путь воина – внедрение в pe/coff файлы

крис касперски

статья подробно описывает формат PE-файлов, раскрывая особенности внутренней кухни системного загрузчика и двусмысленности фирменной спецификации, предостерегая читателя от многочисленных ловушек, подстерегающих его пути внедрения своего кода в чужие исполняемые файлы. здесь вы найдете большое количество исходных текстов, законченных решений и наглядных примеров, упрощающих восприятие материала (а то вот некоторые жалуются, что мои статьи слишком сложные и непроходимые, словно таежный лес, в то время как другие обвиняют их в чрезмерной простоте и откровенной "ламеризации" ой, простите, "юзеризации" материала). статья ориентирована главным образом на Windows NT/9x и производные от них системы, но так же затрагивает и проблему совместимости с Windows-эмуляторами, такими, например, как wine и doswin32.

...хакерство вытеснило все – голод, интерес к девушкам, друзей, учебу, родителей, смысл жизни. Это был дракон, сжигающий все на своем пути, оставляющий лишь запах напалма и смутные картинки прошлого в памяти. Когда я включал компьютер, я испытывал чувства знакомые, наверное, только заядлому наркоману, который наконец ширнулся после двух-месячного "голода"...

аноним

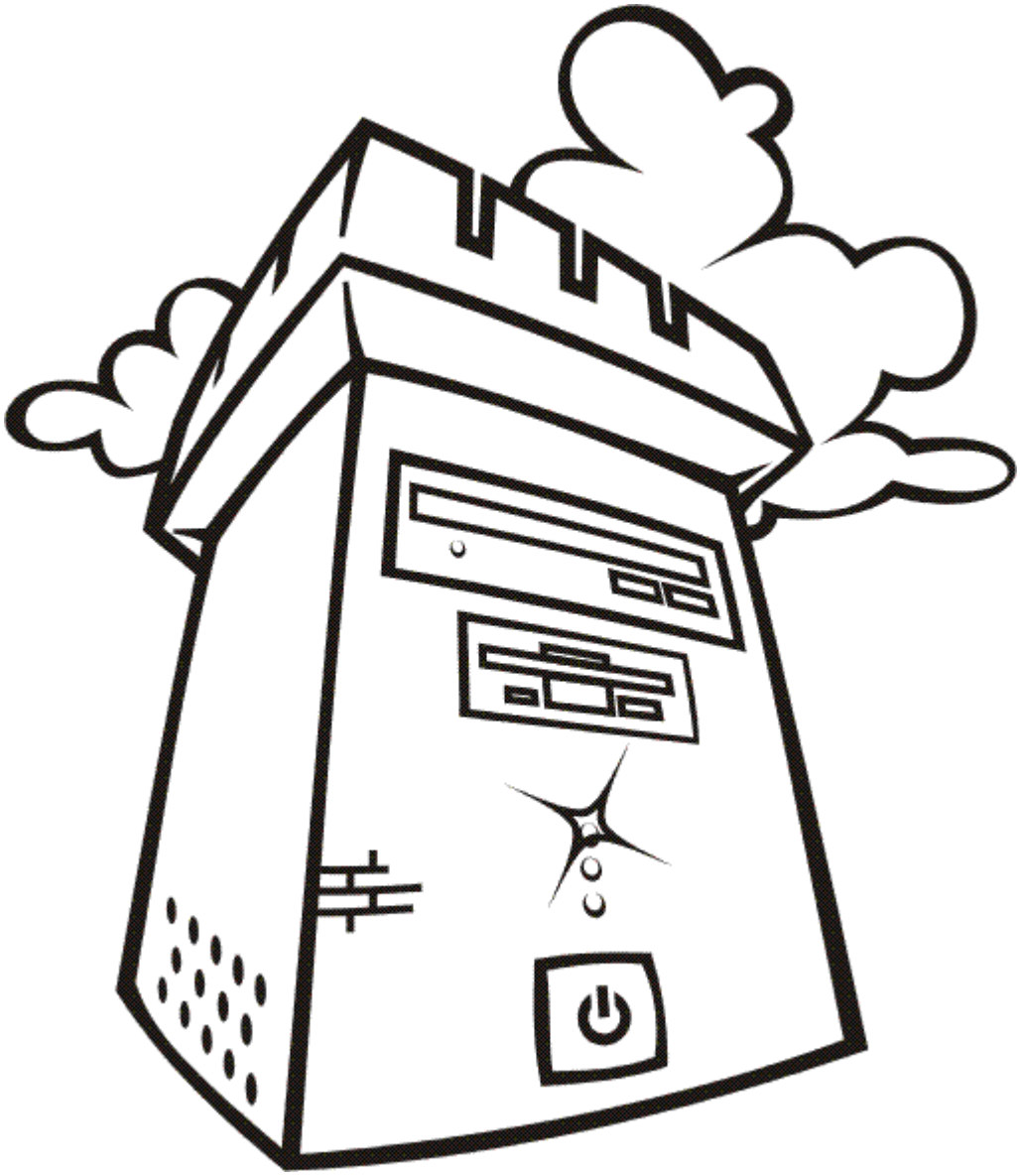

Рисунок 1 рисунок Олега Морозова

введение

После публикации статьи, посвященной UNIX-вирусам, ко мне стали приходить письма с просьбами написать "точно такую же, но только под Windows". Действительно, внедрение постороннего кода в PE-файлы – очень перспективное и нетривиальное занятие, интересное не только вирусописателям, но и создателям навесных протекторов/упаковщиков в том числе.

Что же до этической стороны проблемы… политика "воздержания" и удержания передовых технологий под сукном лишь увеличивает масштабы вирусных эпидемий и когда дело доходит до схватки, никто из прикладных программистов (и администраторов!) к ней оказывается не готов. В стане системных программистов дела обстоят еще хуже. Исходных текстов операционной системы нет, PE-формат документирован кое-как, поведение системного загрузчика вообще не подчиняется никакой логике, а допрос с пристрастием (читай – дизассемблирование) еще не гарантирует, что остальные загрузчики поведут себя точно также.

На сегодняшний день не существует ни одного более или менее корректного упаковщика/протектора под Windows, в полной мере поддерживающего фирменную спецификацию и учитывающего недокументированные особенности поведения системных загрузчиков в операционных системах Windows 9x/NT. Про различные эмуляторы, такие, например, как wine, doswin32, мы лучше промолчим, хотя нас так и подмывает сказать, что файлы, упакованные ASPack в среде doswin32 либо не запускаются вообще, либо работают

крайне нестабильно, а все потому что упаковщик ASPack не соответствует спецификации, закладываясь на те особенности системного загрузчика, преемственности которых никто и никому не обещал. В лучшем случае, авторы эмуляторов после жуткого трехэтажного мата, добавляют в свои продукты обходной код, предназначенный для обработки подобных извращений, в худшем же – оставляют все как есть, мотивируя это словами "повторять чужое пионерство себе дороже…"

А восстановление пораженных объектов? Многие файлы после заражения отказывают в работе и попытка вылечить их антивирусом лишь усугубляет ситуацию. Всякий, уважающий себя профессионал, должен быть готов вычистить вирус вручную, не имея под рукой ничего, кроме hex-редактора! Тоже самое относится и к снятию упаковщиков/дезактивации навесных протекторов. Эй! Кто там начал бурчать про злобных хакеров и неэтичность взлома? Помилуйте, что за фрейдистские ассоциации?! Ну нельзя же всю жизнь что-то ломать (надо на чем-то и сидеть!). В майском номере "Системного Администратора" за 2004 год опубликована замечательная статья Андрея Бешкова, живописно описывающая трах с протекторами под эмулятором wine. Как говориться, тут не до жиру – быть бы живу. Какой смысл платить за регистрацию, если воспользоваться защищенной программой все равно не удастся?!

Собственно говоря, всякое вмешательство в структуру готового исполняемого файла – мероприятие достаточно рискованное и шанс сохранить ему работоспособность на всех платформах достаточно невелик. Однако, если вы все-таки, что это вам необходимо, пожалуйста, отнеситесь к проектированию внедряемого кода со всей серьезностью и следуйте рекомендациям, данным в этой статье.

Широта охватываемых тем не позволила рассказать обо всем в одной статье и ее пришлось разбить на две части – та, которую вы сейчас держите в руках, посвящена описанию малоизвестных особенностей PE-файлов, без знания которых, свой упаковщик/протектор ни за что не написать (по крайней мере *работоспособный* упаковщик/протектор – точно). А конкретные механизмы внедрения чужеродного кода мы рассмотрим в следующий раз.

особенности структуры PE-файлов в конкретных реализациях

Знакомство читателя с PE-форматом не входит в нашу задачу и предполагается, что некоторый опыт работы с ними у него уже имеется. Существует множество описаний PE-формата, но среди них нет ни одного по настоящему хорошего. Официальная спецификация (**Microsoft Portable Executable and Common Object File Format Specification**), написанная двусмысленным библейским языком, скорее напоминает сферического коня в вакууме, чем практическое руководство. Даже среди сотрудников Microsoft нет единого мнения по поводу, как именно следует его толковать, и различные системные загрузчики ведут себя сильно неодинаково. Что же касается сторонних разработчиков, то здесь и вовсе царит полная неразбериха.

Понимание структуры готового исполняемого файла еще не наделяет вас умением самостоятельно собирать такие файлы вручную. Операционные системы облагают нас весьма жесткими ограничениями, зачастую вообще не упомянутыми в документации и варьирующиеся от одной оси к другой. Не так-то просто создать файл, загружающийся больше чем на одной машине (которой, как правило, является машина его создателя). Один шаг в сторону – и загрузчик открывает огонь без предупреждения, выдавая малоинформативное сообщение в стиле "*файл не является win32 приложением*", после чего остается только гадать: что же здесь неправильно (кстати говоря, Windows 9x намного более подробно диагностирует ошибку, чем Windows NT, если, конечно, некорректный файл не вгонит ее в крутой завис, а виснет она на удивление часто – загрузчик там писали пионеры не иначе).

Технические писатели, затрагивающие тему исполняемых файлов, и совершенно не разбирающиеся в предметной области за которую взялись, за неимением лучших идей прибегают к довольно грязному трюку и подменяют одну тему другой. Отталкиваясь от уже существующих PE-файлов, созданных линкером, они долго и занудно объясняют назначение каждого из полей, демонстративно прогуливаясь по ссылочным структурам от вершины до дна. Ха! Это и макака сумеет! Сложнее разобраться почему эти структуры сконструированы именно так, а и иначе. Какой в них заложен запас прочности? Каким именно образом их интерпретирует системный загрузчик? А что на счет предельно допустимых значений? Увы, все эти вопросы остаются без ответа. Чтение статьей в стиле "**The Portable Executable File Format from Top to Bottom**" от Randy Kath'a из Microsoft Developer Network Technology Group это хороший способ запудрить себе мозги и написать мертворожденный PE-дампер, переваривающий только

"честные" файлы и падающий на всех остальных (dumpbin ведь падает!). Аналогичным образом поступает и Matt Pietrek, обходящий базовые концепции PE-файла стороной и начинающий процесс описания с середины, но так и не доводящий его до логического конца.

Иначе поступает автор статьи "**Об упаковщиках в последний раз**" некто Volodya (http://www.wasm.ru/print.php?article=packlast01 и http://www.wasm.ru/print.php?article=packers2), сосредоточивший свои усилия на исследовании системного загрузчика W2K/XP и допустивший при этом большое количество фактических ошибок, полный разбор которых потребовал бы отдельной статьи. При всей ценности этой работы, она нисколько не проясняет ситуацию и только добавляет вопросов. Volodya сетует на то, что работа загрузчика полностью недокументированна и даже у Руссиновича обнаруживаются лишь обрывки информации. Ну была бы она документирована – чтобы от этого изменилось? Какое нам дело до того, что в W2K/XP загрузка файла сводится к вызову MmCreateSection? Во-первых, в остальных системах это не так, а во-вторых, это сегодня Microsoft стремиться весь ввод/вывод делать через mmap, но когда до горячих американских парней дойдет, что это тормоза, а не ускорение, политика измениться и MmCreateSection отправятся на заслуженный отдых (в чулан ее! на полку!)

Дизассемблировать ядро совсем небесполезно, но вот закладываться на полученные результаты, не проверив их на остальных осях, ни в коем случае нельзя! Верить в спецификации по меньшей мере наивно, ведь всякая спецификация – это только слова, а всякий программный код – лишь частный случай реализации. И то, и другое непостоянно и переменчиво. Чтение книжек (равно как и протирание штанов в учебных заведениях различной степени тяжести) еще никого не сделало программистом. Лишь опыт – сын ошибок трудных, – да общение с коллегами-системщиками, позволят избежать грубых ошибок[1]. Как говориться, не падает только тот кто лежит, а кто бежит – падает, наступает на грабли и попадает в логические ямы, глубокие как колодцы из романа Мураками (кстати говоря, колодец во многих религиях символизировал связь с потусторонним миром).

Автор, имеющий богатый опыт сексуальных извращений с PE-файлами и помнящий численные значения смещений всех структур как отче наш, в процессе работы над статьей в такие колодный попадал неоднократно (да и сейчас там сидит). Всякое значение подобно больному зубу, – если его не трогать, он не будет ныть. Отдельные пробелы, неясности и непонятности неизбежны. Когда пишешь рабочие заметки "для себя", просто махаешь рукой и говоришь: да какая разница, что этот большой красный рубильник делает? Работает ведь – и ладно… Статья – дело другое и тут хочешь не хочешь, а будь добр разложить все по полочкам! Автор выражает глубокую признательность удивительному человеку, мудрому программисту и создателю замечательного ликера ulink Юрию Харону, помогающему таким тупым дурням как я преодолевать мифическую реку Стикс (ftp://ftp.styx.cabel.net/pub/UniLink/) и терпеливо отвечавшему на мои сумбурные и нечетко сформулированные вопросы. Если бы не его консультации, эта статья ни за что бы не получилось такой, какова она есть!

общие концепции и требования, предъявляемые к PE-файлам

Структурно PE-файл состоит из ***заголовка*** (*header*), ***страничного имиджа*** (*image page*) и необязательного ***оверлея*** (*overlay*). Представление PE-файла в памяти называется его ***виртуальным образом*** (***virtual image***) или просто образом, а на диске – файлом или дисковым образом. Если не оговорено обратное, то под образом всегда понимается виртуальный образ.

