защита игр от взлома

крис касперски ака мыщъх

> **игру мало написать! еще ее необходимо защитить так, чтобы не взломали (или взломали, но не сразу), причем защита должна быть максимально простой и предельно безглючной. на эту тему написано множество статей (от обзорно-познавательных, до углубленных в частную проблему), но очень мало "рецептурных" справочников, доходчиво объясняющих куда идти и что делать без лишней воды**

введение

Обычно о защите вспоминают в последний момент перед самой сдачей проекта и дописывают несколько десятков строк кода впопыхах, а потому удивляются, почему взломанные версии доходят до рынка раньше официальных?! Защита должна быть глубоко интернирована в программу и разрабатываться заранее. Это скажет любой эксперт. Но! Тогда к багам самой программы добавятся глюки защиты и проект рискует не уложиться в срок, кроме того не ясно как отлаживать программу, если она доверху нашпигована антиотладочными приемами и активно сопротивляется отладчику.

Защитный механизм лучше всего реализовать в виде модулей, готовых к интеграции в программу в любой момент. Рабочая версия вместо защитных функций вызывает процедуры-пустышки, заменяемые в финальной версии реальным кодом. А вот о том, как проектировать эти модули мы сейчас и поговорим.

арсенал защиты

Вот три кита, на которых держатся защитные механизмы: ***машинный код***, ***шифровка*** и ***p-код.*** Электронные ключи и прочие экзотические технологии этого типа здесь не рассматриваются. Массовая продукция именитых фирм давно взломана, а разработка своего собственного ключа с нуля — занятие не для слабонервных, да и выгоден он будет только при серийном производстве, иначе защита просто не окупиться.

Аппаратные защиты — это вообще другой разговор. При желании в микрочип можно перенести хоть всю программу целиком и тогда никто не сможет скопировать (если, конечно, выбрать правильный чип), но... процесс разработки и отладчики усложняется в десятки и даже сотни раз, "железное обеспечение" получается крайне негибким, неудобным в тиражировании и распространении, не говоря уже о невозможности исправления обнаруженных ошибок. Даже если чип имеет перепрошиваемое ПЗУ, разрешение на запись равносильно разрешению на чтение. Почему? Да потому что самое ценное в железе — это прошивка. Скопировать железо намного проще, чем выдрать из защищенного чипа прошивку, а если же она будет свободно распространяется как обновление… Хорошо, будем распространять не всю прошивку, а только измененные модули или даже их части. Что сделает хакер? Он просто создаст слегка модифицированный модуль, считывающий содержимое ПЗУ и выводящий его наружу контрабандным путем.

Защиты, о которых мы будем говорить, не требуют никакого дополнительного железа, они системно независимы, работают на прикладном уровне и не требуют прав администратора. Никаких драйверов! Никакого ассемблера! Возможно ли эффективно противостоять опытным хакерам в таких условиях? Поверьте мне, парни, это возможно! Большинство защит ломаются из-за досадных ошибок и мелких конструктивных просчетов, допущенных потому, что разработчик никогда не заглядывал в дизассемблерный листинг и не пытался взломать свое творение. За время своей жизнедеятельности, мыщъх собрал целую коллекцию таких ошибок и теперь делиться ей с народом, давая советы по усилению защитных механизмов.

тайники машинного кода

Сокрытие алгоритма в машинном коде — широко распространенный, но неэффективный защитный примем. Дизассемблерный листинг это, конечно, не исходный код. Одна строка на языке высокого уровня может транслироваться в десятки машинных команд, зачастую перемешанных компилятор с командными соседних строк для достижения наивысшей скорости выполнения. В двоичном файле нет ни комментариев, ни структуры классов, ни имен

функций/переменных, ни… Стоп! Комментариев там действительно нет (да они и не нужны), а вот все остальное очень даже встречается!

текстовые строки

ASCII и UNICODE строки несут очень богатую информацию. Это и текстовые сообщения, выводимые на экран при регистрации/окончании триального срока/неправильном вводе пароля, и ветви реестра, и имена ключевых файлов, а иногда и сами серийные номера/пароли. Ну с паролями все ясно. Хранить их в открытом виде нельзя и надо хэшировать, а что плохого в текстовых сообщениях? А то, что обнаружив их в теле программы хакер по перекрестным ссылкам очень быстро найдет тот код, который их выводит, вот например (см. листинг 1). Тоже самое относится и к именам ключевых файлов с ветвями реестра.

```
.text:00401016 call    sub_401000
.text:0040101B add     esp, 4
.text:0040101E test    eax, eax
.text:00401020 jz      short loc_40102F
.text:00401022 push    offset aWrongSN ; "wrong s/n\n"       ; указатель на строку
.text:00401027 call    _printf
...
.data:00406030 aWrongSN       db 'wrong s/n',0Ah,0           ; DATA XREF: 00401022h↑o
```

Листинг 1 текстовая строка "wrong s/n" с перекрестной ссылкой ведущей к процедуре sub_401000, которая, очевидно, и является проверкой серийного номера

Чтобы затруднить анализ, все строки необходимо либо зашифровать, расшифровывая только перед непосредственным употреблением (если расшифровать сразу же после запуска программы, хакер просто снимет дамп и увидит их прямым текстом), либо поместить их в ресурсы и грузить через LoadString — это легче программируется, но и легче ломается: хакеру достаточно открыть файл в любом редакторе ресурсов, найти нужную строку, запомнить ее идентификатор, запустить отладчик и установить условную точку останова, дождаться вызова LoadString с нужным uID, определить адрес буфера lpBuffer, установить на него точку останова и... если lpBuffer выводится не сразу, а передается через цепочку промежуточных буферов, хакер материться, но ничего не может сделать. Аппаратных точек останова всего четыре и если защита использует не последний буфер в цепочке, то отследить момент реального обращения к строке становится невозможно (на самом деле возможно — путем применения секретного хакерского оружие NO_ACCESS на страницу, только об этом не все знают).

