трепанация двоичных файлов под Linux

крис касперски, aka мыщъх, aka nezumi, no-email

рост коммерческого программного обеспечения под Linux, распространяющегося без исходных текстов, вызывает озабоченность сообщества Open Source, заинтересованного в создании открытых версий проприетарных продуктов, основная сложность клонирования которых состоит отнюдь не в кодировании, а в расшифровке протоколов обмена и форматов файлов. эту часть работы берут на себя хакеры, свободно владеющие отладчиком, разбирающиеся в дизассемблировании и знающие массу эффективных приемов реверсинга, о некоторых из которых мыщъх и собирается рассказать

введение

Противопоставляя Linux продукции Microsoft, многие почему-то забывают, что помимо операционных систем Microsoft выпускает и программное обеспечение прикладного типа (например, тот же MS Office) и если Linux станет необычайно популярной, то Microsoft (а вместе с ней и другие производители) начнут переносить свои продукты на эту ось, только исходных текстов нам, естественно, никто не предоставит. Сейчас мы имеем свободную операционную систему со свободным ПО с небольшой "примесью" закрытых программ (например, Intel C++, Skype, etc), но уже через несколько лет ситуация рискует превратиться в: псевдо-свободная операционная система с закрытыми драйверами, закрытым ПО и незначительной примесью открытых программ. Это дискредитируют саму идею Open Source и отрицательно влияет на безопасность, поскольку, закрытое ПО может содержать в себе что угодно — от непреднамеренных ошибок, до умышленных закладок.

Microsoft уже заключила договор с Novell о переносе MS Office на SuSE и это только начало. За Microsoft последуют и остальные. Чтобы предотвратить вторжение коммерческого ПО в плоскость свободной оси, необходимо создавать некоммерческие клоны всех закрытых программ, а для этого необходимо "распотрошить" двоичные файлы и "вытянуть" из них всю необходимую информацию о форматах файлах/протоколов, на основании которой кодеры создадут совместимый свободный продукт (он же "клон").