Образ характеризуется двумя фундаментальными – *адресом базовой загрузки* (*image base*) и *размером* (*image size*). При наличии перемещаемой информации (relocation/fixup table), образ может быть загружен по адресу отличному от image base и назначаемому непосредственно самой операционной системой.

Образ естественным образом делится на *страницы* (*pages*), а файл – на *сектора* (*sectors*). Виртуальный размер страниц/секторов явным образом задается в заголовке файла и не обязательно должен совпадать с физическим[2].

---

[1] опыт – чудесная вещь, он позволяет вам узнавать свою ошибку в тех случаях, когда вы ее допускаете снова и снова (с) неизвестен

[2] строго говоря, никаких виртуальных страниц/секторов нет и не вздумайте никому о них говорить, чтобы не подняли на смех, терминология это моя, авторская. Правильнее говорить о минимальной порции (кванте) данных, равной выбранной кратности выравнивания на диске и в памяти, но постоянно набивать такое на клавиатуре слишком длинно и утомительно. Короче говоря – я вас предупредил. По какому праву я ввел свою терминологию? Дык, моя селедка

Системный загрузчик требует от образа непрерывности, документация же обходит этот вопрос стороной. На всем протяжении между image base и (image base + size of image) не должно присутствовать ни одной бесхозной страницы, не принадлежащей ни заголовку, ни секциям – такой файл просто не будет загружен. (С этим не совсем согласен Юрий Харон, однако, ни одного "прерывистого" файла выловить в живой природе мне не удалось, а попытка создать таковой самостоятельно всякий раз заканчивалась неизменным неуспехом). Бесхозных же секторов в любой части файла может быть сколько угодно. Каждый сектор может отображается на любое количество страниц (по одной странице за раз), но никакая страница не может отображать на один и тот же регион памяти более одного сектора.

Для работы с PE-файлами используются три различных схемы адресации: ***физические адреса*** (называемые так же сырыми указателями или смещениями raw pointers/raw offset или просто offset), отсчитываемые от начала файла; ***виртуальные адреса*** (*virtual address* или сокращенное **VA**), отсчитываемые от начала адресного пространства процесса и ***относительные виртуальные адреса*** (*relative virtual address* или сокращенно **RVA**), отсчитываемые от базового адреса загрузки. Все трое измеряются в байтах и хранятся в 32-битных указателях (в PE64 все указатели 64-битные, но где мы, а где PE64?). Параграфы давно вышли из моды, а жаль… Вообще-то, существует и четвертый тип адресации – **RRA**, что расшифровывается как ***R****aw* ***R****elative* ***A****ddress* (*сырые относительные адреса*) или ***R****elative* ***R****elative* ***A****ddress* (*относительно относительные адреса*). Терминология вновь моя, ибо официального названия у такого способа адресации нет и не предвидится (ну прямо как в анекдоте: странно: ж. есть, а слова нет). Иногда его называют offset'ом, что не совсем верно, т. к. offset'ы они сильно разные бывают, а RRVA-адреса всегда отсчитываются от стартового адреса своей структуры (в частности, OffsetModuleName задает смещение от начала таблицы диапазонного импорта).

Страничный имидж состоит из одной или нескольких ***секций***. С каждой секций связано четыре атрибута: физический адрес начала секции в файле/размер секции в файле, виртуальный адрес секции в памяти/размер секции в памяти и атрибут характеристик секции, описывающий правда доступа, особенности ее обработки системным загрузчиком и т. д. Грубо говоря, секция вправе сама решать откуда и куда ей грузиться, однако, эта свобода весьма условна и на ассортимент выбираемых значений наложено множество ограничений. Начало каждой секции в памяти/диске всегда совпадает с началом виртуальных страниц/секторов соответственно. Попытка создать секцию, начинающуюся с середины, жестоко пресекается системным загрузчиком, отказывающимся обрабатывать такой файл. С концом складывается более демократичная ситуация и загрузчик не требует, чтобы виртуальный (и частично физический) размер секций был кратен размеру страницы. Вместо этого он самостоятельно выравнивает секции, забивая их хвост нулями, так что никакая страница (сектор) не может принадлежать двум и более секциям сразу. Фактически это сплошное надувательство – не выровненный (в заголовке!) размер автоматически выравнивается в страничном имидже, поэтому представленные нам полномочия на проверку оказываются сплошной фикцией.

Все секции совершенно равноправны и тип каждой из них тесно связан с ее атрибутами, интерпретируемыми довольно неоднозначным и противоречивым образом (см. "таблица секций"). Реально (читай – на сегодняшний день) мы имеем два аппаратных и два программных атрибута: ***Accessible/Writeable*** и ***Shared/Loadable*** (последний – условно) соответственно. Вот отсюда и следует плясать! Все остальное – из области абстрактных концепций.

"Секция кода", "секция данных", "секция импорта" – не более чем образные выражения, своеобразный рудимент старины, оставшийся в наследство от сегментной модели памяти, когда код, данные и стек действительно находились в различных сегментах, а не были сведены в один, как это происходит сейчас.

Служебные структуры данных (таблицы экспорта, импорта, перемещаемых элементов), могут быть расположены в любой секции с подходящими атрибутами доступа. Когда-то правила хорошего тона диктовали помещать каждую таблицу в свою персональную секцию, но теперь эта методика признана устаревшей. Теперь на смену ей пришла анархия и старый добрый квадратно гнездовой способ, когда содержимое служебных таблиц размазывается тонким слоем по всему страничному имиджу, что существенно утяжеляет алгоритм внедрения в исполняемый файл, но это уже тема другого разговора. Впрочем, как справедливо замечает Юрий Харон, дело тут совсем не в анархии, а в оптимизации по размеру/скорости загрузки.

Оверлей, в своем каноническом определении сводящийся к "хвостовой" части файла, не загружаемой в память, в PE-файлах может быть расположен в любом месте дискового образа – в

(статья в смысле) – что хочу то и делаю ;)

том числе и посередине. Действительно, если между двумя смежными секциями расположено несколько бесхозных секторов, не приватизированных ни какой секцией, такие сектора останутся без представления в памяти и имеют все основания считать себя оверлеями. Впрочем, кем они считают себя – неважно. Важно, кем их считают окружающие, ибо мнение, которое никто не разделяет, граничит с шизофренией, а сумасшедший оверлей – это что-то! Собственно говоря, оверлеями их можно называть только в переносном смысле. Спецификация на PE-файлы этого термина в упор не признает и никаких, даже самых примитивных, механизмов поддержки с оверлеями win32 API не обеспечивает (не считая, конечно, примитивного ввода/вывода).

За сим все! Теперь, после составления контурной карты PE-файлов, можно смело приступать к ее детализации, не рискуя заблудиться в непроходимых терминологических и технических джунглях. Мужайтесь! ELF файл еще более наворочен! И PE в сравнении с ним просто невинность какая-то!

Внимание*! Эту статью нельзя читать как приключенческий роман или детектив. Разумеется, я приложил все усилия и как мог структурировал материал, стремясь писать максимально доходчивым языком, хотя бы и ценой второстепенных деталей. Тем не менее, для оперативного переваривания информации вам придется обложится стопками распечаток и вооружившись hex-редактором, сопровождать чтение статьи перемещением курсора по файлу, чтобы самостоятельно "потрогать руками", все описываемые здесь структуры…*

Да осилит дорогу идущий! Когда вы доберетесь до конца, вы поймете почему не работают некоторые файлы, упакованные ASPack/ASProtect и как это исправить, не говоря уже о том, что сможете создавать абсолютно легальные файлы, которые ни один дизассемблер не дизассемблирует в принципе!

структура PE-файла

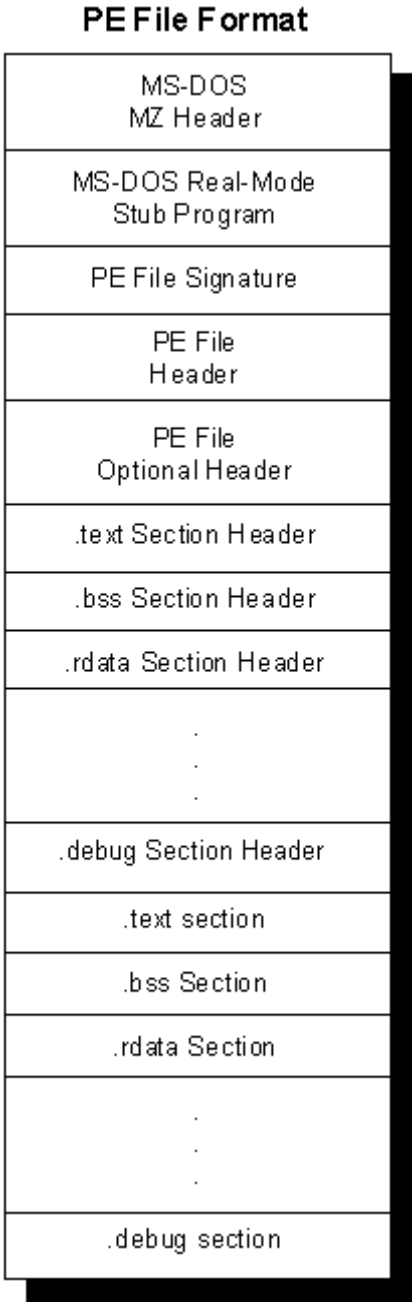

Рисунок 2 схематическое изображение PE-файла

Все PE-файлы без исключения (и системные драйвера в том числе!) начинаются с ***old-exe*** заголовка за концом которого следует *dos-заглушка* (ms-dos real-mode stub program или просто stub), обычно выводящая разочаровывающее ругательство на терминал, хотя в некоторых случаях в нее инкапсулирована MS-DOS версия программы, но это уже экзотика. Мэтт Питтерек в "Секретах системного программирования под Windows 95" пишет "*после того, как загрузчик win32 отобразит в память PE-файл, первый байт отображения файла соответствует первому байту заглушки DOS*". Это неверно! Первый байт отображения соответствует первому байту самого файла, т. е. отображение всегда начинается с сигнатуры "MZ", в чем легко можно убедиться загрузив файл в отладчик и просмотрев его дамп.

PE-заголовок, в подавляющем большинстве случаев начинающийся непосредственно за концом old-exe программы, на самом деле может быть расположен в любом месте файла – хоть в середине хоть в конце, т. к. загрузчик определяет его положение по двойному слову e_lfanew, смещенному на 3Ch байт от начала файла.

PE-заголовок представляет собой 18h-байтовую структуру данных, описывающую фундаментальные характеристики файла и содержащую "PE\x0\x0" сигнатуру, по которой файл собственного говоря и отождествляется.

Непосредственно за концом PE-заголовка, следует ***опциональный заголовок***, специфицирующий структуру страничного имиджа более детально (базовый адрес загрузки, размер образа, степень выравнивания – все это и многое другое задаются именно в нем). Название "опциональный" выбрано не очень удачно и слабо коррелирует с окружающей действительностью, ибо без опционального заголовка файл попросту не загрузиться, так какой же он к черту "опциальный", если обязательный? (впрочем, когда PE-формат только создавался все было по другому, а сейчас мы вынуждены тащить это наследие старины за собой). Важной структурой опционального заголовка является ***DATA_DIRECTORY***, представляющая собой массив указателей на подчиненные структуры данных, как-то: таблицы экспорта и импорта, отладочную информацию, таблицу перемещаемых элементов и т. д. Типичный размер опционального заголовка составляет E0h байт, но может варьироваться в ту или иную сторону, что определяется полнотой занятости DATA_DIRECTORY, а так же количеством мусора за ее концом (если таковой вдруг там есть, хотя его настоятельно рекомендуется избегать). Может показаться забавным, но размер опционального заголовка хранится в PE-заголовке, так что эти две структуры очень тесно связаны.

За концом опционального заголовка следует суверенная территория, оккупированная, ***таблицей секций***. Политическая принадлежность ее весьма условна. Ни к одному из заголовков она не принадлежит и, судя по всему, является самостоятельным заголовком безымянного типа (подробнее см. "SizeOfHeaders" и "таблица секций"). Редкое внедрение в исполняемый файл обходится без правки таблицы секций, поэтому эта структура для нас ключевая.

За концом таблицы секций раскинулась топкое болото ничейной области, не принадлежащей ни заголовкам, ни секциям и образовавшееся в результате выравнивания физических адресов секций по кратным адресам. В зависимости от ряда обстоятельств, подробно разбираемых по ходу изложения материала, заболоченная память может как отображаться на адресное пространство процесса, так и не отображаться на него. Обращаться с ней следует крайне осторожно, т. к. здесь может быть расположен чей-то оверлей, исполняемый код или структура данных (таблица диапазонного импорта, например).

Начиная с raw offset'a первой секции, указанного в таблице секций, простилается страничный имидж, точнее его упакованный дисковый образ. "Упакованный" в том смысле, что физические размеры секций (с учетом выравнивания) включают в себя лишь инициализированные данные и не содержат ничего лишнего (ну, хорошо, "не должны содержать ничего лишнего…"). Виртуальный размер секций может существенно превосходить физический, что с секциями данных случается сплошь и рядом. В памяти секции всегда упорядочены, чего нельзя сказать о дисковом образе. Помимо дыр, оставшихся от выравнивания, между секциями могут располагаться оверлеи, к тому же порядок следования секций в памяти и на диске совпадает далеко не всегда…

Одни секции имеют постоянное представительство в памяти, другие – нанимаются лишь на период загрузки, по завершении которой в любой момент могут быть безоговорочно выдворены оттуда (не сброшены в своп, а именно выдворены, то есть депортированы!). Что же до третьих – они вообще никогда не загружаются в память, ну разве что по частям. В частности, секция с отладочной информацией ведет себя именно так. Впрочем, отладочная информация не обязательно должна оформляться в виде отдельной секции и чаще она подцепляется к файлу в виде оверлея.

За концом последней секции обычно бывает расположено некоторое количество мусорных байт, оставляемых линкером по небрежности. Это не оверлей (к нему никогда не происходит обращений), хотя и нечто очень на него похожее. Разумеется, оверлеев может быть и несколько – системный загрузчик не налагает на это никаких ограничений, однако и не предоставляет никаких унифицированных механизмов работы с оверлеями – программа, создавшая свой оверлей, вынуждена работать с ним самостоятельно, задействовав API ввода/вывода (впрочем, "вывод" не работает в принципе, т. к. загруженный файл доступен только на чтение и запись в него наглухо заблокирована).

