектора. В данном случае оно равно 09h. Само же имя начинается со смещения 42h от начала тела атрибута и со смещения EAh от начала сектора. И здесь находится Ifak.dbx.

Переходим к атрибуту основных данных файла, пропустив атрибут стандартной информации, который не содержит решительно ничего интересного. 8-разрядный флаг нерезидентности, расположенный по смещению 08h от начала атрибутного заголовка и по смещению 108h от начала сектора, равен 01h, следовательно атрибут нерезидентный. 16-разрядный флаг, расположенный по смещению 0Ch от начала атрибутного заголовка и по смещению 10Ch от начала сектора, равен нулю, значит, атрибут не сжат и не зашифрован. 8-разрядное поле, расположенное по смещению 09h от начала атрибутного заголовка и по смещению 109h от начала сектора, равно нулю - атрибут безымянный. Реальная длина тела атрибута (в байтах) содержится в 64-разрядном поле, расположенном по смещению 30h от начала атрибутного заголовка и по смещению 130h от начала сектора. В данном случае она равна 4ED1F0h (5.165.552). Два 64-разрядных поля, расположенных по смещениям 10h/110h и 18h/118h байт от начала атрибутного заголовка/сектора соответственно, содержат начальный и конечный номер виртуального кластера нерезидентного тела. В данном случае они равны: 0000h/4EDh.

Остается лишь декодировать список отрезков, адрес которого хранится в 16-разрядном поле, находящемся по смещению 20h от начала атрибутного заголовка и 120h от начала сектора. В данном случае оно равно 40h, что соответствует смещению от начала сектора в 140h. Сам же список отрезков выглядит так: 32 EE 04 D9 91 00 00. Ага! Два байта занимает поле длины (равное в данном случае 04EEh кластерам) и три - поле начального кластера (0091h). Завершающий ноль на конце говорит о том, что этот отрезок последний в списке отрезков.

Подытожим полученную информацию. Файл называется Ifak.dbx, он начинается с кластера 0091h и продолжается вплоть до кластера 57Fh при реальной длине файла в 5.165.552 байт. За сим - все! Теперь уже ничего не стоит скопировать файл на резервный носитель (например, ZIP или стример).

Заключение

Вооруженные джентльменским набором знаний (а также дисковым редактором впридачу), мы готовы дать решительный отпор энтропии, потеснив ее по всем фронтам. Следующая статья этого цикла расскажет о том, как восстанавливать удаленные файлы, отформатированные разделы и разрушенные служебные структуры данных.