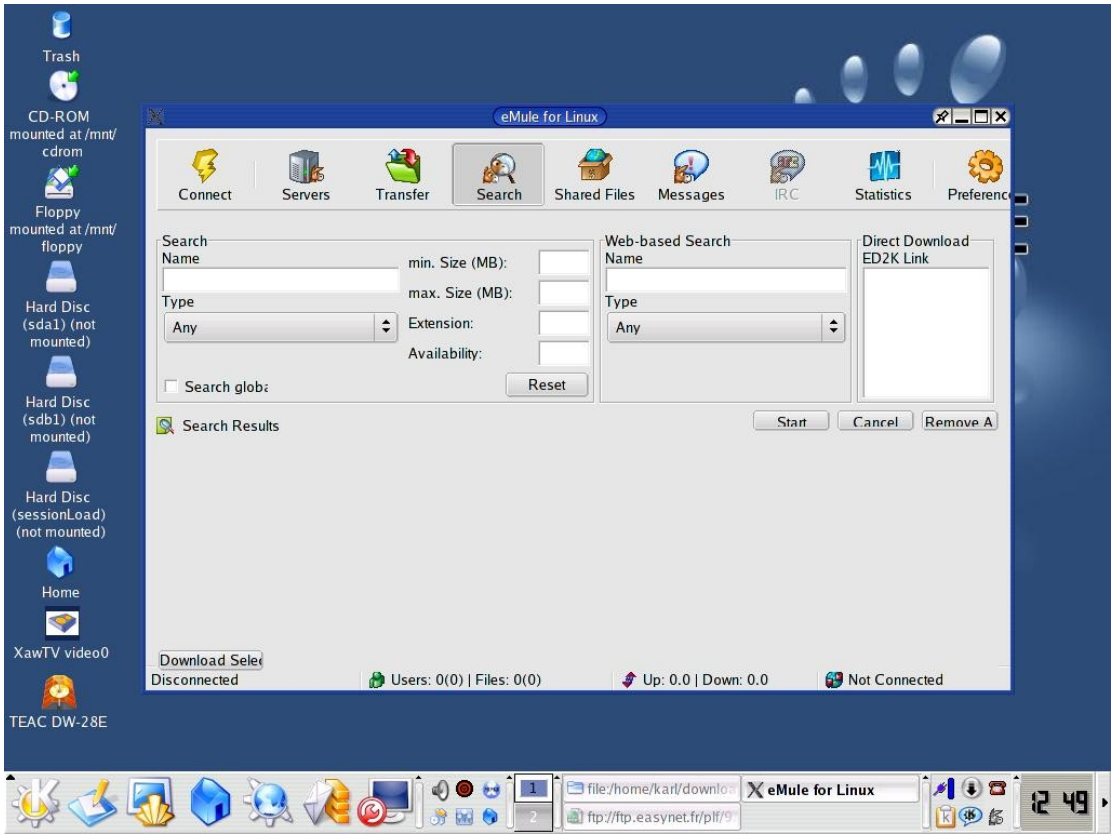

Рисунок 1 eMule – открытый клиент для сети eDonkey

чем мы будем действовать

Основной инструментом хакера является **дизассемблер**, лучшим из которых был и остается IDA Pro, поддерживающая ELF-формат, распознающая большое количество библиотечных функций и обладающая замечательной системой навигации по исследуемому файлу. Короче говоря, равных IDA Pro нет. Остальные дизассемблеры плетутся где-то в хвосте и годятся лишь для анализа небольших программ.