Короче говоря, физическое представление исполняемого файла представляет собой настоящее лоскутное одеяло, напоминающее политическую карту мира в стиле "раскрась сам". Переварить эту кухню очень непросто, поскольку закладываться ни на что нельзя и следует ожидать любых неожиданностей…

что можно и что нельзя делать с PE-файлом

Строго говоря, чужой исполняемый файл лучше не трогать, поскольку заранее не известно к чему именно он привязывается и какие структуры данных контролирует. С другой стороны, поведение подавляющего большинства файлов вполне предсказуемого и внедряться в них таки можно.

Дисковый файл и его виртуальный образ это, как говорят в Одессе: две большие разницы. С момента окончания загрузчики, стандартный PE-файл работает исключительно со своим виртуальным образом и не обращается непосредственно к самому файлу (исключение составляют оверлеи и секции отладочной информации, но это уже тема другого разговора). Нет, не так! Обращение к немодифицированным страницам файла все-таки происходит (при условии, что он загружен с винчестера, а не с дискеты или сетевого диска), Windows не настолько глупа, чтобы вытеснять в своп то, что в любой момент можно подкачать с диска. Впрочем, этот механизм настолько прозрачен, что учитывать его совершенно необязательно.

Внедряемый код может как угодно перекраивать дисковый файл, но виртуальный образ менять не должен. Точнее, после передачи управления на оригинальную точку входа виртуальный образ должен быть приведен в исходный вид. При этом допускается: а) увеличивать размер страничного имиджа, записываясь в его конец; б) оккупировать незанятые области (например, те, что используются для выравнивания); в) выделять память на стеке/куче, перемещая туда свое тело.

Поскольку, секции располагаются в файле по выровненным адресам, между ними практически всегда остается свободное пространство, уверенно вмещающее в себя крохотный загрузчик, подкачивающий "хвост" вируса из оверлея. Как вариант (если нет другого оверлея), можно увеличить размер последней секции и записаться в ее конец. Более радикально настроенный код может сбросить часть чужой секции в оверлей, усевшись на освободившееся место, а затем, непосредственно перед передачей управления, восстановить ее обратно. Внешний антураж выглядит просто замечательно, но задумайтесь, что произойдет, если: а) сбрасываемый фрагмент секции будет содержать одну или несколько служебных таблиц, например, таблицу импорта; б) сбрасываемый фрагмент секции будет содержать один или несколько перемещаемых элементов. Таким образом, перед тем как сбрасывать что бы то ни было в оверлей, внедряемый код должен проанализировать все служебные структуры, прописанные в DATA DIRECTORY, чтобы ненароком не сбросить ничего лишено. Затем, необходимо проанализировать таблицу перемещаемых элементов (если она есть) и либо выбрать участок свободный от перемещений, либо удалить соответствующие элементы из таблицы с тем, чтобы впоследствии обработать их самостоятельно. До ресурсов дотрагиваться ни в коем случае нельзя, иначе проводник иконки не найдет!

Но хватит говорить о плохом. Давайте лучше о хорошем. Все секции стандартного PE-файла, за исключением секции с отладочной информацией, используют только RVA/RRA и VA адресацию, а это значит, что мы можем свободно перемещать секции внутри дискового образа: менять их местами, внедрять между ними оверлеи и все это никак не скажется на работоспособности файла, поскольку страничный имидж во всех случаях будет один и тот же! Это не покажется удивительным, если вспомнить, что виртуальный и физический адреса каждой секции хранятся в различных никак не связанных друг с другом полях, поэтому внедрение кода в середину файла еще не обозначает его внедрения в середину страничного имиджа.

Теперь немного извращений для разнообразия. Внедряться в конец файла – слишком просто, неинтересно и небезопасно (антивирусы при этом матерятся так, что уши вянут). Внедряться в начало кодовой секции со сбросом оригинального содержимого последний в оверлей – слишком сложно. А что если… попробовать внедриться перед началом кодовой секции, передвинув ее начало в область младших адресов? Виртуальный образ окажется при этом практически нетронутым и останется лежать по тем же самым адресам, которые занимал до вторжения, что сохранит файлу работоспособность, попутно лишая разработчика внедряемого кода полового контакта с перемещаемыми элементами и прочими служебными структурами данных. Все это так, за исключением одного досадного "но". Первая секция подавляющего большинства файлов *уже* начинается по наименьшему из всех доступных адресов и передвигать ее просто некуда. Правда, под NT можно отключить выравнивание и делать с секциями все что угодно, но тогда файл не сможет работать под 9x (подробнее см. "FileAlignment/SectionAlignment"). Тоже самое относиться и к уменьшению базового адреса загрузки, компенсируемым увеличением стартовых адресов всех секций, в результате чего положение страничного имиджа не изменяется, а мы выигрываем место для внедрения своего

собственного кода. Увы! Служебные структуры PE-файлов активно используют RVA-адресацию, отсчитываемую от базового адреса загрузки, поэтому, просто взять и передвинуть базовый адрес не получится – необходимо как минимум проанализировать таблицы экспорта/импорта, таблицу ресурсов и скорректировать все RVA-адреса, а как максимум… типичный базовый адрес загрузки для исполняемых файлов – 400000h выбран далеко не случайно. Это минимальный базовый адрес загрузки в Windows 9x и если он будет меньше этого числа, системный загрузчик попытается переместить файл, потребовав таблицу перемещаемых элементов на бочку, а у исполняемых файлов она с некоторого времени по умолчанию отсутствует (ну разве что линкер при компоновке специально попросите). С динамическими библиотеками ситуация не так плачевна (их базовый адрес их загрузки выбирается с запасом, да и таблица перемещаемых элементов как правило есть), однако, сложность реализации внедряемого кода просто чудовищна, к тому же нестандартный адрес загрузки сразу бросается в глаза. Так что ценность этого приема очень сомнительна…

А теперь специально для настоящих извращенцев! Раздвигать страничный имидж все-таки **можно**! Секция кода практически никогда не обращается к секции данных по относительным адресам, а все абсолютные адреса в обязательном порядке должны быть перечислены в таблице перемещаемых элементов (конечно, при условии, что она вообще есть). Остаются лишь RVA/VA адреса служебных структур данных, однако, их реально скорректировать и вручную. Расширение страничного имиджа с внедрением в конец кодовой секции без сброса ее в оверлей – занятие не для слабонервных, однако, игра стоит свеч, поскольку такой код идеально вписывается в архитектуру существующего файла и не привлекает к себе никакого внимания. Грубо говоря, это единственный способ вторжения, который нельзя распознать визуально (подробнее см. статью в октябрьском номере "Системного администратора" за 2003 год).

описание основных полей PE-файла

Как уже говорилось, полностью описывать PE-файл мы не собираемся и предполагаем, что читатели: а) регулярно штудируют фирменную спецификацию перед сном; б) давным-давно распечатали файл WINNT.h из SDK и обклеили им стены своей хакерской берлоги на манер обоев. Все нижеприведенные структуры взяты именно оттуда (внимание – зачастую они именуются совсем не так, как в спецификации, что вносит в ряды разработчиков жуткую путаницу и сумятицу).

Здесь описываются не все, а лишь самые интересные и наименее известные поля, свойства и особенности поведения PE-файлов. За остальными – обращайтесь к документации.

[old-exe] e_magic

Содержит сигнатуру "MZ", доставшуюся в наследство от Марка Збиновски – ведущего разработчика MS-DOS и генерального архитектора EXE-формата. Если e_magic равен "MZ", загрузчик приступает к поиску "PE" сигнатуры, в противном случае его поведение становится неопределенным. NT и 9x поддерживают недокументированную сигнатуру "ZM", передающую управление на MS-DOS заглушку и обычно выводящую на экран "This program cannot be run in DOS mode", что в данном случае не соответствует действительности, поскольку программа запускается из Windows!

Один из приемов заражения PE-файлов сводиться к внедрению в MS-DOS заглушку, динамически восстанавливающую сигнатуру "MZ" и делающую себе exec для передачи управления программе-носителю. Для восстановления пораженных объектов просто замените "ZM" на "MZ" и при запуска файла из Windows (включая MS-DOS сессию) вирус больше никогда не получит управления.

Любители крутого изврата могут использовать сигнатуру "NE", передающую управление на заглушку и устанавливающую значения сегментных регистрах как в com, а не exe (DS == CS). Ни HIEW, ни IDA с таким файлом работать не могут и сразу же после его загрузке вылетают в астрал.

[old-exe] e_cparhdr

Размер old-exe заголовка в параграфах (1 параграф равен 200h байтам). В настоящее время никем не проверяется (ну разве, что дампером каким), однако, закладывается на это не стоит. Минимальный размер заголовка составляет 1 параграф, а максимальный – ограничен

размером самой MS-DOS заглушки, т. е. если он будет больше поля e_lfanew, файл может и не загрузиться.

[old-exe] e_lfanew

Смещение PE-заголовка в байтах от начала файла. Должно указывать на первый байт PE-сигнатуры "PE\x0\x0", выровненной по границе двойного слова, причем если сумма image base и e_lfanew вылетает за пределы отведенного загрузчиком адресного пространства, такой файл не грузится.

В памяти PE-заголовк (вместе со вмести остальными заголовками) всегда располагается перед первой секцией, вплотную прижимаясь к ее передней границе ("вплотную" значит, что расстояние между виртуальным адресом первой секции и концом заголовка должно быть меньше, чем Section Alignment). На диске, PE-заголовок может быть расположен в любом месте файла, например, его середине или конце (т. е. между началом файла и первым байтом PE-заголовка могут обосноваться она или несколько секций). Не знаю, сойдет ли какой загрузчик от этого с ума, но в Windows 9x/NT все работает. При этом, SizeOf Header должно быть равно действительному размеру PE-заголовка плюс e_lfanew; SectionAlignment >= SizeOfHeaders и FirstSection.RVA >= SizeOfHeaders.

[IMAGE_FILE_HEADER] Machine

Тип центрального процессора под который скомпилирован файл. Если здесь будет что-то отличное от 14Ch, на I386-машинах файл просто не загрузится.

[IMAGE_FILE_HEADER] NumberOfSections

Количество секций. Файл, не содержащий ни одной секции, завешивает Windows 9x и корректно прерывает свою загрузку под Windows NT. Максимальное количество секций определяется особенностями реализациями конкретного лоадера. Так, NT переваривает "всего" 60h секций. Другие загрузчики могут иметь и более жесткие ограничения. В общем, количество секций должно быть сведено к минимуму.

Если заявленное количество секций меньше числа записей в Section Table, то остальные секции просто не грузятся, но в целом такой файл обрабатывается вполне нормально. Настоящее веселье начинается, когда NumbersOfSection превышает количество реально существующих секций, вылетая за конец Section Table. Если здесь окажутся нули (как чаще всего и бывает), Windows 9x отреагирует вполне нормально, чего нельзя сказать об Windows NT, наотрез отказывающейся загружать такой файл. Файл, с количеством секций равным нулю, мертво завешивает Windows 9x, в то время как Windows NT обрабатывает такую ситуацию вполне нормально, выдавая неизменное "файл не является приложением win32".

Попутно заметим, что многие упаковщики исполняемых файлов по окончании процесса распаковки искажают это поле в памяти либо увеличивая, либо уменьшая его значение, в результате чего дамперы не могут корректно сбросить такой образ на диск. В pe-tools/lord-pe используется довольно ненадежный алгоритм, сканирующий Section Table и отталкивающийся от того, что если PointerToRelocations, PointerToLinenumbers, NumberOfRelocations и NumberOfLinenumbers равны нулю, а Characteristics – нет, значит, это секция. Эту святую простоту ничего не стоит обмануть! На самом деле, проверку следует ужесточить: если очередная запись в Section Table выглядит как секция (т.е. *все* поля валидны) – это секция и, соответственно, наоборот. Под валидностью здесь понимается, что адрес начала секции выровнен в памяти и лежит непосредственно за концом предыдущей секции, а размер секции не вылетает за пределы страничного имиджа.

Ниже приведен простой макрос, считывающий содержимое поля NumberOfSection по указателю на первый байт PE-заголовка.

```
#define xNumOfSec(p) (*((WORD*) (p+0x6)))    // p - указатель на PE-заголовок
```

Листинг 1 считыватель содержимого NumberOfSection

[image_file_header] PointerToSymbolTable/NumberOfSymbols

Указатель на/размер отладочной информации в объективных файлов. В настоящее время не используется (да и раньше оно не использовалось тоже). Линкеры топчут оба поля в ноль, отладчики, дизассемблеры и системный загрузчик игнорирует его. Для предотвращения

сброса дампа программы на диск запишите сюда нечто отличное от нуля и подтяните (в памяти) поле NumberOfSection от реального значения до безобразия. Текущие версии pe-tools'a сдохнут от зависти, но если NEOx сподобится встроить нормальный валидатор, этот трюк перестанет работать.

[image_file_header] SizeOfOptionalHeader

Размер опционального заголовка, идущего следом за IMAGE_FILE_HEADER'ом. Должен указывать на первый байт Section Table (т. е. e_lfanew + 18h + SizeOfOptionalHeader = &Section Table), где 18h – sizeof(IMAGE_FILE_HEADER). Если это не так, файл не загружается. И хотя некоторые загрузчики вычисляют указатель на Section Table отталкиваясь от NumberOfRvaAndSizes, закладываться на это не стоит, т. к. системные загрузчики этого мнения не разделяют.

```
#define xopt_sz(p)             (*((WORD*)(p + 0x14 /* size of optional header */)))
#define pSectionTable(p)       ((BYTE*)(xopt_sz(p)+0x18 /* size of image heafer */+p))
#define pSectionTable_alt(p)   ((BYTE*)((*((DWORD*)(p+0x74)))*8 + 0x78 + p))
```

Листинг 2 макросы, возвращающие размер опционального заголовка, указатель на таблицу секций, вычисленный стандартным и альтернативным методами. в качестве входного аргумента все трое принимают указатель на первый байт PE-заголовка

[image_file_header] Characteristics

Атрибуты файла. Если (Characteristics & IMAGE_FILE_EXECUTABLE_IMAGE) == 0 файл не грузиться, т. е. первый, считая от нуля, бит характеристик обязательно должен быть установлен. У динамических библиотек должно быть установлено как минимум два атрибута: IMAGE_FILE_EXECUTABLE_IMAGE/0002h и IMAGE_FILE_DLL/2000h, тоже самое относится и к исполняемым файлам, экспортирующим одну или более функций. Если атрибут IMAGE_FILE_DLL установлен, но экспорта нет, исполняемый файл запускаться не будет.