В дизассемблере же отследить перемещение строки по локальным буферам практические невозможно (во всяком случае автоматически). Глобальные же буфера (в которые компилятор пихает константные строки, в том числе и зашифрованные) отслеживаются сразу, так что LoadString по некоторым позициям очень даже рулит!

символьная информация

Отладочная информация. По умолчанию компилятор генерирует файл без отладочной информации и туда она попадет только в исключительных случаях, но все-таки попадает. Причем не только у начинающих программистов, не знающих чем отличается Debug от Release, но и "маститых" фирм, выпускающих достойные программные продукты. Допустим, у нас имеется глобальная программа IsRegistered, тогда смысл следующей пары машинных команд будет ясен и без комментариев (см. листинг 2). ***Никогда не оставляйте отладочную информацию в откомпилированной программе!***

```
.text:00405664        cmp     _IsRegistered, 0
.text:0040566B        jz      short loc_40567A
```

Листинг 2 дизассемблерный листинг исполняемого файла с отладочной информацией — осмысленные имена переменных и функций необычно упрощают взлорм

Динамические библиотеки. Имена неэкспортируемых функций уничтожаются компилятором, экспортируемые же по умолчанию остаются "как есть". Си++ компиляторы в дополнении к этому "замангляют" имена, дописывая к нем "зашифрованный" прототип функции, но IDA PRO с легкостью возвращает их в исходный вид. Если защитный модуль реализуется в виде динамической библиотеки (как очень часто и бывает), наличие символьных имен (да еще с готовыми прототипами!) значительно упрощает анализ. В частности, OO Software (создатель одноименного дефрагментатора) любит таскать за собой библиотеку

oorwiz.dll, что очевидно расшифровывается как "OO Registration Wizard", экспортирующую всего три функции, но зато каких (см. листинг 3). ***Всегда удаляйте все символьные имена из экспорта и вызывайте функции только по ординалу.***

```
3       0 00001FD0 RegWiz_InitLicMgr
1       1 000019D0 RegWiz_InitReadOnly
2       2 00001D00 RegWiz_InitTrial
```

Листинг 3 библиотека oorwiz.dll от OO Software экспортирует функции, говорящие за себя

RTTI. Динамические классы, тесно связанные с механизмом RTTI (Runtime Type Identification) и активно используемые компиляторами DELPHI/Borland C++ Builder, сохраняют в откомпилированном файле не только свою структуру, но и символьные имена! Вот результат работы утилиты DEDE, реконструировавшей структуру классов программы Etlin HTTP Proxy Server. В глаза сразу бросается класс TfrmRegister, соответствующей форме fRegister, обрабатывающий нажатие кнопки "OK" процедурой bOKClick, расположенной по адресу 48D2DCh. Благодаря динамическим классам сердце защитного механизма было локализовано всего за несколько секунд! ***Не используйте RTTI в защитных механизмах или по крайней мере не давайте формам и обработчикам осмысленные имена!***

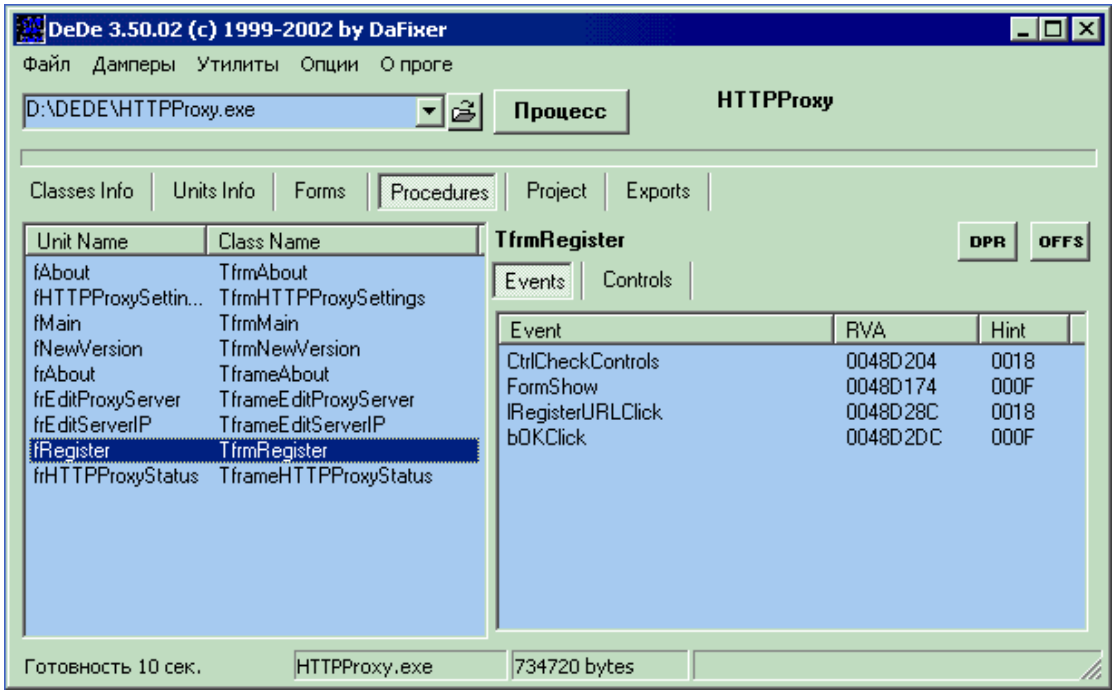

Рисунок 1 утилита DEDE восстанавливает RTTI информацию из программы Etlin HTTP Proxy, видны надписи: fRegister, CtrlCheckControls, FormShow, IRegisterURLClick, bOKClick и другие