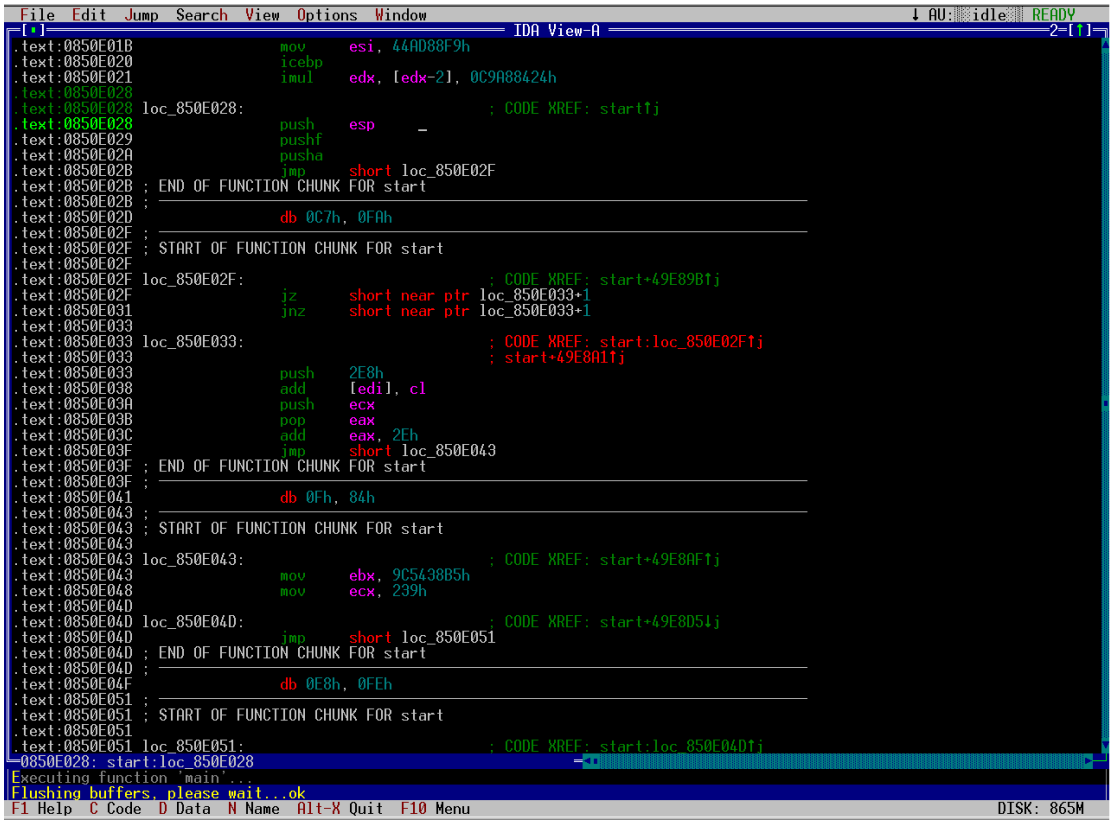

Рисунок 2 консольная версия IDA Pro под Linux

Техника дизассемблирования двоичных файлов под Linux мало чем отличается от анализа Windows-файлов, только вместо API-функций, мы будем иметь дело с функциями стандартных библиотек. Системные вызовы встречаются намного реже, т. к. они по разному реализованы в различных версиях Linux'a и производитель, заботящийся о совместимости, ни за что не будет к ним прибегать (если, конечно, он не наменян привязать свой продукт к конкретной оси). Упаковщиков исполняемых файлов под Linux практически нет, и большинство коммерческих программ распространяются в неупакованном виде, что значительно упрощает анализ, но "большинство" — еще не означает _все_ и тот же Skype активно использует многоуровневую динамическую шифровку кода, на снятие которой уходит огромное количество сигарет, пива и времени (подробнее о технике борьбы с упаковщиками можно прочитать в подборке статье http://nezumi.org.ru/unpack-pack.zip).

```
# gdb -q debug_demo
(no debugging symbols found)...(gdb) b main
Breakpoint 1 at 0x80484ca
(gdb) display/3i $pc
(gdb) r
Starting program: /root/debug_demo
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x80484ca in main ()
1: x/3i $eip
0x80484ca <main+6>:     movl   $0x1,0xfffffff8(%ebp)
0x80484d1 <main+13>:    movl   $0x0,0xfffffffc(%ebp)
0x80484d8 <main+20>:    cmpl   $0x5,0xfffffffc(%ebp)
(gdb) ni
0x80484d1 in main ()
1: x/3i $eip
0x80484d1 <main+13>:    movl   $0x0,0xfffffffc(%ebp)
0x80484d8 <main+20>:    cmpl   $0x5,0xfffffffc(%ebp)
0x80484dc <main+24>:    jle    0x80484e0 <main+28>
(gdb) ni
0x80484d8 in main ()
1: x/3i $eip
0x80484d8 <main+20>:    cmpl   $0x5,0xfffffffc(%ebp)
0x80484dc <main+24>:    jle    0x80484e0 <main+28>
0x80484de <main+26>:    jmp    0x80484f8 <main+52>
(gdb)
```

Рисунок 3 gdb – основной отладчик под Linux

Инструмент номер два — **отладчик**, в роли которого обычно выступает gdb, хотя в некоторых случаях удобнее пользоваться отладчиком, интегрированным в IDA Pro. И тот, и другой основаны на системной функции ptrace, с которой разработчики защит уже давно научились бороться, и против которой существует целый легион эффективных антии-отладочных приемов. Поэтому, в хакерском арсенале должны быть и другие отладчики, не использующие ptrace, как-то: ALD, linice, etc (желающих узнать поподробнее о linux-отладчиках и антиотладочных приемах мы отсылаем к подборке статей: http://nezumi.org.ru/linux-debug-pack.zip, а так же к мыщъх'иной книге "техника отладки программ без исходных текстов").