Остальные атрибуты не столь фатальны и под Windows NT/9x безболезненно переносят любые значения, хотя по идее этого делать этого не должны. Взять хотя IMAGE_FILE_BYTES_REVERSED_LO и IMAGE_FILE_BYTES_REVERSED_HI, описывающие порядок следования байт в слове. Можно глупый вопрос? Какому абстрактному состоянию процессора соответствует одновременная установка обоих атрибутов? И какие действия должен предпринять загрузчик, если установленный порядок следования байт будет отличаться от поддерживаемого процессором? Операционные системы от Microsoft, писанные через известное место, просто игнорируют эти атрибуты за ненадобностью. Тоже самое относится и к атрибуту IMAGE_FILE_32BIT_MACHINE/0100h, которым по умолчанию награждаются все 32-разрядные файлы (16-разрядный PE – это сильно). Впрочем, без крайней нужды лучше не извращаться и заполнять все поля правильно.

Весьма интересен флаг IMAGE_FILE_DEBUG_STRIPPED/0200h, указывающий на отсутствие отладочной информации и запрещающий отладчикам работать с ней даже тогда, когда она есть. Отладочная информация привязана к абсолютным смещениям, отсчитываем от начала файла и при внедрении в файл чужеродного кода путем его расширения, отладочная информация перестает соответствовать действительности и поведение отладчиков становится крайне неадекватным. Для решения проблемы существуют три пути: а) скорректировать отладочную информацию (но для этого нужно знать ее формат); б) отрезать отладочную информацию от файла (но для этого ее надо найти, кроме того, за концом файла может быть расположен посторонний оверлей); в) установить флаг IMAGE_FILE_DEBUG_STRIPPED. Последний способ самый простой, но и самый надежный. Соответственно, для восстановления пораженных объектов необходимо выкусить чужеродный код из тела файла и сбросить флаг IMAGE_FILE_DEBUG_STRIPPED, в противном случае отладчик не покажет исходный код отлаживаемого файла.

Иначе ведет себе флаг IMAGE_FILE_RELOCS_STRIPPED, запрещающий перемещать файл когда релокаций нет. Когда же они есть, загрузчик может с полным основанием на него покласть (и ведь кладет!). Зачем же тогда этот атрибут нужен? Ведь переместить файл без таблицы перемещаемых элементов все равно невозможно… А вот это еще как сказать! Служебные структуры PE-файла используют только относительную адресацию и потому любой PE-файл от рождения уже перемещаем. Вся загвоздка в программном коде, активно использующем абсолютную адресацию (ну так уж устроены современные компиляторы).

Технически ничего не стоит создать PE-файл не содержащий перемещаемых элементов и способный работать по любому любому адресу (давным-давно, когда землей владели динозавры и никаких операционных систем еще не существовало, этим мог похвастаться практически каждый). Таким образом, возникает неоднозначность: толи перемещаемых элементов нет, потому что файл полностью перемещаем и fixup'ы ему не нужны, толи они просто недоступны и перемещать такой файл ни в коем случае нельзя.

По умолчанию ms link версии и 6.0 и старше внедряет перемещаемые элементы только в DLL, а исполняемые файлы сходят с конвейера неперемещаемыми, однако, закладываться на это нельзя и при внедрении собственного кода в чужеродный PE-файл необходимо удостовериться, что он не содержит перемещаемых элементов, в противном случае возникают следующие программы: а) ваш код не может закладываться на image base и должен быть готов к загрузке по любому адресу; б) модификация ячеек, относящихся к перемещаемым элементам, обычно заканчивается крахом программы, поскольку они автоматически "исправляются" системным загрузчиком. Допустим, в программе был код типа: mov eax, 0400000h (B8 **00 00 40 00**), поверх которого мы начертали: push ebp/mov ebp, esp (55/8B EC). Допустим также, что в силу некоторых причин базовый адрес загрузки изменился с 40.00.00h на 1.00.00.00h. Ячейка памяти, ранее хранящая непосредственный операнд инструкции mov, будет переделана в 1.00.00.00h, что превратит команду mov ebp, esp в add [eax], al со всеми вытекающими отсюда последствиями.

Существует по меньшей мере три пути решения этой проблемы: а) убить fixup'ы (но тогда файл станет неперемещаемым, а ведь некоторые исполняемые файлы подспудно экспортируют одну или несколько функций и без fixup'ов не смогут работать); б) перезаписывать только неперемещаемые ячейки (но это приведет к размазыванию кода по всему файлу, существенно усложняя его алгоритм); в) обрабатывать перемещаемые элементы самостоятельно – чтобы система могла перемещать файл при необходимости, но не корежила наш код, подсуньте ей пустую таблицу перемещаемых элементов (подробнее см. "перемещаемые элементы").

[image_optional_header] Magic

Состояние отображаемого файла. Если здесь будет что-то отличное от 10Bh (сигнатура исполняемого отображения), файл не загрузиться. PE64-файлам соответствует сигнатура 20Bh (все адреса у них 64-разрядные), а в остальном они ведут себя как и нормальные, 32-разряные PE-файлы.

[image_optional_header] SizeOfCode/SizeOfInitializedData/SizeOfUninitializedData

Суммарный размер секций кода, инициализированных и неинициализированных данных (т. е. секций, имеющих атрибуты `IMAGE_SCN_CNT_CODE/20h`, `IMAGE_SCN_CNT_INITIALIZED_DATA/40h` и `IMAGE_SCN_CNT_UNINITIALIZED_DATA/80h`). Никем не проверяется и может принимать любые, в том числе и заведомо бессмысленные значения.

Всякий линкер заполняет эти поля по-своему: одни берут физический размер секций на диске, другие – виртуальный размер в памяти, выровненный по границе Section Alignment, причем алгоритм определения принадлежности секции к тому или иному типу не стандартизирован и в полку разработчиков наблюдается большой разброд и шатание. Наиболее демократичное сословие определяет "родословную" по принципу OR (т. е. секция с атрибутами 60h считается и секций кода, и секцией данных). Иначе действует аристократическая прослойка, придерживающаяся принципа XOR и относящая к данным только секции с атрибутами 40h (80h?). Для секции кода сделано некоторое послабление (ведь всякий код на каком-то этапе обработки представляется данными) и секция с атрибутами 60h или A0h все-таки относится к коду (в противном случае, образовались бы не классифицируемые секции, размер которых не был подсчитан, а этого допускать нельзя – религия не велит).

Как бы там ни было, системному загрузчику на это глубоко наплевать (давным-давно, когда секции кода, данных и неинициализированных данных помещались в "свои" сегменты эти поля еще имели какой-то смысл, но сейчас это рудиментый пережиток старины).

[image_optional_header] BaseOfCode/BaseOfData

Относительные базовые адреса кодовой секции и секции данных. Никем не проверяется и всяким компоновщиком заполняется по своему. Для восстановления душевного равновесия оба поля можно смело сбросить в ноль, отдавая дань древним буддийским традициям.

[image_optional_header] AddressOfEntryPoint

Относительный адрес точки входа, отсчитываемый от начала Image Base. Может указывать в любую точку адресного пространства, в том числе и не принадлежащую страничному имиджу (например, направленную на какую-нибудь функции внутри ядра или dll). Для передачи управления на адреса, лежащие ниже Image Base можно использовать целочисленное переполнение. Правда, не факт, что все загрузчики поймут нас правильно, (NT поймет точно, остальные не проверял), так что закладываться на это нельзя.

Если точка входа направлена на заголовок или последнюю секцию файла, антивирусы начинают жутко материться, обвиняя файл в зараженности вирусом, поэтому во избежание недоразумений точку входа лучше всего располагать в первой секции файла, которой по обыкновению является кодовая секция .text

Для exe-файлов точка входа соответствует адресу с которого начинается выполнение и не может быть равна нулю, а для динамических библиотек – функции диспетчера, условно называемой нами DllMain, хотя на самом деле при компоновке dll с настройками по умолчанию, компоновщик внедряет стартовый код, перехватывающий на себя управление и вызывающий "настоящую" DllMain по своему желанию. DllMain вызывается при следующих обстоятельствах – загрузка/выгрузка dll и создание/уничтожение потока, если точка входа в dll равна нулю, функция DllMain не вызывается.

Обязательно учитывайте это при внедрении собственного кода в dll! Чтобы отличить dll от обычных файлов следует проанализировать поле характеристик (см. "Characteristics"). Опираться на наличие/отсутствие таблицы экспорта ни в коем случае нельзя, поскольку экспортировать функции могут не только динамические библиотеки, но исполняемые файлы! К тому же, иногда встречаются динамические библиотеки не экспортирующие ни одной функции.

[image_optional_header] ImageBase

Базовый адрес загрузки страничного имиджа, измеряемый в абсолютных адресах, отсчитываемых от начала сегмента или, в терминологии оригинальной спецификации, preferred address (предпочтительный адрес загрузки). При наличии таблицы перемещаемых элементов, файл может быть загружен по адресу отличному от указанного в заголовке. Это происходит в тех случаях, когда требуемый адрес занят системой, динамической библиотекой или загрузчику захотелось что-то подвигать.

Если предпочтительный адрес совпадает с адресом уже загруженной системной библиотеки, поведение последней становится неадекватной. Отладчик, интегрированный в Microsoft Visual Studio, запущенный под управлением NT, проскакивает точку входа и умирает где-то в окрестностях ядра (отлаживая программа при этом продолжает исполняться). Под Windows 98 такие файлы отлаживаются вполне нормально, но при выходе из Windows уводят ее в астрал.

Менять чужой Image Base ни в коем случае нельзя, т. к. перемещаемым элементов в этом случае будет просто не отчего отталкиваться. И хотя системный загрузчик в большинстве случаев загрузит такой файл вполне нормально, работать он не сможет, ну во всяком случае до тех пор, пока все перемещаемые элементы не будет скорректированы надлежащим образом.

[image_optional_header] FileAlignment/SectionAlignment

Кратность выравнивания секций на диске и в памяти. Очень интересное поле! Официально о кратности выравнивая известно лишь то, что она представляет собой степень двойки, причем: а) Section Alignment должно быть больше или равно 1000h байт; б) File Alignment должно быть больше или равно 200h байт; в) Section Alignment должно быть больше или равно File Alignment. Если хотя бы одно из этих условий не соблюдается, файл не будет загружен.

В Windows NT существует недокументированная возможность отключения выравнивания, основанная на том, что загрузку прикладных исполняемых файлов/динамических библиотек и системных драйверов обрабатывает один и тот же загрузчик.

Если Section Alignment == File Alignment, то последнее может принимать любой значение, представляющее собой степень двойки (например, 20h). Условимся называть такие файлы "не выровненными". Хотя этот термин не вполне корректен, лучшего пока не придумали.

К не выровненным файлам предъявляется следующее, достаточно жесткое требование – виртуальные и физические адреса всех секций обязаны совпадать, т. е. страничный имидж должен полностью соответствовать своему дисковому образу. Впрочем, никакое правило не обходится без исключений и виртуальный размер секций может быть меньше их физического размера, но не более чем Section Alignment – 1 байт (т. е. секция все равно будет выровнена в памяти). Самое интересное, что это данное правило рекурсивно и даже среди исключений встречаются исключения – если физический размер последней секции вылетает за пределы загружаемого файла, операционная система выбрасывает голубой экран смерти и… погибает (во всяком случае, w2k sp3 ведет себя именно так, остальные не проверял). Полномочия администратора для этого не требуются и даже самая ничтожная личность может устроить грандиозный DoS. Демонстрационные файлы прилагаются.

Операционные системы семейства Windows 9x неспособны обрабатывать не выровненные файлы и с возмущением отказывают им в загрузке, выплевывая целых два диалоговых окна. Впрочем, ареал обитания Windows 9x неуклонно сокращается и будущее принадлежит NT.

Для создания не выровненных файлов можно воспользоваться линкером от Microsoft, задав ему ключ /ALIGN:32 совместно с ключом /DRIVER. Без ключа /DRIVER ключ /ALIGN будет проигнорирован и линкер использует кратность выравнивания по умолчанию.

```
#define Is2power(x)                 (!(x & (x-1)))
#define ALIGN_DOWN(x, align)        (x & ~(align-1))
#define ALIGN_UP(x, align)          ((x & (align-1))?ALIGN_DOWN(x,align)+align:x)
```

Листинг 3 макросы для выравнивания с округлением "вниз" и "вверх"

[image_optional_header] SizeOfImage

Размер страничного имиджа, выровненный на величину Section Alignment. Размер страничного имиджа всегда равен виртуальному адресу последней секции плюс ее размер (выровненный, виртуальный). Если размер страничного образа вычислен неправильно, файл не загружается.

```
#define xImageSize(p) (*(DWORD*)(pLastSection(p) + 0xC /* va */) +\
        ALIGN_UP(*(DWORD*)(pLastSection(p) + 0x8 /* v_sz */), xObjectAlign(p)))
```

Листинг 4 макрос для вычисления реального размера страничного имиджа

[image_optional_header] SizeOfHeaders

Суммарный размер всех заголовков, сообщающий загрузчику сколько байт читать от начала файла. С этим полем связано два ограничения: во-первых, SizeOfHeaders должен быть выбран так, чтобы загрузчик считал все, что ему необходимо прочитать, а во вторых, он не может превышать RVA первой секции (поскольку, в противном случае, какая-то часть секции оказалась бы спроецированной на область памяти, принадлежащей заголовку, а это недопустимо, ибо ни на какую страницу файла не могут отображаться более одного сектора одновременно).

Обычно SizeOfHeaders устанавливается на конец Section Table, однако, это не самое лучшее решение. Судите сами. Совокупный размер всех заголовков при стандартной MS-DOS заглушке составляет порядка ~300h байт или даже менее того, в то время как физический адрес первой секции – от 400h байт и выше. Отодвинуть секцию назад нельзя – выравнивание не позволяет (см. "FileAlignment/SectionAlignment"). Правда, если вынуть MS-DOS заглушку, можно ужать SizeOfHeaders до 200h байт, в аккурат перед началом первой секции, но это уже изврат. Короче говоря, если следовать рекомендациям от Microsoft, ~100h байт мы неизбежно теряем, что не есть хорошо. Вот некоторые линкеры и размешают здесь таблицу имен, содержащую перечень загружаемых DLL или что-то типа того. Поэтому, чтобы ненароком не нарваться на коварный конфликт, лучше всего подтянуть SizeOfHeaders к min(pFirstSection->RawOffset, pFirstSection->va).