обфускация

Код, генерируемый компилятором, очень громоздок и разобраться в нем крайне непросто, однако, все-таки возможно. Чтобы этому помешать, некоторые протекторы используют "запутывание" или обфускацию (от английского obfuscation). В простейшем случае автоматический кодогенератор, свинченный с полиморфного движка, внедряет в код огромное количество незначащих команд типа NOP, XCHG EAX,EBX/XHG EBX,EAX, нашпиговывая ими программу как рождественскую утку или гуся. Более совершенные генераторы используют разветвленную систему условных переходов, математические операции и присвоения, результат которых никак не используется, и другие полиморфные технологии.

```
.00434000: 60                pushad
.00434001: E800000000        call    .000434006   -------- (1)
.00434006: 5D                pop     ebp
.00434007: 50                push    eax
.00434008: 51                push    ecx
```

```
.00434009: EB0F              jmps    .00043401A   -------- (2)
.0043400B: B9EB0FB8EB        mov     ecx,0EBB80FEB ;"e?☼e"
.00434010: 07                pop     es
.00434011: B9EB0F90EB        mov     ecx,0EB900FEB ;"e?☼e"
.00434016: 08FD              or      ch,bh
.00434018: EB0B              jmps    .000434025   -------- (3)
.0043401A: F2                repne
.0043401B: EBF5              jmps    .000434012   -------- (4)
.0043401D: EBF6              jmps    .000434015   -------- (5)
.0043401F: F2                repne
.00434020: EB08              jmps    .00043402A   -------- (6)
.00434022: FD                std
.00434023: EBE9              jmps    .00043400E   -------- (7)
.00434025: F3                repe
.00434026: EBE4              jmps    .00043400C   -------- (8)
.00434028: FC                cld
.00434029: E959585051        jmp     051533887
.0043402E: EB0F              jmps    .00043403F   -------- (9)
```

Листинг 4 фрагмент программы, защищенной протектором armadillo

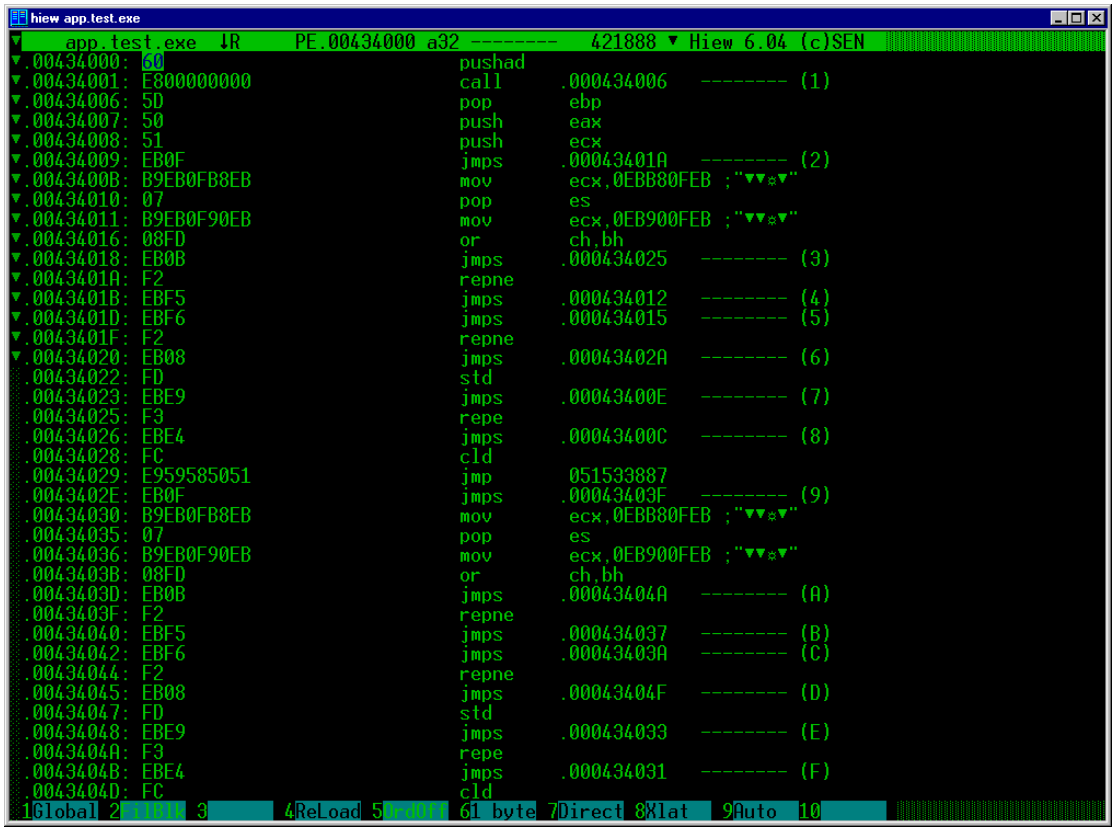

Рисунок 2 фрагмент программы, защищенной протектором armadillo

Сгенерировать можно хоть миллион команд. Это легко. А вот проанализировать их намного сложнее, если вообще возможно. Назначающие команды и заведомо никогда не исполняющиеся условные переходы типа (XOR EAX,EAX/JNZ trg) сможет отсеять и компьютер (достаточно написать простенький плагин к дизассемблеру IDA PRO), освободиться же от ненужных вычислений значительно сложнее. Как минимум необходимо загнать все команды на граф, отображающий зависимости по данным, и убрать замыкающиеся ветви. Некоторые хакерские команды уже решили эту задачу (например, группа Володи с wasm'a), однако, в публичном доступе готовых инструментов что-то не наблюдается, а это значит, что юные взломщики, наткнувшись на обфускаторный код скорее обломаются, чем его взломают и это (с точки зрения разработчика программы), будет очень хорошо!