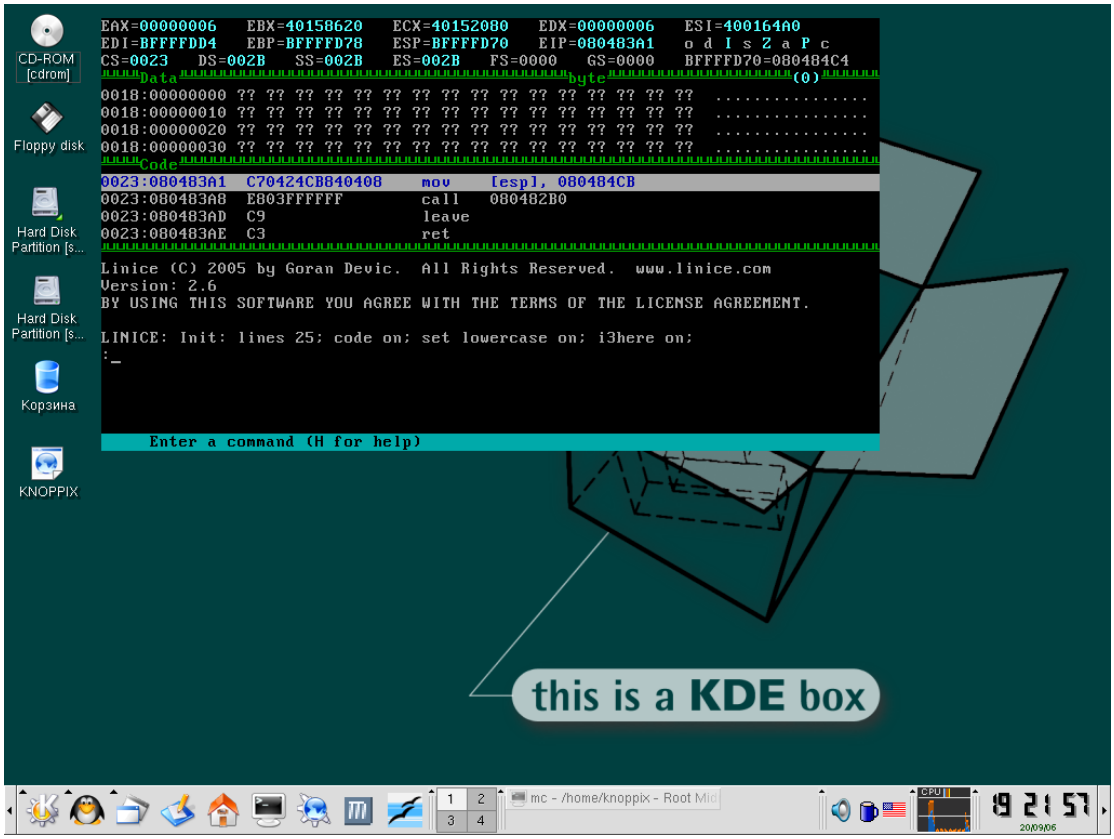

Рисунок 4 linice – альтернативный отладчик под Linux

С перехватом системных вызовов неплохо справляются штатные утилиты **truss** и **ktrace**, первая из которых работает в прикладном режиме, вторя — на уровне ядра. Для сбора сетевого трафика как правило используется **tcpdump**, входящий в комплект поставки большинства дистрибутивов.

Так же хакерам используется большое количество второстепенных утилит, простое перечисление которых заняло бы весь остаток статьи и потому они здесь не рассматриваются, т.к. нас интересуют общие методики анализа, а не пошаговая инструкция по взлому чего бы то ни было.

как мы будем действовать

Программы — они такие огромные, а жизнь — такая короткая, как мыщъхиный хвост. Если попытаться дизассемблировать приложение весом в несколько десятков мегабайт от начала и до конца, то не известно, что из этих двух закончится раньше. Да и смысла в дизассемблировании все подряд никакого нет. Интерфейсную часть и прочий тупой код можно переписать с нуля и без дизассемблера. Тоже самое относится ко многим стандартным алгоритмам типа быстрого дискретного преобразования Фурье.

Достаточно просто понять, что делает та или иная функция, а уж за конкретной реализацией дело не станет (существуют десятки алгоритмов Фурье-преобразования, "заточенные" под различные типы процессоров и создатели открытого клона, могут использовать более компактный/быстродействующий вариант без нарушения совместимости).

Вообще, можно выделить два уровня реконструкции программы: "микро" (алгоритмы работы отдельных функций) и "макро" (назначение функций и анализ связей между ними). Каждый из уровней имеет свою специфику, которая сейчас и будет рассмотрена.

реконструкция формата на микро-уровне

Техника отождествления функций требует определенной математической подготовки и программистского опыта. Знакомые функции зачастую распознаются с первого взгляда, незнакомые — не распознаются вообще. Можно сколько угодно ковыряться в дизассемблерном листинге, но этот ни на йоту не приблизит нас к разгадке, какую _цель_ преследуют все эти операции и какой _смысл_ несет преобразование массива чисел X в Y. В этом случае остается только одно — построчено переписать код функции с Ассемблера на Си (или на любой другой язык высокого уровня). При этом мы теряем шанс выбрать более эффективную реализацию и подходим к опасной черте, разделяющей независимую свободную реализацию от воровства двоичных модулей. И хотя, код, сгенерированный компилятором, будет значительно отличаться от дизассемблируемого кода, нас могут поймать на "водяных знаках" — dummy-коде, не несущем полезной нагрузки, но неизбежно сохраняющемся в открытом клоне при построчном переносе функции, что является достаточно убедительным доказательством плагиата. Обнаружить "водяные знаки" возможно только после отождествления алгоритма функции. Не стоит надеяться, что это будет тривиальный мусор в стиле MOV EAX,ECX/MOV ECX,EAX, скорее всего производитель использует избыточные преобразования данных, которые в _грубом_ приближении выглядят так: x=f(y)/1. Очевидно, что операция деления на единицу - лишняя, но чтобы ее исключить, требуется понять, что это именно деление, а не что-то другое.