Некоторые нехорошие программы (вирусы, упаковщики, дамперы) устанавливают SizeOfHeader на raw offset первой секции, что неправильно. Между концом всех заголовков и

физическим началом первой секции может быть расположено любое, кратное File Alignment, количество байт, например, 1 гигабайт, и это при том что виртуальный адрес первой секции – 1000h. Как такое может быть? А очень просто – SizeOfHeaders <= 1000h и остаток нашего гигабайта не читается и не проецируется в память, поэтому никаких конфликтов и не возникает. Что может быть в этом гигабайте? Ну, например, хитрый оверлей, внедренный тем же вирусом (и такие вирусы уже есть).

[image_optional_header] CheckSum

Контрольная сумма файла. Проверяется только NT, да то и лишь при загрузке некоторых системных библиотек и, разумеется, самого ядра. Алгоритм расчета можно найти в IMAGEHEL.DLL функция CheckSumMappedFile. По слухам ее исходные тексты входят в SDK. У меня есть SDK, но ничего подобного я там не видел (может, плохо искал?). Впрочем, алгоритм расчета тривиален и декомпилируется на ура.

[image_optional_header] Subsystem

Требуемая подсистема, которую операционная система должна предоставить файлу. Может принимать следующие значения:

00h IMAGE_SUBSYSTEM_UNKNOWN:
неизвестная подсистема, файл не загружается.

01h IMAGE_SUBSYSTEM_NATIVE:
подсистема не требуется, файл исполняется в "родном" окружении ядра и скорее всего представляет собой драйвер устройства. обычным путем не загружается. если вы пишите вирус/упаковщик/протектор ни в коем случае не обрабатывайте таких файлов, если только точно не уверены, в том, что вы делаете. ***внимание****: при загрузке драйверов Windows игнорирует поле подсистемы и оно может быть любым, поэтому, если Subsystem != IMAGE_SUBSYSTEM_NATIVE это еще не значит, что данный файл не является драйвером.*

02h IMAGE_SUBSYSTEM_WINDOWS_GUI:
графическая win32 подсистема. операционная система загружает файл нормальным образом, ну а дальше флаг ему в руки и пусть все что ему нужно добываем сам.

03h IMAGE_SUBSYSTEM_WINDOWS_CUI:
терминальная (она же консольная) win32 подсистема. тоже самое, что и IMAGE_SUBSYSTEM_WINDOWS_GUI, но в этом случае файлу на халяву достается автоматически создаваемая консоль с готовыми дескрипторами ввода/вывода. вообще говоря, разница между консольными и графическими приложениями очень условна – консольные приложения могут вызывать GUI32/USER32-функции, а графические приложения – открывать одну или несколько консолей (например, в отладочных целях). кстати говоря, с этим связана одна забавная проблема, с которой сталкиваются многие "программисты", пытающие подавить создание ненужного им окна (ну мало ли, может они шпиона какого пишут, а это окно его демаскирует). предотвратить автоматическое создание окна очень просто – достаточно… не создавать его!

05h IMAGE_SUBSYSTEM_OS2_CUI:
подсистема OS/2. только для приложений OS/2 (одним из которых, кстати говоря, является всем известный HIEW) и только для Windows NT. Windows 9x не может обрабатывать такие файлы

07h IMAGE_SUBSYSTEM_POSIX_CUI:
подсистема POSIX. только для приложений UNIX и только для Windows NT.

09h IMAGE_SUBSYSTEM_WINDOWS_CE_GUI:
файл предназначен для исполнения в среде Windows CE. Ни Windows NT, ни Windows 9x не могут обрабатывать такие файлы.

0Ah MAGE_SUBSYSTEM_EFI_APPLICATION:

0Bh IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER:
0Ch IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER:

подсистема EFI (Extensible Firmware Initiative).

[image_optional_header] DllCharacteristics

очень странное поле. Мэтт Питрек пишет, что оно определяет набор флагов. указывающих при каких условиях точка входа в DLL получает управление (как-то, загрузка dll в адресное пространство процесса, создание/завершение нового потока и выгрузка dll из памяти). В спецификации на PE-формат эти поля помечены как зарезервированные и Windows игнорирует их значение, поэтому у большинства файлов оно равно нулю.

Согласно спецификации 6.0 от 1999 года (самой свежей спецификации на сегодняшний день), загрузчик должен поддерживать и другие флаги: 800h – не биндить образ, 2000h – загружать драйвер как WDM драйвер; 8000h – файл поддерживает работу под терминальным сервером. Экспериментальная проверка показала, что W2K игнорирует эти флаги.

[image_optional_header] SizeOfStackReserve/SizeOfStackCommit, SizeOfHeapReserve/SizeOfHeapCommit

объем зарезервированной/выделенной памяти под стек/кучу в байтах. Если SizeOfCommit > SizeOfReverse файл не загружается. Ноль обозначает значение по умолчанию.

[image_optional_header] NumberOfRvaAndSizes

Количество элементов (не байт) в DATA_DIRECTORY следующей непосредственно за этим полем. Из-за грубых ошибок в системном загрузчике компоновщики от Borland и Microsoft **всегда** выставляют полный размер директории, равный 10h, даже если реально его не используют. Например, Windows 9x не проверяет, что NumberOfRvaAndSizes >= RELOCATION и/или RESOURCE и если подсунуть ему запрос к одной из этих секций, а таких директорий нет – кранты. Windows NT не проверяет (при загрузке dll) "достаточности" TLS_DIRECTORY и если этот TLS-механизм активирован, а TLS-директории нет – опять кранты.

Компоновщик Юрия Харона выгодно отличается тем, что усекает размер директории до минимума, но и кода вокруг процедуры "сокращений" там строк пятьсот, а уж сколько времени было убито в ИДЕ…

Есть и другая проблема. По спецификации DATA_DIRECTORY располагается в самом конце опционального заголовка и непосредственно за его концом начинается таблица секций. Таким образом, указатель на таблицу секций может быть получен либо так: `((BYTE*) ((*((WORD*)(p + 0x14 /* size of optional header */)))+ 0x18 /* size of image header */ + p))`, либо так: `((BYTE*) ( (*((DWORD*)(p+0x74 /* NumRVAandSize */)))*8 + 0x78 /* begin DATA_DIRECTOTY */+ p))`. Системный загрузчик использует первый способ и допускает, что за между DATA_DIRECTORY и SECTION_TABLE может быть расположено некоторое количество "бесхозных" байт. Некоторые дизассемблеры и упаковщики считают иначе и ищут SECTION_TABLE непосредственно за концом DATA_DIRECTORY. Вот и давайте подсунем им подложную SECTION_TABLE! Пускай их авторы почаще заглядывают в WINNT.H, который недвусмысленно говорит, что:

```
#define IMAGE_FIRST_SECTION( ntheader ) ((PIMAGE_SECTION_HEADER)        \
            ((ULONG_PTR)ntheader +                                      \
             FIELD_OFFSET( IMAGE_NT_HEADERS, OptionalHeader ) +         \
             ((PIMAGE_NT_HEADERS)(ntheader))->FileHeader.SizeOfOptionalHeader   \
            ))
```

…так что лезть дизассемблером в системный загрузчик совсем необязательно!

DATA DIRECTORY

00h IMAGE_DIRECTORY_ENTRY_EXPORT:

Указатель на таблицу экспортируемых функций и данных (далее по тексту просто функций). Встречается преимущественно в динамических библиотеках и драйверах, однако, заниматься экспортом товаров может и рядовой исполняемый файл. Использует RVA и VA адресацию. (подробнее см. "экспорт").

01h IMAGE_DIRECTORY_ENTRY_IMPORT:

Указатель на таблицу импортируемых функций, используемую для связи файла с внешним миром, и активируемую системным загрузчиком когда все остальные механизмы импорта недоступны. Использует RVA и VA адреса. (подробнее см. "импорт").

02h IMAGE_DIRECTORY_ENTRY_RESOURCE:

Указатель на таблицу ресурсов, хранящую строки, пиктограммы, курсоры, диалоги и прочие кирпичики пользовательского интерфейса (хотя какие это кирпичики? настоящие бетонные блоки!). Таблица ресурсов организована в виде трехуровневого двоичного дерева, слишком запутанного и разлапистого, чтобы его было можно привести здесь, но к счастью использующего только RVA адресацию, т. е. не чувствительного к смещению "свой" секции (а это как правило секция .rsrc) внутри файла. Однако, если вы вздумаете править RVA (например, для внедрения новой секции в середину страничного имиджа или переносу image base), вам придется основательно потрудиться с этой структурой, подробное описание которой кстати говоря можно найти в уже упомянутой статье "The Portable Executable File Format from Top to Bottom".

03h IMAGE_DIRECTORY_ENTRY_EXCEPTION:

указывает на exception directory (директорию исключений), обычно размещаемую в секции .pdata (хотя это и необязательно). используется только на следующих архитектурах: MIPS, Alpha32/64, ARM, PowerPC, SH3, SH, WindowsCE. К микропроцессорам семейства Intel это не относится и IX386-загрузчик игнорирует это поле, поэтому оно может принимать любое значение.

04h IMAGE_DIRECTORY_ENTRY_SECURITY:

указывает на Certificate Table (таблицу сертификатов), располагающуюся строго в .debug-секции и адресуемой не по RVA-адресам, а по физическим смещениям внутри файла (так происходит потому, что таблица сертификатов не грузиться в память и обитает исключительно на диске).

Если IMAGE_DIRECTORY_ENTRY_SECURITY != 0, ни в коем случае не пытайтесь внедрять в файл посторонний код, иначе он откажет в работе.

05h IMAGE_DIRECTORY_ENTRY_BASERELOC:

он же fixup используют RVA адреса. (см. "перемещаемые элементы").

06h IMAGE_DIRECTORY_ENTRY_DEBUG:

отладочная информация, используемая дизассемблерами и дебеггерами. использует RVA и RAW OFFSET адресацию. системный загрузчик ее игнорирует.

07h IMAGE_DIRECTORY_ENTRY_ARCHITECTURE:

оно же "description". на I386-платформе судя по всему предназначен для хранения информации о копирайтах (на это в частности, указывает определение IMAGE_DIRECTORY_ENTRY_COPYRIGHT, данное в WINNT.H), за формирование которых отвечает ключ –D, переданный Багдадскому линкеру ilinlk32.exe, при этом в IMAGE_DIRECTORY_ENTRY_ARCHITECTURE помещается RVA-указатель на строку комментариев, по умолчанию располагающуюся в секции .text. Компоновщик ms link при некоторых до конца не выясненных обстоятельствах помещает в это поле информацию об архитектуре, однако, системный загрузчик ее никогда не использует.

08h IMAGE_DIRECTORY_ENTRY_GLOBALPTR:

Указатель на таблицу регистров глобальных указателей. используется только на процессорах ALPHA и PowerPC. На I386-платформе это поле лишено смысла и загрузчик его игнорирует.

09h IMAGE_DIRECTORY_ENTRY_TLS:

Хранилище статической локальной памяти потока (Thread Local Storage). TLS-механизм обеспечивает "прозрачную" работу с глобальными переменными в многопоточных средах без риска, что переменная в самый неподходящий момент будет

модифицирована другим потоком. Сюда попадают переменные, объявленные как __declspec(thread). По причине большой причудливости и крайней тяжеловесности реализации (один шаг в сторону и операционная система стреляет без предупреждения) используется крайне редко. К тому же Windows NT и Windows 9x обрабатывают это поле сильно неодинаково. Хранилище обычно размещается в секции .tls, хотя это и необязательно. Использует RVA и VA адреса.

10h IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG:

Содержит информацию о конфигурации глобальных флагов, необходимых для нормальной работы программы, имеет смысл только в Windows NT и производных от нее системах. Это поле практически никем не используется, но если возникнет желание узнать о нем больше – см. прототип структуры IMAGE_LOAD_CONFIG_DIRECTORY32 в WINNT.h, а так же ее описание в Platform SDK. За описанием самих флагов обращайтесь к утилите gflags.exe, вохоящей в состав Resource Kit и NTDDK. Информация о конфигурации использует VA адресацию (точнее, пока еще не использует, но резервирует эту возможность на будущее).

11h IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT:

указатель на таблицу диапазонного импорта, имеющей приоритет над IMAGE_DIRECTORY_ENTRY_IMPORT и обрабатываемой загрузчиком в первую очередь (зачатую, до IMAGE_DIRECTORY_ENTRY_IMPORT дело вообще не доходит). По устоявшейся традиции таблица диапазонного импорта размещается в PE-заголовке, хотя это и не обязательно и некоторые линкеры ведут себя иначе. Используется RVA и RRAW OFFSET адресация. (подробнее см. "импорт").

12h IMAGE_DIRECTORY_ENTRY_IAT:

указатель на IAT (подчиненная структура таблицы импорта). используется загрузчиком Windows XP, остальные операционные системы это поле во видимому игнорируют. подробнее см. "импорт")

13h IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT:

указатель на таблицу отложенного импорта, использующей RVA/VA-адресацию, но фактически остающейся не стандартизованной и отданной на откуп воле конкретных реализаторов (подробнее см. "импорт").

14h IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR:

если не равно нулю, то файл представляет собой .NET-приложение, состоящее из байт-кода, поэтому попытка внедрения в него x86 когда ничего хорошего не принесет.

таблица секций

Четкого определения термина "**секция**" не существует. Упрощенно говоря, секция – это непрерывная область памяти внутри страничного имиджа со своими атрибутами, независящими от атрибутов остальных секций. Представление секции в памяти не обязательно должно совпадать с ее дисковым образом, который в принципе может вообще отсутствовать (секциям неинициализированных данных нечего делать на диске и потому они представлены исключительно в памяти).

Каждая секция управляется "своей" записью в одноименной структуре данных, носящей имя "таблицы секций". Таблица секций начинается сразу же за концом опционального заголовка, размер которого содержится в поле SizeOfOptionalHeader, и представляет собой массив структур IMAGE_SECTION_HEADER, количество задается полем NumberOfSection.

Порядок секций может быть любым, но системный загрузчик оптимизирован под следующую последовательность: сначала идет кодовая секция, за ней следует одна или несколько секции инициализированных данных и замыкает строк секция неинициализированных данных.