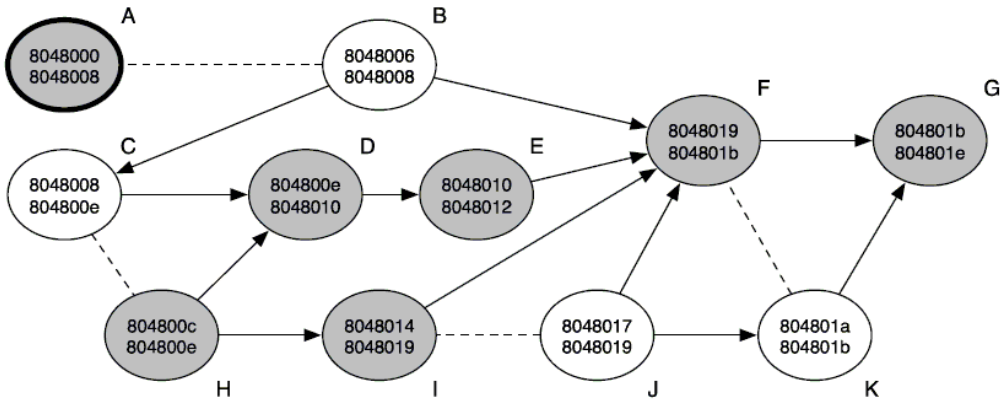

Рисунок 3 визуализация программы на графе

Высаживаться на разработку собственного обфускатора совершенно необязательно. Есть и готовые. Как коммерческие, так и бесплатные. Например, .NET Obfuscator (http://blogs.msdn.com/obfuscator/default.aspx). Забавно, но большинство обфускаторов не используют обфускацию для защиты самих себя от взлома! А все потому, что в программах, критичных к производительности (какими, в частности, являются трехмерные игры), обфускация вызывает значительные тормоза и запутывать можно только редко вызываемые модули, например, код защитного механизма, но при этом возникает угроза, что хакер просто "выломает" защитный механизм из программы без анализа его устройства. В большинстве случаев для этого достаточно проанализировать код материнской процедуры (которая не подвергалась обсуфскации) и удалить вызов "запутанной" защитной функции, подсунув "правильный" код возврата, который ожидает вызывающая функция. Чтобы этого не произошло, защитная процедура помимо проверки аутентичности копии программы должна делать что-то полезное, такое, без чего программа работать не сможет. Но и в этом случае у хакера остаются хорошие шансы на взлом — шпионаж за API-функциями и реестром дает богатую пищу для размышлений, зачатую избавляющую от необходимости анализа машинного кода. ***Так что обфускация — не панацея и слепое использование готовых обфускаторов всего лишь увеличивает объем защищаемой программы и ухудшает производительности, но далеко не всегда затрудняет взлом!***

шифровка

Шифровка — это мощное оружие против взлома, бьющее точно в цель и высаживающее хакера на конкретный геморрой. Она бывает двух видов: *статическая* и *динамическая*. При статической шифровке зашифрованный код/данные расшифровываются один единственный раз на самой ранней стадии инициализации программы, после чего расшифрованному коду передается управление. Это просто программируется и еще проще ломается — дождавшись завершения расшифровки, хакер снимает с программы дамп и дизассемблирует его.

Динамические шифровщики расшифровывают код по мере возникновения в нем необходимости и после возвращения управления тут же зашифровывают его вновь. Чем меньшие порции кода (данных) расшифровываются за раз, тем лучше (если извернуться, можно расшифровывать по одной машинной команде или байту данных). Таким образом, при динамической шифровке, в каждый момент времени в памяти присутствуют только крохотные куски расшифрованного кода/данных и снятие дампа дает немого пользы.

Широкому внедрению динамической шифровке препятствуют следующие проблемы: во-первых: сложность разработки и трудоемкость отладки, во-вторых: производительность, точнее полное отсутствие таковой (а в играх это очень критично) и, в-третьих, потенциальная возможность снять дамп руками самого расшифровщика, которому последовательно передаются адреса всех зашифрованных регионов или легким битхаком его тело исправляется так, чтобы он только расшифровывал, но ничего не зашифровывал. Конечно, этому можно противостоять, например, использовав перекрывающие шифроблоки, которые могут быть расшифрованы только по очереди, но не все сразу, или многослойную шифровку типа "луковицы", когда одни шифровщик плюс немного полезного кода вложен в другой, а тот в третий и т.д. Шифровщики как бы перемешаны с кодом и "отодрать" чистый дамп невозможно.

Это очень надежно, но очень, очень сложно реализуемо. ***Бесспорных лидеров среди алгоритмов шифровки нет! Выбор предпочтительного метода защиты не столь однозначен и статическая шифровка при всех своих недостатках продолжает удерживать прочные позиции и сдавать их не собирается.***

p-код

Термин p-код восходит к интерпретатору Visual Basic'a, однако, в хакерских кругах означает нечто большее и подразумевает любой интерпретируемый код произвольной виртуальной машины (не путать с VMWare). Непосредственное дизассемблирование в этом случае становится невозможно и приходится прибегать к декомпиляции, а для этого необходимо проанализировать алгоритм работы интерпретатора, написать (и отладить!) декомпилятор, что требует пива и времени. Правда, разработка (и отладка!) интерпретатора тоже даром не обходится и нам ним приходится попыхтеть (Джа в помощь!). К тому же разработка языка тянет за собой кучу вспомогательного инструментария (в первую нам понадобиться отладчик), иначе как на нем программировать?!

Все это выливается в солидный проект, который может быть использован всего один раз, в одной-единственной программе, да и то после выхода нескольких версий ядро интерпретатора желательно слегка изменять, чтобы написанный хакером декомпилятор перестал работать. Что поделаешь! Защита требует жертв и больших вложений. Ладно бы только вложений! Производительность интерпретируемого кода плетется в самом хвосте, отставая от динамической расшифровки и обфускации, но... может быть, есть возможность реализовать на p-коде только защитные модули? Нет! Тогда их отломают не глядя! На p-коде должна быть реализована вся программа целиком, в том числе и защитный механизм, тогда без декомпилятора его будет не хакнуть. Но это — в теории. На практике же, полностью загнать программу в p-код не удается по причине производительности и критичные к быстродействую функции пишутся на языке высокого уровня или даже ассемблере. А вот вызываются они уже из p-кода, в котором сосредоточена основная логика по типу "если нельзя, то все-таки".