Существует три пути отождествления функций: а) анализ алгоритма; б) сопоставление обрабатываемых данных с возвращенным результатом; в) уникальные константы. Ну, с анализом алгоритма, все понятно. Если мы _знаем_ тот или иной алгоритм (не важно чего — быстрой сортировки, обхода дерева, вычисления квадратного корня), то сможем отождествить его на любом языке — хоть на Паскале, хоть на Ассемблере. Проблема в том, что никакой отдельно взятый человек не в состоянии удержать в голове все алгоритмы (или хотя бы самые популярные из них), особенно если дело касается математики, поэтому, в хакерской команде должны быть узкие специалисты по криптографии, сжатию цифрового аудио/видео и т. д.

Для ускорения (и упрощения) анализа широко используется метод черного ящика — вместо того, чтобы изучать код функции, мы перехватываем передаваемые ей данные и смотрим на возвращаемый результат. Сопоставляя первое со вторым, пытаемся постичь суть. Алгоритмы поиска или сортировки распознаются сразу же. С упаковкой/распаковкой дела обстоят несколько сложнее. Сам факт упаковки/распаковки обнаруживается с первого взгляда, но вот _алгоритм_ упаковки/распаковки остается загадкой. Если только функция не работает с общепринятыми форматами типа gzip, то для создания совместимого клона, необходимо полностью реконструировать ее алгоритм.

К счастью, многие алгоритмы оперируют уникальными константами: стандартные полиномы позволяют отождествить методику шифрования/расчета контрольной суммы буквально за несколько секунд, а рефересные матрицы квантования легко разоблачают аудио/видео кодеки, причем, операция отождествления легко поддается автоматизации. В частности, для IDA Pro существует несколько хороших плагинов (которые можно найти на www.idapro.com), распознающих большое количество алгоритмов шифрования.

С реконструкцией форматов файлов и протоколов обмена все обстоит и проще и сложнее. Проще — потому, что реконструкция базовой структуры формата не требует никаких специальных знаний и вполне довольствуется минимальными навыками владения отладчиком/дизассемблером. Сложнее — потому что на низком уровне практически всякий формат включает в себя алгоритмы шифрования, подсчета контрольной суммы, сжатия (с потерями или без) и т. д. и т. п. То есть, если написать свой "парсер" закрытого формата сможет и начинающий хакер, то написание совместимого открытого клиента потребует усилий целой команды хакеров, впрочем, это не обязательно должна быть именно _команда_ и реконструкцией формата на "микро" и "макро" уровне может заниматься толпа никем не координируемых посторонних людей.

реконструкция формата на макро-уровне

Запустив tcpdump (или любой другой сниффер по вкусу) мы сможем перехватывать сообщения, которыми обмениваются клиент с сервером, наблюдая как происходит процедура регистрации/авторизации, передачи данных и т. д. Аналогичным образом обстоят дела и с реконструкцией форматов файлов. Перехват системных вызовов открытия, позиционирования, чтения и записи в файл несет в себе огромное количество информации и разбивает монолитную структуру файла на составляющие его "кирпичики", однако, внутренняя структура кирпичиков, на данном этапе остается загадкой.

Допустим, программа читает первые N байт от начала файла, затем прыгает по смещению X, считывает еще K байт, прыгает по смещению Y… Что это может означать? Если формат не зашифрован/упакован, то скорее всего где-то в заголовке содержится ссылка на индексы (смещение позиции X, задаваемое как правило либо от начала файла, либо от конца заголовка), где лежат указатели на подчиненные структуры данных, одна из которых и расположена по смещению Y, где может находится (например) текст, подготовленный к выводу на экран.

Для реконструкции неупакованного/незашифрованного файла вполне достаточно утилит truss/ktrace и hex-редактора. Все индексы и данные там будут лежать в открытом виде "как есть" и располагая одной лишь последовательностью вызовов seek/read несложно разобраться где какое поле лежит и чему принадлежит, однако, "открытые" форматы встречаются достаточно редко, поэтому к hex-редактору добавляется дизассемблер и отладчик.