Структура IMAGE_SECTION_HEADER состоит из следующих полей:

```
typedef struct _IMAGE_SECTION_HEADER {
        BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];
        union {
                DWORD   PhysicalAddress;
```

```
            DWORD   VirtualSize;
            } Misc;
        DWORD   VirtualAddress;
        DWORD   SizeOfRawData;
        DWORD   PointerToRawData;
        DWORD   PointerToRelocations;
        DWORD   PointerToLinenumbers;
        WORD    NumberOfRelocations;
        WORD    NumberOfLinenumbers;
        DWORD   Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

Листинг 5 прототип структуры IMAGE_SECTION_HEADER

Поле **Name** представляет собой восьми байтовый массив с ASCII-именем секции внутри (именно именем, а не указателем на имя!). Если длина имени меньше восьми байт, остающийся хвост дополняется нулями, если же имя занимает весь массив целиком, завершающий нуль в его конце не ставится (некоторые дизассемблеры не учитывают этого обстоятельства и захватывают примыкающий к массиву мусор).

Само по себе имя секции не несет никакого метафизического смысла и было введено в эксплуатацию исключительно из эстетических соображений. Системный загрузчик его игнорирует, хотя некоторые вирусы/протекторы/упаковщики распознают "свои" секции только так и всякое искажение имени валит их наповал. Ходят слухи по поводу того, что библиотека oleaut32.dll, входящая в состав Windows, опознает секцию ресурсов по ее имени, а не по записи в DATA_DIRECTORY. В исходных текстах популярного упаковщика UPX присутствует следующий комментарий: "*...after some windoze debugging I found that the name of the sections DOES matter :(.rsrc is used by oleaut32.dll (TYPELIBS) and because of this lame dll, the resource stuff must be the first in the 3rd section - the author of this dll seems to be too idiot to use the data directories... M$ suxx 4 ever! ...even worse: exploder.exe in NiceTry also depends on this to locate version info*". Дизассемблирование подтверждает, что библиотека oleaut32.dll действительно содержит внутри себя текстовую строку ".rsrc" и активно ее использует. Да мало ли на свете идиотов, привязывающихся к именем секций? Поэтому без особой нужны имена секций чужого файла лучше не изменять.

Поля **VirtualAddress** и **PointerToRawData** содержат RVA-адрес начала секции в памяти и ее смещение относительно начала файла соответственно. Виртуальный и физический адреса должены быть выровнены на величину Section Alignment/File Alignment, прописанную в опциональном заголовке, причем виртуальный адрес первой секции должен быть равен ALIGN_UP(SizeOfHeaders, SectionAlignment), в противном случае файл не загрузиться. Физический адрес секции может быть любым, достаточно только, чтобы он был выровнен на величину File Alignment.

Поля **VirtualSize** и **SizeOfRawData** содержат виртуальную и физическую длину секции соответственно. Вот тут-то и начинается самое интересное! Если виртуальный размер больше физического, то при загрузке секции в память ее хвост заполняется нулями, при этом наличие атрибута инициализированных/неинициализированных данных совершенно необязательно. Если физический размер больше виртуального, то… единственное, что можно сказать с уверенностью, такой файл будет нормально загружен в память. Как? А вот это уже зависит от реализации! Начнем с того, что нулевой виртуальный размер предписывает загрузчику отталкиваться от физического размера секции, предварительно округлив его на величину Section Alignment и заполнив хвост нулями. Все промежуточные состояния неопределенны – загрузчик может считать: а) ровно Virtual Size байт; б) ALIGN_UP(Virtual Size, File Alignment) байт; в) ALIGN_UP(Virtual Size, Phys Sector Size) байт. Вообще-то, все пункты кроме первого – грубые ошибки реализации, но и… суровая реальность бытия вместе с тем, поэтому таких ситуаций лучше всего избегать. Физический размер должен быть выровнен на величину File Alignment, выравнивать виртуальный размер необязательно (загрузчик выравнивает его автоматически). Однако, и это правило не обходится без исключений: если физический размер меньше или равен виртуальному, то и его выравнивать необязательно, правда, смысла в этом немного, поскольку начало следующей секции в файле по любому должно быть выровнено на величину File Align.

Виртуальный адрес следующей секции обязательно должен быть равен виртуальному адресу предыдущей секции плюс ее размер, выровненный на величину Section Alignment. Секции не могут ни перекрываться, ни образовывать виртуальные дыры. На физические адреса

секций таких ограничений не наложено и они могут быть разбросаны по файлу в живописном беспорядке. Впрочем, увлекаться разбрасыванием право же не стоит – не ровен час системный загрузчик запутается и откажет файлу в загрузке, если еще не выпадет в синий экран.

Кстати, на счет синих экранов. Напомним читателю, что если Section Alignment < 1000h, а физический размер секции вылетает за пределы файла, W2K SP3 (и вероятно все остальные представители линейки NT) выбрасывают синий экран и системе наступает кранты.

Поле **Characteristics** определяет атрибуты доступа к секции и особенности ее загрузки. Имеются три атрибута как будто бы определяющих содержимое секции как код, инициализированные и неинициализированные данные (IMAGE_SCN_CNT_CODE/20h, IMAGE_SCN_CNT_INITIALIZED_DATA/40h, IMAGE_SCN_CNT_UNINITIALIZED_DATA/80h соответственно). Однако, системный загрузчик игнорирует их значение и потому опираться на них ни в коем случае нельзя. Теоретически секция неинициализированных данных при отсутствии прочих атрибутов не должна грузиться с диска, но… ведь грузиться!

Некоторые вирусы/упаковщики/протекторы определяют кодовую секцию по наличную атрибута IMAGE_SCN_CNT_CODE. Что ж! Не такое уж и плохое решение, только будьте готовы к тому, что этого атрибута не окажется ни у одной из секции (что встречается достаточно часто), либо же он будет присвоен секции данных (что встречается пореже, но все-таки встречается).

Другая триада атрибутов описывает права доступа ко всем страницам секции, назначаемым системным загрузчиков по умолчанию (будучи загруженным, файл может свободно манипулировать ими вызывая API-функцию VirtualProtectEx). В настоящее время определено три атрибута: исполнения, чтения и записи (IMAGE_SCN_MEM_EXECUTE/20000000h, IMAGE_SCN_MEM_READ/40000000h, IMAGE_SCN_MEM_WRITE/80000000h). На платформе Intel атрибуты чтения/исполнения полностью эквивалентны и соответствуют аппаратному атрибуту доступности (accessible) страницы. Атрибут записи обрабатывается вполне естественным образом. Следовательно, отличить секцию кода от секции данных в общем случае невозможно и приходится действовать исподтишка, объявляя секцией кода ту, в которую указывает точка входа.

Два других интересных атрибута это – IMAGE_SCN_MEM_DISCARDABLE/2000000h (после загрузки файла секция может быть уничтожена в памяти) и IMAGE_SCN_MEM_SHARED/10000000h (секция является совместно используемой).

Атрибут IMAGE_SCN_MEM_DISCARDABLE обычно присваивается секциям, содержащим вспомогательные структуры данных такие как, например, таблица перемещаемых элементов, необходимые лишь на этапе загрузки файла и впоследствии никем не используемые. А раз так – зачем они будут жрать память? Фатальная ошибка подавляющего большинства вирусов состоит в том, что внедрясь в последнею секцию файла (коей как раз DISCARDABLE-секция обычно и оказывается), они не проверяют ее атрибутов не "выкупают" права на память. Операционная система в любой момент может выгрузить оккупированные ими страницы и тогда инфицированный процесс рухнет, выдавая хорошо известное всем сообщение о критической ошибке приложения.

Атрибут IMAGE_SCN_MEM_SHARED намного менее безобиден, но тоже с характером и помещать сюда исполняемый код категорически не рекомендуется. Во-первых, в любой момент он может быть затер посторонним процессом и тогда зараженное приложение опять-таки рухнет, а, во-вторых, Windows 9x насильно перегоняет SHARED-секции в верхнюю половину адресного пространства и действительный адрес загрузки уже не будет соответствовать виртуальному адресу секции (правда, полностью перемещаемый код в таких условиях вполне сможет работать).

Остальные атрибуты либо неинтересны, либо имеют отношение только к объективным coff-файлам (не PE) и потому здесь не рассматриваются. Это в частности относится к атрибутам из семейства IMAGE_SCN_ALIGN_xBYTES, индивидуально настраивающим кратность выравнивая каждой секции. Для объективных файлов это, быть и может и так, но системный загрузчик эти атрибуты в упор игнорирует.

Поля **PointerToRelocations/NumberOfRelocations** (указатель на таблицу перемещаемых элементов и количество элементов в этой таблице соответственно) имеют отношение только к объективным файлам, а исполняемые файлы и динамические библиотеки управляют своими перемещаемыми элементами через одноименную запись в DATA_DIRECTORY, поэтому эти поля могут содержать любые значения. Некоторые

вирусы/упаковщики/проекторы помечают таким образом свои файлы, чтобы их не обрабатывать дважды. Способ глупый и ненадежный (задумайтесь, что произойдет с файлом после его упаковки любым посторонним упаковщиком?).

Поля **PointerToLinenumbers/NumberOfLinenumbers** (указатели на таблицу номеров строк и количество элементов в этой таблице соответственно) ранее использовались для хранения отладочной информации, связывающей номера строк исходной программы с адресами откомпилированного файла. В настоящее время используется только в объективных файлах, а в исполняемых файлах отладочная информация хранится совсем в другом месте и в другом формате.

Ниже приведен код, сканирующий таблицу секций и выводящий извлеченную информацию на терминал.

```
#define xopt_sz(p)            (*((WORD*)(p + 0x14 /* size of optional header */)))
#define pSectionTable(p)      ((BYTE*) (xopt_sz(p) + 0x18 /*sizeofimageheafer*/ + p))
#define pFirstSection(p)      (pSectionTable(p))
#define pLastSection(p)       (pSectionTable(p) + (xNumOfSec(p) - 1) * 40)
```

Листинг 6 макросы, возвращающие указатели на IMAGE_SECTION_HEADER первой и последней секции файла

```
a = xNumOfSec(p); pNextSection = pFirstSection(p);
while(a--)
{
        printf( "Name: %s\n"\
                "\tVirtualSize           : %04Xh RVA\n"\
                "\tVirtualAddress        : %04Xh RVA\n"\
                "\tSizeOfRawData         : %04Xh RVA\n"\
                "\tPointerToRawData      : %04Xh RVA\n"\
                "\tPointerToRelocations  : %04Xh RVA\n"\
                "\tPointerToLinenumbers  : %04Xh RVA\n"\
                "\tNumberOfRelocations   : %04Xh RVA\n"\
                "\tNumberOfLinenumbers   : %04Xh RVA\n\n",
                pNextSection,         pNextSection[0x8],     pNextSection[0xC],
                pNextSection[0x10],   pNextSection[0x14],    pNextSection[0x18],
                pNextSection[0x1C],   pNextSection[0x20],    pNextSection[0x14]);

        pNextSection+=40;  // следующий элемент Section Table
}
```

Листинг 7 прогулка по таблице секций с выводом ее содержимого на терминал

экспорт

Таблица экспорта представляет собой сложную иерархическую структуру, каждый из компонентов которой может быть расположен в любом месте страничного имиджа, хотя по спецификации она должна быть сосредоточена в одной области. Когда-то таблице экспорта выделялась своя персональная секция .edata, но теперь этого правила практически никто не придерживается, поэтому говорить о секции импорта не совсем корректно (впрочем, если вы назовете директорию секций большой беды не будет и все вас поймут).

На вершине иерархии находится структура IMAGE_EXPORT_DIRECTORY, так же известная под именем export directory table, содержащая указатели на три подчиненные структуры: ***таблицу экспортируемых имен*** (Name Pointer), ***таблицу экспортируемых ординалов*** (Ordinal Table) и ***таблицу экспортируемых адресов*** (Export Address Table). Поле Name RVA указывает на строку с именем динамической библиотеки, которое судя по всему игнорируется и может принимать любые значения.

Экспорт функций/данных может производится как по их имени, так и по ординалу. Таблицы имен и адресов представляют собой массивы из RVA-указателей, ссылающихся на ASCIIZ-строки с именами функций и адреса экспортируемых функций/данных соответственно. Таблица ординалов представляет собой массив 16-битных индексов (ординалов) и служит своеобразным связующим звеном между таблицей имен и таблицей адресов. Пусть i-элемент таблицы имен указывает ASCIIZ-строку с именем интересующей нас функции "my_func", тогда i-элемент таблицы ординалов содержит индекс элемента таблицы адресов с RVA-адресом функции my_func или говоря другими словами ее ordinal.

В переводе на язык Си это выглядит так:

```
i = Search_ExportNamePointerTable (ExportName);
ordinal = ExportOrdinalTable [i];
SymbolRVA = ExportAddressTable [ordinal - OrdinalBase];
```

Листинг 8 экспорт по именам

Если нам известен ординал функции, то обращаться к таблицам имен/ординалов необязательно. Определенная путаница связана с тем, что ординал задает отнюдь не индекс в таблице ординалов, а индекс в таблице адресов. Таблица ординалов представляет собой вспомогательную подструктуру не имеющую самостоятельной ценности и всегда использующуюся только в паре с таблицей имен. Поэтому, таблицы имен и ординалов всегда содержат одинаковое количество элементов, задаваемое полем Number of Name Pointers, которое может и не совпадать с количеством элементов таблицы адресов, задаваемое полем Export Address Table RVA.

Теперь о тонкостях. Таблица адресов может содержать "разрывы", т. е. элементы обращенные в нуль и указывающие в никуда. К счастью, их легко отсеять. Хуже, что далеко не всякий элемент таблицы адресов представляет собой действительный адрес экспортируемой функции, ведь динамические библиотеки поддерживают форвардинг (forwarding), т. е. сквозное перенаправление экспорта в другую DLL и тогда соответствующий элемент таблицы адресов содержит RVA-адрес ASCIIZ-строки типа "NTDLL.RtlDeleteCriticalSection", описывающей переназначение. Как отличить forward-строки от действительных адресов экспортируемых функций? Да очень просто, forward-строки всегда расположены внутри таблицы экспорта (именно по этому спецификация настоятельно рекомендует делать ее непрерывной, никаких других причин для этого у системного загрузчика нет). Размер таблицы экспорта содержится в DATA_DIRECTORY, там же где находится адрес export directory table и разоблачение forward-строк осуществляется тривиально.

Приведенный ниже демонстрационный пример сканирует всю таблицу экспорта, отображая ее на экране в удобно читаемом виде. Обратите внимание, что обработка ordinal BASE несколько изменена на идеологически более правильную:

```
// получаем указатель на PE
p = *(DWORD*)(pBaseAddress + 0x3C /*e_lfanew */) + pBaseAddress;

// получаем указатель на DATA_DIRECTORY
pDATA_DIRECTORY = (DWORD*)(p + 0x78);

// получаем указатель на экспорт
pExport = pDATA_DIRECTORY[0] + pBaseAddress;
xExport = pDATA_DIRECTORY[1]; // берем размер, но не проверяем

// извлекаем сведения об основных структурах
nameRVA               = *(DWORD*) (pExport + 0xC) + pBaseAddress;
ordinalBASE           = *(DWORD*) (pExport + 0x10);
addressTableEntries   = *(DWORD*) (pExport + 0x14);
numberOfNamePointers  = *(DWORD*) (pExport + 0x18);
exportAddressTableRVA =  (DWORD*) (*(DWORD*) (pExport + 0x1C) + pBaseAddress);
namePointerRVA        =  (DWORD*) (*(DWORD*) (pExport + 0x20) + pBaseAddress);
ordinalTableRVA       =  (WORD* ) (*(DWORD*) (pExport + 0x24) + pBaseAddress);

// распечатываем все имена/ординалы/адреса
printf( "name                          ordinal/hint VirtualAddress Forward\n"\
        "------------------------------------------------------------------\n");

for (a = 0; a < _MAX(addressTableEntries, numberOfNamePointers); a++)
{
        // два вида обработки - по именам и по ординалам
        if (a < numberOfNamePointers)
        {
                // выделение индекса функций, экспортируемых по именам
                name = namePointerRVA[a] + pBaseAddress; f_index = ordinalTableRVA[a];
        }
                else
        {
                // выделение индекса функций, экспортируемых только по ординалам
                name = "n/a";  f_index = a;
        }

        // определение адреса функции
```

```
        f_address =  (DWORD)(exportAddressTableRVA[f_index] + pBaseAddress);

        // поиск "разрывов" в таблице адресов
        if (f_address == pBaseAddress) continue;

        // определение оридинала
        ordinal = f_index + ordinalBASE;

        // поиск форвардов (если есть)
        if ((f_address > (DWORD) pExport) && (f_address < (DWORD) (pExport + xExport)))
                       pForward = (BYTE*)f_address; else pForward  = 0;

        // вывод результатов на терминал
        printf("%-30s [%03d/%03d] %08Xh %s\n",
                              name, ordinal, a, f_address, (pForward)?pForward:"");

} printf("================================================================\n");
```

Листинг 9 простейший разбор таблицы экспорта

импорт

Если с экспортом все более или менее понятно, но импорт это какой-то кошмар. Это целых три различных механизма, один страшнее другого, управляемые четырьмя записями в DATA_DIRECTORY.

Стандартный механизм импорта работает приблизительно так: специальная таблица (называемая таблицей импорта) перечисляет имена/ординалы всех импортируемых функций, указывая в какое место страничного имиджа загрузчик должен записать эффективный адрес каждой из них. Это просто, но до ужаса тормозно. Грубо говоря, на каждую импортируемую функцию приходится один вызов GetProcAddress, фактически сводящийся к поэлементному перебору всей таблицы экспорта.

Более производителен механизм диапазонного импорта (**bound import**), сводящийся к тривиальному проецированию необходимых библиотек на адресное пространство процесса, с жесткой прошивкой экспортируемых адресов еще на стадии компиляции приложения. Это быстро, но не универсально. Перекомпиляция DLL требует обязательной перекомпиляции приложения, поскольку по старым адресам теперь ничего хорошего уже нет.

Между двумя этими крайностями окопался механизм отложенного импорта (**delay import**), реализованный с большим количеством ошибок, поддерживаемых далеко не всеми компоновщиками, но все-таки работающий. В общих чертах основная идея заключается в перенаправлении элементов таблицы импорта на специальный обработчик, динамически загружающий соответствующие функции по мере возникновения в них необходимости и подставляющий их адреса в таблицу импорта.

Приоритет различных механизмов импорта не определен и загрузчик вправе использовать любой доступный, переходя к другому только в случае неудачи. Эксперимент показывает, что Windows 9x/NT сначала используют bound import и только если штамп времени/предпочтительный адрес загрузки импортируемой библиотеки не совпал с ожидаемым, пытается импортировать функции обычным путем. Windows XP поступает иначе и после неудачи с bound import'ом, пытается импортировать функции непосредственно по таблице адресов, указатель на которую содержится в поле IMAGE_DIRECTORY_ENTRY_IAT. Штатно таблица адресов содержит копию таблицы имен и потому обращаться к последней нет никакой необходимости. Если же это не так, загрузчик вынужден импортироваться обычным путем.

Короче говоря, секса (с отладчиком) будет предостаточно, но мы все-таки начнем… Стандартная таблица импорта представляет собой сложную иерархическую структуру, каждый из элементов которой может быть расположен в любом месте страничного имиджа.

На вершине иерархии находится структура Import Directory Table, представляющая собой массив структур IMAGE_IMPORT_DESCRIPTOR, завершаемых нулевым элементом. Каждый IMAGE_IMPORT_DESCRIPTOR содержит ссылки на две подчиненные структуры – **lookup-таблицу**, содержащую имена и/или ординалы импортируемых функций, и таблицу **импортируемых адресов**, так же известную как Thunk Table и содержащую RVA-адреса ячеек страничного имиджа, поверх которых загрузчик должен записать эффективные адреса соответствующих им функций. Пусть необходимая нам функция my_func находится в i-элементе lookup-таблицы, тогда i-индекс таблицы импортируемых адресов содержит RVA-указатель на ячейку, куда загрузчику следует записать ее адрес.

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
union {
        DWORD   Characteristics;       // 0 for terminating null import descriptor
        DWORD   OriginalFirstThunk;    // RVA to original unbound IAT
        };
        DWORD   TimeDateStamp;         // 0 if not bound,
                                       // -1 if bound, and real date\time stamp
                                       // in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (new)
                                       // O.W. date/time stamp of DLL bound to (old)
        DWORD   ForwarderChain;        // -1 if no forwarders
        DWORD   Name;
        DWORD   FirstThunk;            // RVA to IAT
} IMAGE_IMPORT_DESCRIPTOR;
```

Листинг 10 прототип структуры IMAGE_IMPORT_DESCRIPTOR

Имя загружаемой DLL содержится в поле Name структуры IMAGE_IMPORT_DESCRIPTOR, представляющим собой RVA-указатель на ASCIIZ-строку.

Остальные поля не так интересы. Если временная отметка TimeDateStamp равна нулю (как чаще всего и бывает), то системный загрузчик обрабатывает таблицу импорта по всем правилам. Если же она равна минус одному (FFFFFFFFh), загрузчик игнорирует указатели OriginalFirstThunk и FirstThunk, полгая, что данная библиотека импортируется через BOUND_IMPORT и только лишь когда BOUND_IMPORT провалится (например, из-за несовпадение TimeDateStamp) возвращается к IAT.

На этом основан один любопытный пример противостояния отладчикам и дизассемблерам –сбрасываем TimeDateStamp в FFFFFFFFh, добавляем в BOUND_IMPORT импорт библиотеки, указанной в Name, ставим в BOUND_IMPORT'e TimeDateStamp в ноль, чтобы гарантированно загрузить ее (конечно, значения экспортируемых адресом в различных версиях DLL могут и не совпадать, ну и хвост с ними, главное что библиотека спроецирована на адресное пространство процесса, а разгрести экспорт можно и руками). Теперь искажаем указатели OriginalFirstThunk и FirstThunk, придавая им заведомо некорректное значение. Системный загрузчик, обнаружив, что TimeDateStamp == -1, просто проигнорирует их обработает такой файл вполне нормально. Дизассемблеры/отладчики иное дело. О BOUND_IMPORT'e подавляющее большинство из них ничего не знает и, честно ринувшись в IAT, они в лучшем случае сообщат, что таблица импорта искажена, а в худшем – поедут крышей и аварийно завершат свою работу. Старые версии ИДЫ и hiew'a на этот ломались только так. Новые – нет, поэтому этот трюк уже утратил свою былую актуальность.

Любое другое значение TimeDateStamp обозначает действительную временную метку и, если она совпадает с временной меткой импортируемой DLL, загрузчик просто проецирует ее на адресное пространство процесса, не настраивая таблицу адресов. Предполагается, что эффективные адреса заданы еще на времени компиляции. На этом основан другой хитрый трюк (все еще актуальный). Подменив один или несколько элементов таблицы адресов адресом другой функции, мы введем дизассемблер в глубокое заблуждение (ведь он игнорирует таблицу адресов и предпочитает разбирать весь импорт самостоятельно).

ForwarderChain – очень странное поле, вроде бы имеющее отношение к форвардингу функций. По одним данным индекс в цепочке форварда, по другим – RVA-указатель на массив IMAGE_IMPORT_BY_NAME. Обычно равно нулю (нет здесь никакого форварда), так что навряд ли это указатель, скорее уж адрес. Хотя спецификация и утверждает, что за отсутствием форварда закреплено значение FFFFFFFFh, линкеры похоже придерживаются совершенно иного мнения. Что же до системного загрузчика, то это поле он попросту игнорирует и здесь может быть все, что угодно. Тоже самое относится и к отладчикам/дизассемблерам.

Пример практической работы с таблицей импорта приведен ниже:

```
// ПЕЧАТАЕМ ТАБЛИЦУ ИМПОРТА
n2k_print_IAT(DWORD* importLookupTable, DWORD* importAddressTable, BYTE* pBaseAddress)
{

        DWORD lookup, hint, address;
        BYTE  *name;  char  buf[MAX_BUF_SIZE];
        name   = "not pressent"; lookup = address = hint = 0;

        printf( " hint  name/ordinal                    address\n"\
                "-----------------------------------------------\n");

        while(1)              // сканируем таблицу импорта пока не встретим нуль
        {
```

```
                // извлекаем очередные элементы из loockup и address таблиц
                if (importLookupTable) lookup = *importLookupTable++;
                if (importAddressTable)       address= *importAddressTable++;
                if (!address) break; // это конец?

                if (importLookupTable)
                {
                        if (lookup & 0x80000000) // функция экспортируется по ординалу
                        {
                                sprintf(buf,"#%d",lookup & ~0x80000000);name=buf;hint=0;
                        }
                        else                    // функция экспортируется по имени
                        {
                                name=(lookup+pBaseAddress+2);
                                hint=*((WORD*)(lookup+pBaseAddress));
                        }
                } printf("[%04d] %-30s:%08Xh\n",hint, name, address);
        }printf("================================================\n\n");
}

// прогуливаемся по таблице импорта, выводя ее всю
n2k_walk_idex(BYTE* pImport, BYTE* pBaseAddress)
{
        int   a;
        BYTE  *nameRVA;
        DWORD *importLookupTable;
        DWORD *importAddressTable;

        // перебираем все таблицы дескрпиторов сколько их есть там...
        while(1)
        {
                // извлекаем основные параметры
                nameRVA               =*(DWORD*)(pImport + 0x0C) + pBaseAddress;
                importLookupTable     =(DWORD*)(*(DWORD*)(pImport+0x00)+ pBaseAddress);
                importAddressTable    =(DWORD*)(*(DWORD*)(pImport+0x10)+ pBaseAddress);

                //printf("%s %x %x\n",nameRVA,importLookupTable,pBaseAddress);

                // переходим на следующий дескриптор
                pImport += 0x14 /* size of _IMAGE_IMPORT_DESCRIPTOR */;

                if ((BYTE*)importLookupTable == pBaseAddress) break; // это конец?

                // печатаем имя DLL
                printf("%s:\n",nameRVA);
                for(a=0;a<strlen(nameRVA);a++) printf("-"); printf("\n");

                // печатаем импортируемые функции
                n2k_print_IAT(importLookupTable, importAddressTable, pBaseAddress);
        }
}
```

Листинг 11 дампер таблицы импорта

IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT:

BOUND_IMPORT до ужаса незамысловат и прост. С ним связана всего один массив структур IMAGE_BOUND_IMPORT_DESCRIPTOR, состоящая из трех полей: временной отметки; смещения имени DLL, отсчитываемом от начала таблицы BOUND_IMPORT'a и количество форвардов, точное назначение которых неясно.

Если временная отметка импортируемой библиотеки соответствует с ее собственной временной отметкой, прописанной в PE-заголовки, загрузчик просто проецирует последнюю на адресное пространство и умывает руки, предоставляя программе действовать самостоятельно. Захочет – будет разбирать таблицу экспорта импортируемой библиотеки вручную, захочет – жестко пропишет экспортируемые адреса еще на этапе компиляции, как обычно и происходит.

Нулевое значение временной отметки соответствуют любому времени и обращаться с ним следует предельно осторожно, ибо при перекомпиляции библиотеки жестко прописанные адреса будут указывать в космос и программа повиснет.

```
typedef struct _IMAGE_BOUND_IMPORT_DESCRIPTOR {
    DWORD   TimeDateStamp;
    WORD    OffsetModuleName;
    WORD    NumberOfModuleForwarderRefs;
```

```
// Array of zero or more IMAGE_BOUND_FORWARDER_REF follows
} IMAGE_BOUND_IMPORT_DESCRIPTOR,  *PIMAGE_BOUND_IMPORT_DESCRIPTOR;
```

Листинг 12 прототип структуры IMAGE_BOUND_IMPORT_DESCRIPTOR

Практический пример работа таблицы BOUND_IMPORT'а приведен ниже:

```
n2k_walk_bound(BYTE *pBound, BYTE *pBaseAddress)
{
        DWORD time_x; WORD name_offset; WORD n_ref;
        if (!pBaseAddress) pBaseAddress = pBound;

        while(1)              // разбираем bound'ы
        {
                // извлекаем все значения
                time_x = *(DWORD*) pBound;    n_ref = *((WORD*) (pBound+6));
                name_offset = *((WORD*) (pBound+4)); if (!name_offset) break;

                // выводим их на терминал
                printf("[%04X] %-30s %d\n",time_x, name_offset + pBaseAddress, n_ref);

                // следующий элемент
                pBound += 8;
        } printf("\n");
}
```

Листинг 13 простой дампер таблицы диапазонного импорта

IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT:

Вот мы и добрались до отложенного импорта… Рассмотрим его лишь вкратце, поскольку ничего хорошего ожидать все равно не приходится. Для экспериментов нам понадобиться по меньшей мере один файл с отложенным импортом. Если же такого в вашем распоряжении нет, создайте его самостоятельно. Пользователи Microsoft Linker могут поступить так: link dll.test.impl.obj /DELAYLOAD:dll.dll dll.lib DELAYIMP.LIB, а пользователи ликера ulink от Юрия Харона (которым я сам давно пользуюсь и который всем настоятельно рекомендую), так: ulink -d dll.test.impl.obj dll.lib.

```
typedef struct ImgDelayDescr {
        DWORD           grAttrs;        // attributes
        LPCSTR          szName;         // pointer to dll name
        HMODULE*        phmod;          // address of module handle
        PimgThunkData   pIAT;           // address of the IAT
        PCImgThunkData pINT;            // address of the INT
        PCImgThunkData pBoundIAT;       // address of the optional bound IAT
        PCImgThunkData pUnloadIAT;      // address of optional copy of original IAT
        DWORD           dwTimeStamp;    // 0 if not bound,
                                        // O.W. date/time stamp of DLL bound to Old BIND
} ImgDelayDescr, * PImgDelayDescr;
```

Листинг 14 прототип структуры ImgDelayDescr

Поле grAttrs задает тип адресации, применяющийся в служебных структурах отложенного импорта (0 – VA, 1 – RVA); поле szName содержит RVA/VA-указатель на ASCIIZ-строку с именем загружаемой DLL (тип адреса определяется особенностями реализации конкретного delay helper'a, внедряемого в программу линкером и варьирующегося от реализации к реализации, короче говоря, будьте готовы ко всяким пакостям). В изначально пустое поле phmod загрузчик (все тот же Delay Helper) помещает дескриптор динамически загружаемой DLL.