Кстати говоря, большинство игровых миров управляются своим собственным скрпитовым языком, описывающим движение облаков, поведение мячика и характер разных монстров. На чистом Си никакую игру сложнее "тетриса" не запрограммировать! А раз мы уже имеем скриптовый язык, то почему бы не включить в него несколько защитных функций? Тогда чтобы взломать программу, хакеру придется разобраться в работе скрипиового движка.

Чаще всего разработчики используют Паскаль- или Бейсик-подобные языки, что не самый лучший выбор с точки зрения защищенности. Программа на p-коде при этом представляет собой последовательность ключевых слов (операторов языка) с аргументами и очень просто декомпилируется. На другом конце шкалы сложности находится Машина Тьюринга, Сети Петри, Стрелка Пирса и прочие примитивные виртуальные машины, реализующие фундаментальные логические операции, в результате чего даже такая конструкция как "IF THEN ELSE" распадается на сотни микрокоманд! Солнце погаснет прежде, чем хакер их проанализирует или… все-таки не погаснет? Существует множество продвинутых способов наглядной визуализации таких алгоритмов и к тому же… мы совсем забыли о производительности! Реализовать crackme на базе Машины Тьюринга еще можно, а вот коммерческое приложение — едва ли.

Хорошая идея — приложить к этому делу Форт. Вот его преимущества: простота реализации Форт-машины, компактность p-кода, довольно высокая производительность, сложность декомпиляции и ни на что не похожесть. Форт стоит особняком от всех языков, совсем не стремясь соответствовать человеческим привычкам, "здравому смыслу" и "логике". Разобраться в его идеологии с нуля — будет непросто. Хакеру придется нарыть кучу литературы и запастись терпением. Даже если ему не наскучит, разработчик защиты получит хорошую фору по времени, ну а там… мыщъх что-нибудь придумает!

Ниже приведен код Форт-машины, выдранный из программы see.exe, поставляемой вместе с Interrupt List'ом Ральфа Брауна.

```
seg000:1F29 loc_1F29:                               ; CODE XREF: start+39B9↓j
seg000:1F29     call    sub_1F04
seg000:1F2C     mov     word_A5C, bx
seg000:1F30     mov     si, dx
seg000:1F32     mov     bp, [bx+38h]
seg000:1F35     mov     sp, [bx+36h]
seg000:1F38     lodsw
seg000:1F39     mov     bx, ax
```

```
seg000:1F3B    jmp     word ptr [bx]
```

Листинг 5 дизассемблерный листинг Форт-машины

Другая хорошая идея — использовать готовые интерпретаторы малораспространенных языков (Пролог, Хаскель, Оберон) под которые еще не написаны декомпиляторы. Многие из них разработаны в недрах исследовательских институтов и поставляются в исходных текстах, что упрощает модификацию ядра интерпретатора и позволяет легко адоптировать чужой язык к нашим целям.

Круче всего — нарыть в сети эмулятор малоизвестной ЭВМ, под которую есть компилятор с языка высокого уровня (Си, ФОРТАН) и... хакеру придется очень худо. Придется осваивать неизвестный ему машинный язык, искать дизассемблер (или, что более вероятно, писать свой собственный), прилагая титанические усилия для взлома, а разработчик будет пить пиво и кодить в привычной для него среде. Самое замечательное, что в следующей версии программы можно использовать эмулятор от другой ЭВМ, перекомпилировав весь код без адоптации и каких-либо изменений. Ну совсем уж без изменений, конечно, не получится, но трудоемкость переноса не сравнить с тем объемом работы, что предстоит проделать взломщику!

секреты привязки

Рассуждая о методах защиты, мы не касались вопроса привязки, то есть такой характеристики, которая бы отличала одну копию программы от другой. Самые надежные защиты основаны на привязке к носителю. Они же самые капризные и глюкавые. Мыщъх подробно исследовал лазерные диски (даже книжку "техника защиты лазерных дисков" написал!) и пришел к выводу, что создать защиту, которая безошибочно бы распознавалась любым приводом (включая что-то сильно раздолбанное или слишком нестандартное), но в то же время не копировалась ни одним известным копировщиком — практически нереально. Наилучший результат дает привязка к геометрии спиральной дорожки, (защиты типа CD-COPS, Star-Force), ее невозможно скопировать, но легко проэмулировать или… подобрать диск с похожими характеристиками. Для предотвращения эмуляции разработчикам Star-Force пришлось очень глубоко зарыться в систему, в результате чего получился полный глюкодром и установка очередного пакета обновлений на Windows требует параллельной установки соответствующего обновления для Star-Force, что есть саксь и маст дай. Так что лучше похоронить эту тему раз и навсегда, даже не пытаясь возвращаться к ней, в противном случае, мы столкнемся с таким количеством проблем, осилить которые может только крупная компания, да и то…

серийные номера

Самое простое — защитить программу серийным номером, который элементарно программируется и никаких конфликтов ни с чем не вызывает. Конечно, это не помешает Пете скопировать понравившуюся ему программу у Васи или даже выложить серийный номер в сеть, где его смогут увидеть все желающие, но… С этим можно бороться путем ведения черного списка "засвеченных" серийных номеров, проверяя их на соответствие при установке обновлений. Если защищенная программа предполагает взаимодействие с удаленным сервером, процедура проверки валидности серийных номеров значительно упрощается и доступа к серверу нелегальный пользователь просто не получит (для сетевых игр это очень актуально!).