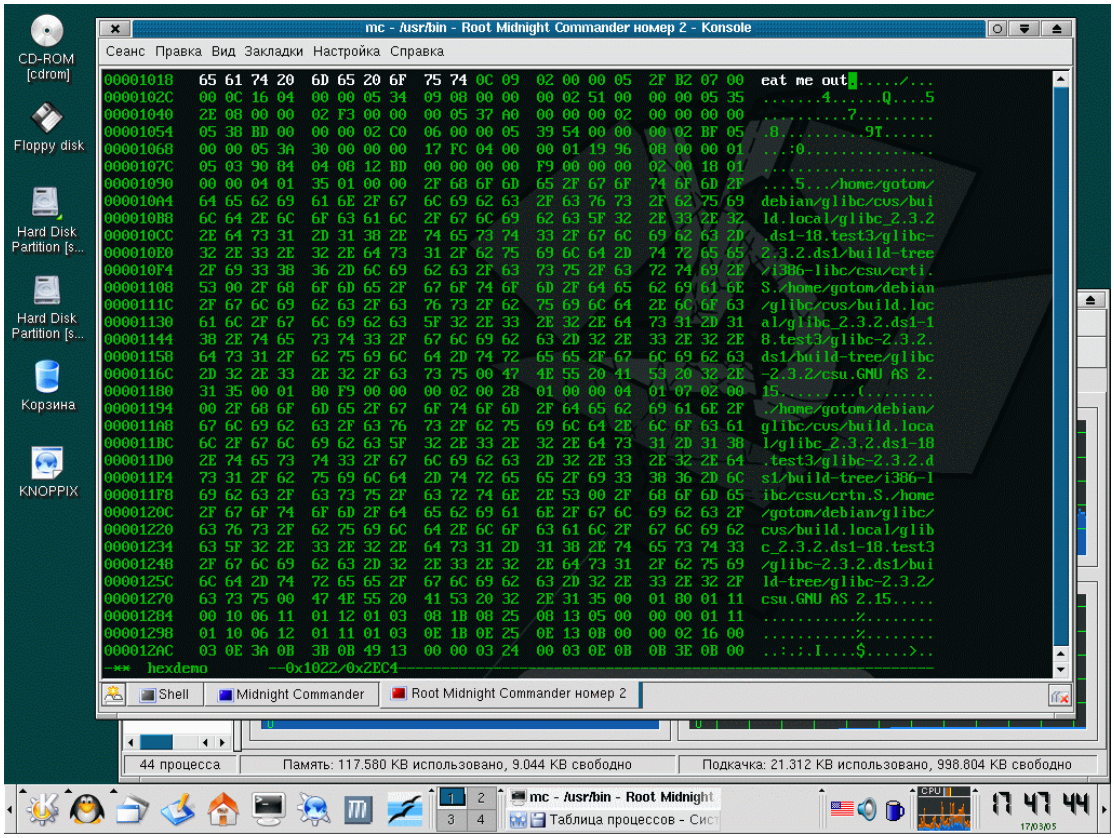

Рисунок 5 hexedit – простой hex-редактор под Linux

В отладчике мы устанавливаем точки останова на все функции ввода/вывода (как файловые, так и сетевые) и ведем тот же протокол, который создают truss/ktrace с той лишь разницей, что мы ставим точки останова на блоки памяти, возвращенные функцией read. При первом же обращении к ним, отладчик "всплывает", позволяя нам "запеленговать" функцию обработчик. На первом этапе мы лишь записываем адреса функций-обработчиков и наблюдаем за возвращаемыми ими данными, устанавливая на них дополнительные точки останова и пытаясь понять, чем именно данная функция занимается (в частности, функции-распаковщики распознаются практически сразу).

Это довольно кропотливая работа, требующая большой усидчивости и хорошей памяти (не оперативной), позволяющей удержать в голове множество последовательностей вызова вложенных друг в друга функций, в которых поначалу не видно никакой структуры, но по мере продолжения исследований между функциями образуются "прочные" мосты, транспортирующие данные вполне предсказуемыми маршрутами.

Основная проблема в том, что аппаратных точек останова на x86 всего четыре и максимальная длина каждой из них составляет двойное слово. Аппаратных точек останова катастрофически не хватает, а невозможность установить точку останова на регион памяти вгоняет хакеров в глубокую тоску, граничащую с суицидом. Можно, конечно, попробовать установить точку останова на первый байт данных, возвращенных функцией read, надеясь на то, что разбор данных происходит с самого начала. В большинстве случаев именно так все и происходит, однако, никаких гарантий в этом у нас нет. К тому же, некоторые программы сразу считывают весь файл (или все служебные структуры файла) в память и в дальнейшем работают с ним напрямую, не обращаясь к seek.