Поле pIAT содержит указатель на таблицу адресов отложенного импорта, организованную точно так же, как и обычная IAT, с той лишь разницей, что все элементы таблицы отложенного импорта ведут к delay load helper'y – специальному динамическому загрузчику, так же называемому переходником (thunk), который вызывает LoadLibrary (если только библиотека уже не была загружена), а затем дает GetProcAddress и замещает текущий элемент таблицы отложенного импорта эффективным адресом импортируемой функции, благодаря чему все последующие вызовы данной функции осуществляется напрямую в обход delay load helper'a.

При выгрузке DLL из памяти, последняя может восстановить таблицу отложенного импорта в исходное состояние, обратившись к ее оригинальной копии, RVA-указатель на которую хранится в поле pUnloadIAT. Если же копии нет, ее указатель будет обращен в ноль.

Поле pINT содержит RVA-указатель на таблицу имен, во всем повторяющую стандартную таблицу имен (см. name Table). Тоже самое относится и к полю pBoundIAT, хранящим RVA-указатель на таблицу диапазонного импорта. Если таблица диапазонного импорта не пуста и указанная временная метка совпадает с временной меткой соответствующей DLL, системный загрузчик просто проецирует ее на адресное пространство данного процесса и механизм отложенного импорта дезактивируется.

```
// прогуливаемся по таблице delay-импорта
n2k_walk_delay(BYTE* pDelay, BYTE *pBaseAddress)
{

        WORD  a = 0, hint;
        BYTE  *name, *f_name;
        DWORD attr,  ordinal;
        char  buf[MAX_BUF_SIZE];
        DWORD *INT,  *IAT, *f_addr;

        //attr = *(DWORD*)pDelay;

        while(1)
        {
                // извлекаем указатели на IAT и INT
                IAT = (DWORD*)*((DWORD*)(pDelay + 0x0C));
                INT = (DWORD*)*((DWORD*)(pDelay + 0x10));

                // извлекаем указатель на имя модуля
                name = (BYTE*) *((DWORD*) (pDelay + 0x04));

                // это конец?
                if (!IAT || !INT) break;

                // эвристическое распознавание адреса
                if ((DWORD) name < (DWORD) pBaseAddress) name += (DWORD) pBaseAddress;
                if ((DWORD) IAT  < (DWORD) pBaseAddress)
                                IAT   = (DWORD*)((DWORD) IAT + (DWORD) pBaseAddress);

                if ((DWORD) INT  < (DWORD) pBaseAddress)
                                INT   = (DWORD*)((DWORD) INT + (DWORD) pBaseAddress);

                // печатаем имя модуля
                printf("%s\n",name);for(a;a<strlen(name);a++)printf("-");printf("\n");

                printf( " hint  name/ordinal                    address\n"\
                                "-------------------------------------------------\n");

                // печать имен
                while(1)
                {

                        f_name = (BYTE*) *INT++; f_addr = (DWORD*) *IAT++;
                        if (!f_name || !f_addr) break;

                        if ((DWORD)f_name < (DWORD)pBaseAddress)
                                f_name += (DWORD) pBaseAddress;

                        if ((DWORD)f_addr < (DWORD)pBaseAddress)
                                f_addr = (DWORD*)((DWORD)f_addr+(DWORD) pBaseAddress);

                        if ((DWORD) f_name & 0x80000000)
                        {
                                sprintf(buf, "#%d",((DWORD) f_name) & 0xFFFF);
                                f_name = buf; hint = 0;
                        }
                                else
                        {
                                hint = *(WORD*) f_name;  f_name = &f_name[2];
                        }
                        printf("[%04d] %-30s:%08Xh\n",hint,f_name, f_addr);
                }       printf("=================================================\n\n");
                pDelay += 0x20;  // следующий элемент
        }
}
```

Листинг 15 простейший дампер таблицы отложенного импорта

перемещаемые элементы

Таблица перемещаемых элементов не является обязательной и используется только кода загрузка по адресу, прописанному в image base оказывается невозможной. Тогда системный загрузчик обращается к таблице перемещаемых элементов, представляющий собой массив указателей на RVA-адреса страничного имиджа, требующих коррекции и увеличивает их на разницу предполагаемого и фактического адресов загрузки.

Допустим, в программном коде имелась инструкция типа mov eax, [401000h], где 401000h – абсолютный адрес ячейки памяти страничного имиджа. Если файл будет загружен не по адресу 400000h, на который он и рассчитывал, а, скажем, по адресу 10000000h, ячейка 401000h в обязательном порядке должна быть скорректирована, иначе в регистр eax попадет совершенно непредсказуемое значение. Вычислив дельту загрузки (10000000h – 400000h == FC00000h) и выудив из таблицы перемещаемых элементов RVA-адрес корректируемой ячейки, системный загрузчик складывает его с дельтой загрузки и получат: mov eax, [10001000h].

При внедрении в файл путем замещения секции (например, сжатии и/или сбрасывании части ее содержимого в оверлей) это создает следующие проблемы. Первое и главное: если хотя бы один перемещаемый элемент попадет внутрь внедренного нами кода и файл будет действительно перемещен, внедренный код окажется полностью или частично испорчен и его поведение станет непредсказуемым (все зависит от того, куда придется "ранение"). Во-вторых, даже если он и выживет, то восстановленная секция окажется неработоспособной, ведь соответствующие адреса не были скорректированы.

Многие руководства советуют либо прибивать таблицу перемещаемых элементов, обнуляя поле IMAGE_DIRECTORY_ENTRY_BASERELOC в DATA_DIRECTOTY (но это делает файл немобильным), либо же вовсе не связываться с файлами, содержащими таблицу перемещаемых элементов (но это не по хакерски). Можно ли, запретить системному загрузчику гробить внедренный нами код, не лишая файл свойства перемещаемости? Оказывается можно – достаточно создать пустую таблицу перемещаемых элементов, переустановив на нее IMAGE_DIRECTORY_ENTRY_BASERELOC, а оригинальную таблицу перемещаемых элементов обрабатывать самостоятельно, деля это уже после того, как все секции будут приведены в исходное состояние (распакованы и/или извлечены из оверлея).

Почему подложная таблица перемещаемых элементов должна быть пустой? Потому что, системный загрузчик содержит грубую ошибку и в отсутствии таблицы перемещаемых элементов не перемещает файл вообще (а ведь по спецификации – должен). Разумеется, внедряемый код необходимо спроектировать с учетом непостоянства базового адреса загрузки, т. е. использовать абсолютную адресацию нельзя и необходимо либо ограничиться одной относительной, либо автоматически определять место своей дислокации в памяти в дальнейшем плясать уже от него.

К сожалению, микропроцессоры семейства I386 с перемещаемым кодом не в ладах, т. к. ориентированы на абсолютную адресацию и налагают запрет на явное использование регистра EIP (указатель следующей выполняемой машинной инструкции). Мы не можем сказать процессору: mov eax, [eip+666h] (занести в регистр eax двойное слово, лежащие на 666h байт ниже следующей исполняемой команды) и приходится прибегать ко всевозможным ухищрениям проталкивая регистр EIP через стек, добираясь до него так: call @label/@label:pop eax, что эквивалентно: mov [esp], eip/mov eax,[esp], где esp – указатель вершины стека. Кстати, о стеке. Это удобное хранилище данных, не требующее к тому же задания абсолютных адресов.

По соображениям эффективности таблица перемещаемых элементов храниться в упакованном формате, вместо массива 32-разрядных RVA-адресов, указывающих на модифицируемую ячейку памяти внутри страничного имиджа, мы имеем массив 16-разрядных слов, 4 старших бита которых задают тип перемещаемой ячейки, а 12 младших бит – смещение, отсчитываемое от начала страницы (page). Под "станицей" здесь понимается отнюдь не станицы памяти, а непрерывный регион памяти, RVA-адрес которого задается внутри специальной структуры. Таким образом, таблица перемещаемых элементов состоит из одного или нескольких последовательно расположенных блоков. В начале блока идет его RVA-адрес и размер, а за ними 16-битный массив упакованных смещений.

Заглянув в файл WINNT.H, мы обнаружим структуру _IMAGE_BASE_RELOCATION, определенную так (не путайте ее с _IMAGE_RELOCATION, относящуюся к объективным файлам):

```
typedef struct _IMAGE_BASE_RELOCATION {
        DWORD   VirtualAddress;
```

```
        DWORD   SizeOfBlock;
        //  WORD        TypeOffset[1]; // массив упакованных перемещаемых элементов
} IMAGE_BASE_RELOCATION;
```

Листинг 16 прототип структуры IMAGE_BASE_RELOCATION

```
TypeOffset[(SizeOfBlock - sizeof(VirtualAddress) - sizeof(SizeOfBlock))/sizeof(WORD)]
```

Листинг 17 истинный размер массива TypeOffset

I386-загрузчик поддерживает двенадцать типов перемещаемых элементов, но на практически обычно используется лишь один из них: IMAGE_REL_BASED_HIGHLOW (03h), указывающий на младший байт 32-разрядного значений, к которого следует добавить дельту загрузчики. В переводе на межсистемный программистский это звучит так: `if ((TypeOffset[i] >> 12) == 3 ) *(DWORD*) ((TypeOffset[i] & ((1<<12)-1)) + pageRVA + (DWORD) pBaseAddress) += ((DWORD) pBaseAddress - (DWORD)pPreferAddress)`. Когда будете это делать, не забудьте, предварительно убедиться, что соответствующая страница памяти имеет атрибут Writable и, если его нет, временно измените атрибуты страницы, обратившись к API-функции функции VirtualProtectEx, а после исправления всех перемещаемых элементов, верните атрибуты назад.

Остальные типы перемещаемых элементов описаны в спецификации на PE-файл. Обещаю, что вы узнаете много интересного. В частности, перемещаемые элементы типа IMAGE_REL_BASED_HIGHADJ хранят целевой адрес сразу в двух TypeOffset'ах. Первый указывает на ячейку, содержащую старшее перемещаемое слово, а второй – младшее. На I386-процессорах такая комбинация не имеет никакого смысла (разве, что специально со встроенным ассемблером поизвращаетесь), но на других платформах может быть широко распространена.

Ниже приведен исходный текст простейшего дампера таблицы перемещаемых элементов:

```
n2k_walk_reloc(BYTE* pReloc, BYTE *pBaseAddress, BYTE *pPreferAddress)
{

        BYTE *pageRVA; DWORD a, blockSize, typeX, offsetX;

        // вычисляем дельту загрузки
        printf( "\ndelta := %08Xh\n"\
                "==================\n\n",pBaseAddress - pPreferAddress);

        // перебираем все fixup блоки один за другим
        while(1)
        {
                // вычисляем адрес начала страницы и размер блока
                pageRVA = (BYTE*)(*(DWORD*) pReloc); blockSize = *(DWORD*) (pReloc+4);

                if (!blockSize) break;                // это конец?

                // распаковываем перемещаемые элементы,
                // вычисляя адреса корректируемых ячеек
                printf( "FIXUP BLOCK - pageRVA: %06Xh, size %06d bytes\n"\
                        "-------------------------------------------------\n",
                        pageRVA, blockSize);

                for (a = 8; a < blockSize; a += 2)
                {
                        // извлекаем тип fixup'a и смещение относительно pageRVA
                        typeX   = (*(WORD*)(pReloc + a)) >> 12;
                        offsetX = (*(WORD*)(pReloc + a)) & ((1<<12)-1);

                        // обработка разных типов fixup'ов
                        switch(typeX)
                        {
                        case 0: printf("\tIMAGE_REL_BASED_ABSOLUTE\n");
                                break;

                        case 3: printf("\tIMAGE_REL_BASED_HIGHLOW  @ %08Xh --> %08Xh\n",
                                offsetX + pPreferAddress, offsetX + pBaseAddress);
                                break;
                        default:
                                printf("\t%x - not supported\n", typeX);
                                break;
                        }
                } printf("\n");
```

```
                // берем следующий блок
                pReloc += blockSize;
        }
}
```

Листинг 18 разбор таблицы перемещаемых элементов

заключение

…уф! наконец-то мы добрались до кочки, одиноко торчащей среди топкого болота. Теперь можно обсохнуть, отмыться, собраться с мыслями и какое-то время передохнуть. Вы еще не передумали писать свой вирус? Да уж! Такой марш бросок любе влечение угробит… И правильно! В этом мире выживают лишь те, чье стремление разобраться в системе доминирует над желанием напакостить ближнему своему. Написать грамотный и во всех отношениях корректный "внедритель" ох как непросто! И пока вы будете переваривать полученную информацию, незаметно подоспеет следующий номер со следующей порцией информационного концентрата. Когда они соединяться вместе произойдет своеобразная алхимическая реакция и на свет родится крохотный организм огромного кибернетического мира. Конкретно, в статье будет показано именно осуществляется внедрение машинного кода в посторонние тело.