Несетевые программы так же могут периодически ломиться в сеть, передавая свой серийный номер. Если этот номер не подтверждается, регистрация считается недействительной и работа программы прекращается (форматировать в отместку диск или производить иные пакости категорически недопустимо!). При этом возникают следующие проблемы: поддержка выделенного и постоянно работающего сервера стоит денег, к тому же такой сервер сам по себе представляет весьма нехилую мишень для атаки. Что если хакеры завесят его или, что еще хуже, проникнуть внутрь и украдут легальные номера? Значит, необходимо содержать толкового администратора и разрабатывать протокол проверки серийных номеров с учетом возможного перехвата, например, использовать несимметричную криптографию — тогда украденная база не позволит восстановить ни один легальный сернум. Но это мелочи. Технические детали, решаемые пивом и хорошей травой. Как быть с персональными брандмауэрами, популярность которых все растет? Пользователь просто не выпустит программу в сеть! И хотя любой брандмауэр легко обойти установкой своего драйвера, так поступать нельзя. Журналисты тут же поднимут шумиху и смешают программиста с дерьмом. Требовать же обязательного наличия Интернета — нельзя. Еще не у всех он есть (но если уж требовать, необходимо как минимум

уметь работать через рогху). Правда, можно прибегнуть к одному очень хитрому маневру. Через функцию InternetGetConnectedState определить доступен ли в настоящее время Интернет и если сетевое соединение, действительно, присутствует, потребовать от пользователя открыть брандмауэр или что там у него есть. Популярная программа Macro Express именно так и поступает, но... этот путь несвободен от проблем. Если выход осуществляется через локальную сеть, функция InternetGetConnectedState не сможет сказать есть ли оттуда выход в Интернет или его нет. С удаленным доступом по модему все намного проще, однако, не стоит забывать, что пользователь может подключаться не только к провайдеру, но и своему приятелю у которого стоит импровизированный "сервер", который не предоставляет доступа в Интернет, но ведь InternetGetConnectedState этого не объяснишь! Наконец, системную библиотеку WININET.DLL, экспортирующую функцию InternetGetConnectedState, очень просто хакнуть в контексте памяти ломаемого приложения и тогда защите настанут кранты.

криптография и все что с ней связно

Вместо серийных номеров некоторые разработчики предпочитают использовать криптографические ключи и прочие системы цифровой подписи типа сертификатов, усиленно продвигаемые на рынок компанией Microsoft и прочими гигантами. Так вот, все это чушь собачья! Цифровая подпись хорошо работает только в связке клиент-сервер, но на локальной машине она бессмысленна! Да, сгенерировать "левый" ключ в этом случае невозможно, но если как следует пошурудить hiew'ом в исполняемом файле, никакие ключи после этого уже не понадобятся! Поэтому, цифровая подпись должна использоваться совместно с шифровкой кода и проверкой целостности!

Можно (и нужно) шифровать ключевым файлом критические модули защищаемой программы, однако, главное тут — не перестараться! Независимо от алгоритма реализации, такая защита вскрывается при наличии одного-единственного ключа, кроме того она не защищена от его (ключа) распространения.

оборудование — физическое и виртуальное

Привязка к аппаратуре — это грязный и пакостный трюк, отравляющий жизнь легальным пользователям и все равно не спасающий от взлома, поскольку любая привязка опытным хакером обнаруживается элементарно и либо эмулируется, либо вырезается. Виртуальные машины типа VM Ware представляют собой большую проблему. На варезных серверах уже появились программы с примечанием: "запускать под VM Ware"! Достаточно один раз зарегистрировать программу и дальше ее можно тиражировать сколько угодно.

Распознать виртуальные машины довольно легко (они несут на своем бору довольно специфичный набор оборудования), однако, уже появились патчи, скрывающие присутствие VM Ware, и, что хуже всего, многие легальные пользователи предпочитают запускать второстепенные утилиты из под виртуальных машин, чтобы не засирать основную систему. Защита не может, просто не имеет ни морального, ни юридического права отказывать VM Ware в исполнении, в противном случае, пользователи могут снести защиту на хрен мотивировав это "производственной необходимостью" и будут правы (правда, это не освободит их от регистрации, но это уже другой разговор). А некоторые могут даже в суд подать, за умышленной ограничение функциональности не отраженное на упаковке. Вот такая, значит, ситуация. Но это ладно. Это все философия. Перейдем к практике.

Привязываться лучше всего к жесткому диску — как показывает практика, он меняется реже всего. Чтобы прочесть серийный номер совершенно необязательно писать свой драйвер! Достаточно воспользоваться библиотекой ASPI (но она не установлена по умолчанию) или SPTI (есть только в NT/XP и требует прав администратора). Конкретные примеры реализации можно найти в "Технике защиты лазерных дисков", однако все они… ну, скажем так, непрактичны, неудобны и вообще нежизнеспособны. А вот древний мыщъхиный трюк: вызываем GetVolumeInformation и смотрим на размер тома в байтах. Не слишком-то уникальная информация, но все-таки на другой машине он с вероятностью близкой к единице будет иной. К тому же такая проверка очень просто реализуется, стабильно работает на всем семействе Windows-подобных систем (включая эмуляторы) и не требует никаких прав.

компиляция on demand

При наличии ресурсов, можно установить сервер, генерирующий программы индивидуально для каждого пользователя, специально под его регистрационные данные. Это

выглядит так: пользователь скачивает крошечный инсталлятор. Инсталлятор извлекает с компьютера ключевую информацию, необходимую для привязки (например, размер тома), спрашивает имя пользователя и передает эту информацию серверу, который жестко прописывает все это "хозяйство" в исходном тексте, перекомпилирует его, зашифровывает любым понравившимся навесным упаковщиком и отсылает назад. Что выигрывает разработчик?