При исчерпании аппаратных точек останова отладчик gdb предлагает установить программную (количество которых не ограничено), однако, при этом он переходит в режим пошагового исполнения программы, проверяя каждую машинную инструкцию, в результате чего скорость выполнения программы катастрофически замедляется. Для анализа дисковых файлов это, в общем-то некритично, а вот при реконструкции протокола обмена, время прогона очередного куска под отладчиком может превысить время ожидания сервера и нас вышибут по time-out'у, вынуждая приобретать более быстрый процессор или…

…эмулировать аппаратные точки останова путем выставления атрибутов страниц в PROT_NONE с помощью функции mprotetc, которому можно вызывать непосредственно из-под

gdb (который, в отличии от soft-ice позволяет вызывать любые функции). При обращении к странице памяти, отладчик всплывает по исключению типа нарушения доступа и нам остается только записать адрес "дерзнувшей" инструкции, вернуть атрибуты на место, выполнить инструкцию (или даже всю функцию целиком) и вызвать mprotect еще раз, забирая атрибуты обратно (при этом отладчик, естественно, должен быть настроен так, чтобы "поглощать" сигнал о нарушении доступа, не передавая его прикладной программе, о том как это сделать рассказано в статье, входящей в linux-debug-pack.zip).

Этот прием не накладывает никаких ограничений на количество точек останова, требуя лишь чтобы их размер и базовый адрес был кратен длине одной странице памяти. Это достаточно жесткое требование. А что делать, если нам требуется установить точку останова на 300h байт памяти по адресу 8048123h? Очень просто! Ставим точку останова на страницу 8048000h, а затем "вручную" отбрасываем ложные срабатывания, находящиеся вне адресов 8048123h - 8048423h (если же регион памяти, на который необходимо установить точку останова, пересекает границу страниц памяти, приходится отнимать атрибуты доступа сразу у двух страниц). При желании, эту работу легко автоматизировать, поскольку gdb поддерживает довольно развитую систему макросов, на которых можно написать практически все, что угодно.

Как вариант, можно использовать BOCHS – популярный эмулятор PC, в исходные тексты которого входит неплохой отладчик в стиле Turbo-Debugger, правда, работающий только в Windows-версии BOCHS'a, однако, на самом BOCHS'e может быть установлено все, что угодно (Linux, например).