Во-первых, "отломать" защиту становится намного сложнее, поскольку она не спрашивает никаких серийных номеров, не требует ключевых файлов и хакеру просто не за что зацепиться, во-вторых, даже если программа и будет взломана, распространять ее придется только в исполняемом файле (в который легко внедрить "водяные знаки", идентифицирующие владельца, например, зашифрованный MAC адрес его сетевой карты). К исполняемым файлам, добытым противоестественным путем (то есть скаченным из ненадежных источников), народ испытывает традиционное недоверие и далеко не каждый рискнет запускать их. Красота! Вот на этой торжественной ноте мы и закончим с привязкой.

как затруднить распаковку

Откомпилированная программа обычно подвергается упаковке. И цель здесь совсем не в уменьшении размера, а в затруднении анализа. Упаковщик набивает программу антиотладочными приемами и прочей бодягой, затрудняющей пошаговую трассировку или даже делающей ее невозможной. На самом деле, хакер ничего трассировать и не собирается. Он просто снимает с работающего приложения дамп и дизассемблирует его (реконструировать exe-файл для этого необязательно). Надежно противостоять снятию дампа на прикладном уровне нельзя, а спускаться на уровень драйверов как-то не хочется. Некоторые защиты искажают PE-заголовок, гробят таблицу импорта и используют другие грязные трюки, затрудняющие дампинг, но не предотвращающие его в принципе.

Глобальные инициализированные переменные. Мыщъх призывает идти другим путем. Не препятствовать снятую дампа, но сделать полученный образ бесполезным. Для этого достаточно использовать глобальные инициализированные переменные, "перебивая" их новыми значениями, например:

```
char *p = 0;                    // глобальная переменная 1
DWORD my_icon = MY_ICON_ID;     // глобальная переменная 2
…
if (!p) p = (char*) malloc(MEM_SIZE);
my_icon = (DWORD) LoadIcon (hInstance, my_icon);
```

Листинг 6 защитный механизм, предотвращающий снятие дампа с работающей программы путем использования инициализированных глобальных переменных

Задумаемся, что произойдет, если сбросить дамп с работающей программы? А произойдет вот что: в переменной p окажется указатель на когда-то выделенный блок памяти, условие (!p) обломится и новая память выделена не будет (!), а при обращении по старому указателю произойдет исключение. То есть, изготовить исполняемый файл из дампа хакер уже не сможет! Как минимум придется восстановить значения всех глобальных переменных, а это — геморрой! Ладно, изготовить исполняемый файл из дампа нельзя, но ведь дизассемблировать его можно? А вот и ни хрена!

После выполнение функции LoadIcon переменная my_icon будет содержать не идентификатор иконки, а ее обработчик! То есть, хакер не сможет установить, что это за иконка такая (строка, битмап или другой ресурс) и ему придется обращаться к отладчику, противостоять которому намного проще, чем дизассемблеру. Кстати говоря, такой прием экономит память и широко используется во многих программах (например, в стандартном "Блокноте" — попробуйте снять с него дамп и обломайтесь).

Стартовый код. Единственная надежда хакера — отловить момент завершения распаковки и тут же сбросить дамп, пока защита еще не успела нагадить в глобальные переменные. Пошаговая трассировка исключается (ей очень легко противостоять) и остается... только хвост! Он же стартовый код. Он варьируется от компилятора к компилятору, и в случае с Microsoft Visual C++ MFC выглядит так:

```
.text:00402A82        push    ebp
.text:00402A83        mov     ebp, esp
.text:00402A85        push    0FFFFFFFFh
.text:00402A87        push    offset unk_403748
.text:00402A8C        push    offset loc_402C06
.text:00402A91        mov     eax, large fs:0
```

```
.text:00402A97        push    eax
.text:00402A98        mov     large fs:0, esp
.text:00402A9F        sub     esp, 68h
...
.text:00402BAA        call    ds:GetModuleHandleA
.text:00402BB0        push    eax
.text:00402BB1        call    _WinMain@16     ; WinMain(x,x,x,x)
```

Листинг 7 типичный представитель стартового кода

Точка останова на GetModuleHandleA. Вызов API-функции GetModuleHandleA сразу же бросается в глаза. Если хакер установит сюда точку останова, отладчик/дампер "всплывет" в start-up коде, еще до передачи управления WinMain (так же можно поставить точки останова на GetVesion/GetVersionEx, GetCommandLine, GetStartupInfo и т. д.). Если точка останова программная, распаковщик может обнаружить ее по наличию CCh в начале API-функции, и, с некоторой долей риска, снять. Если второй байт функции равен 8Bh, то это, очевидно, стандартный пролог, первый (оригинальный) байт которого равен 55h. Получаем права на запись через VirtualAlloc, меняем CCh на 8Bh и продолжаем распаковку в обычном режиме. Пусть хакер крякнет! Правда, в последующих версиях Windows пролог API-функций может быть модифицирован и тогда этот трюк не сработает.

Отладочные регистры. Аппаратную точку останова можно обнаружить через чтение регистров Drx. Команда mov eax,DrX на прикладном уровне приводит к исключению, кроме того, отладчик (теоретически) может отслеживать обращение к отладочным регистрам, чтобы маскировать свое присутствие – все необходимое для этого x86 процессоры предоставляют, но! Если распаковщик прочитает свой контекст, он сможет дотянуться и до Drx, причем не только на чтение, но и на запись! То есть, можно не только обнаружить точки останова, но и обезвредить их! Весь вопрос в том, как получить контекст. Чтение SDK выявляет API-функцию GetThreadContext, которая как раз для этого и предназначена, однако, пользоваться ей нельзя, иначе хакер установит сюда точку останова и защита проиграет войну! Надо действовать так: регистрируем из распаковщика собственный обработчик SEH, возбуждаем исключение (делим на ноль, обращаемся по недействительному указателю) и... получаем контекст в одном из аргументов структурного обработчика. Что остается делать хакеру? Правильно! Устанавливать точку останова на fs:0, где и хранится указатель на SEH обработчик (только до этого додумается не каждый хакер).