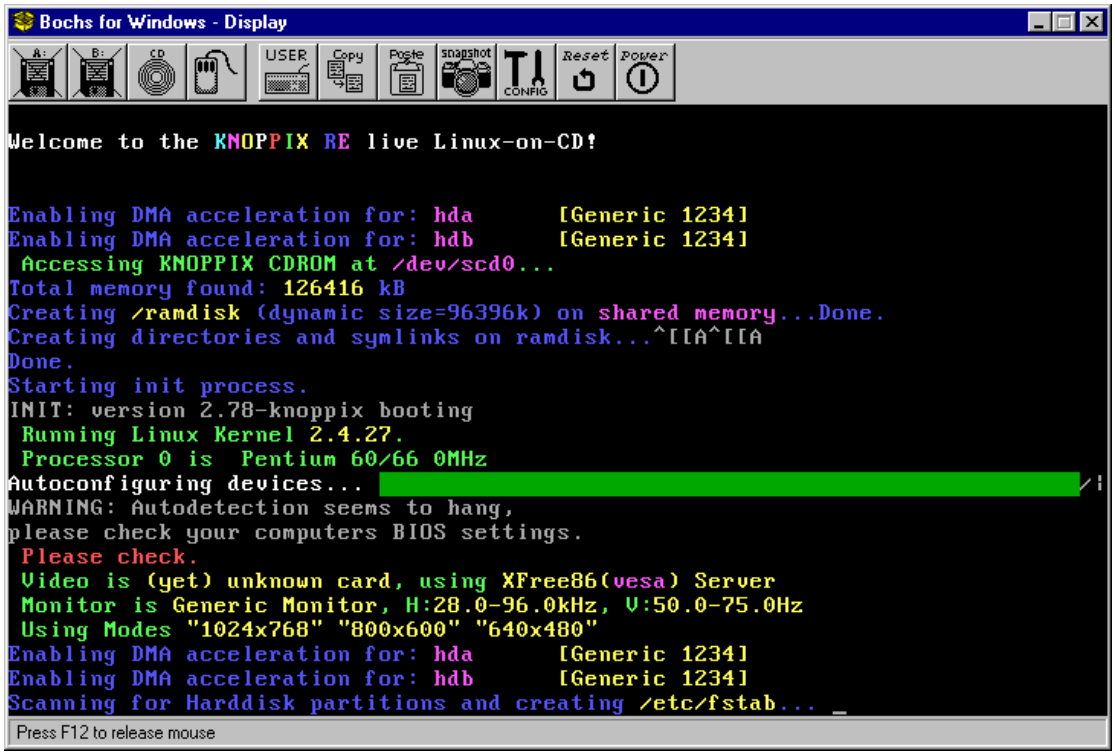

Рисунок 6 использование эмулятора PC для отладки программ

Таким образом, в нашем распоряжении окажется список адресов, откуда происходит вызов функций read/seek, а так же список адресов, где осуществляется обработка данных. Исследуя окрестности этих адресов в дизассемблере, мы сможем реконструировать протокол обмена/формат файла за достаточно продолжительное, но все-таки конечное время, причем, эту фазу работы легко распараллелить между несколькими участниками — ведь на макро-уровне связи между функциями уже ясны и теперь пришла пора рыть в глубь, анализируя алгоритм работы каждой функции в отдельности.

вертикальный лимит пределов и ограничений

Полная реконструкция формата файла/протокола обмена на основе наблюдений за "живым" обменом с помощью truss/ktrace/tcpdump невозможна в принципе, поскольку далеко не в каждом файле (сеансе обмена с сервером) задействуются все поля/команды. Открытый клон,

созданный на основе таких данных, будет падать при открытии каждого N'го файла или при впадать в ступор при получении от сервера "непонятного" сообщения. Ну, и куда такое годится?! Пользователи тут же взвоют и расстанутся с открытым клоном в пользу оригинального продукта, даже если клон дешевле/удобнее и т. д.

Поэтому, необходимо пройтись по всему двоичному коду исследуемой программы, отыскивая по перекрестным ссылкам функции-обработчики, которые не вызывались в текущем сеансе, но все-таки присутствуют и обрабатывают определенные поля файла/команды протокола, часть из которых вообще никогда не встречается (устарели или не реализованы на серверной стороне), а часть относится к экзотическим породам, встречающихся только в определенных файлах (например, "японской" версии, поддерживающей особенности их алфавита). Дизассемблировать такие функции, не имя образца файла, на котором их можно протестировать, это все равно что стрелять в темноте наугад, но… кто говорил, что быть хакером легко?! Вот почему многие форматы/протоколы обмена реконструируются годами, а протокол обмена Skype до сих пор реконструирован только в самых общих чертах, несмотря на то, что его "грызут" многие хакерские коллективы, пытающиеся использовать Skype-сеть для распространения червей, дронов и прочей живности, но… далеко им продвинуться так и не удалось.

Самое неприятное, что назначение некоторых незадействованных в данной версии программы функций, _невозможно_ выяснить путем отладки/дизассемблирования, поскольку их алгоритмы могут быть завязаны на обрабатываемые данные, образцов которых в нашем распоряжении нет и не будет пока производитель не выпустит новую версию, полностью совместимую со старой, чего нельзя сказать про наш свободный клон (если, конечно, не прибегнуть к построчному переносу всех незадействованных функций, но даже в этом случае мы не сможем их отладить, а переписать несколько тысяч строк кода без ошибок — крайне маловероятно, даже при самой тщательной проверке).

заключение

Свободные клоны коммерческих программ с закрытым протоколом обмена/форматом файлов заслужили репутацию нестабильных (OpenOffice открывает далеко не все документы, созданные MS Office и это всего лишь один пример), однако, ими пользуется огромное количество людей по всему миру, выявляя все новые ошибки и несовместимости, которые позднее исправляют разработчики, в результате чего качество свободного клоны неуклонно растет, но… увы… производители закрытых программ не сидят сложа руки и выпускают новые версии, поэтому открытые клоны обречены на пожизненное отставание от прогресса, а хакеры вынуждены расходовать огромное количество времени на исследование, но… другого выхода нет. Если рынок уже захвачен несвободным продуктом, то писать что-то свое, пусть даже в десять раз лучшее, в отсутствии совместимости с уже имеющимся — бесполезно.