Структурные исключения. Кстати, о fs:0. Первое, что делает стартовый код, это регистрирует собственный SEH-обработчик, поэтому установка точки останова на fs:0 позволяет хакеру всплыть сразу же после завершения распаковки, следовательно, распаковщик должен обращаться к этой ячейке как можно чаще. Десятки или даже тысячи раз, причем ложить туда не абы что, а именно ESP, иначе хакер установит условную точку останова (soft-ice это позволяет) и легко обойдет защиту.

Поиск по сигнатуре. Козырный хакерский трюк — взломщик снимает дамп с работающей программы, находит там стартовый код по сигнатуре, определяет его адрес и ставит на его начало аппаратную точку останова (программную ставить нельзя, она будет затерта при распаковке). Разработчику защиты необходимо либо распознавать аппаратные точки останова и снимать их, действия по методике, описанной выше, либо использовать модифицированный стартовый код, который не сможет распознать хакер.

Контроль за $PC. А вот еще один трюк. Большинство распаковщиков располагаются в стороне после распакованного кода и при передаче управления на оригинальную точку входа прыгают куда-то далеко. Хакер может использовать этот факт как сигнал, что распаковка уже завершена. Конечно, установить аппаратную точку останова на это условие уже не удастся и придется прибегнуть к пошаговой трассировке (которой легко противостоять), но ради этого случая хакер может написать и трейсер нулевого кольца. Это не сложно. Сложно определить когда же заканчивается распаковка. Контроль на $pc (в терминологии x86 – eip) – это единственный универсальный способ, который позволяет это сделать без особых измен и чтобы обломать хакера, распаковщик должен как бы "размазывать" себя вдоль программы, тогда он победит!

боремся с отладчиком

Распространенный миф гласит, что штурмовать отладчик уровня soft-ice можно только из ядра, а с приказного уровня надежно обуть его невозможно. Это не так. В частности, эффективная отладчика форт-программ (равно как и другого p-кода) под soft-ice невозможна. Отладчик просто вращается внутри форт-машины, наматывая мили на кардан, хакер материться,

меняет одну сигарету за другой, но ничего конструктивного предложить не может. Разве что написать декомпилятор, но это требует времени, которого нету.

Антиотладка. К слову сказать, использовать антиотладочные приемы следует с большой осторожностью, а лучше не использовать их вообще. То, что работает под 9x, зачастую не работает под NT и наоборот, обращение к недокументированным структурам операционной системы ставит программу в зависимость от левой пятки Microsoft, привязывая ее к текущей версии Windows. Кроме того, многие антиотладочные приемы, опубликованные в различных статьях, на проверку оказываются сплошной фикцией. Отладчик работает нормально и на них не ведется, а вот глюки идут косяками. К тому же следует помнить о существовании такой штуки как IceExt – специальной примочки для soft-ice, скрывающей его от многих защитных механизмов. Всегда проверяйте все применяемые приемы на вшивость и не только под soft-ice+IceExt, но и альтернативных отладчиках типа OllyDbg. Совершенно недопустимо отказываться работы в присутствии пассивного отладчика. Аргумент "soft-ice держат на компьютере только хакеры" идет лесом, то есть на ху… в смысле на хутор. Туда же посылаются и разработчики. Необходимо противодействовать лишь активной отладке. Нейтрализации точек останова (о которой мы уже говорили) обычно бывает вполне достаточно. Не нужно мешать отладчику — нехай хакер трассирует! Главное — сделать этот процесс максимально неэффективным.

Вероятностное поведение программы. Пусть вызовы защитных функций следует из разных мест с той или иной вероятностью, либо определяемой оператором типа rand(), либо действиями пользователя. Например, если сумма последних шести символов, набранных пользователем, равна 69h — происходит "внеплановый" вызов защитного кода. Если программа при каждом прогоне ведет себя слегка по разному, реконструкция ее алгоритма чрезвычайно усложняется.

Взлом по следам, взлом без следов. Основная ошибка большинства создателей защит заключается в том, что они дают хакеру понять, что защита распознала попытку взлома, а этого делать ни в коем случае не следует! Пускай хакер сам догадывается, перелопачивая тонны машинного кода. Ах, если бы только мы могли не выводить диалоговое окно с надписью "неверный серийный номер/инвалидный ключ", ведь тогда хакеру остается только поставить точку останова и посмотреть на код, который его выводит — защитный механизм сразу же будет пойман за хвост! Приходится идти в обход: вместо немедленного выполнения какого-либо действия, программист создает список отложенных процедур (просто массив указателей на функции) и проверяет его в цикле выборки сообщений или во время простоя системы из отдельного потока. Один поток проверяет регистрационные данные и кладет сюда указатель на функцию, которую нужно выполнить вместе с другими функциями, выполняемыми программой. Все! Цепь разомкнулась! Простая трассировка топит отладчик в цикле выборки сообщений и хакеру приходится разбираться со всеми этими списками, очередями и т. д., что в отладчике сделать очень непросто. Для реконструкции алгоритма требуется помощь дизассемблера, с которым мы еще успеем побороться! Пока же мыщъх даст несколько простых но полезных советов.

Полезные советы россыпью. Поверяйте серийный номер (пароль) не с первого байта (лучше всего с пятого), этим защита обломает начинающего хакера, установившего точку останова на начало. Так же не проверяйте последние символы серийного номера. Ага! И не проверяйте середину! Проверяйте не более половины символов вразброс. Смешно, конечно, он очень выручает. Никогда не считывайте серийный номер (пароль, имя пользователя) из строки редактирования все целиком! Хакер тут же найдет его в памяти и расставит точки останова, в которые угодит защитный механизм. Считывайте только по одному вводимому символу за раз (через WM_CHAR или DDE) и тут же их шифруйте (если скалывать считанные символы в локальный буфер, то получится тот же самый WM_GETTEXT, только реализованный своими руками. На фиг! В топку!).

боремся с дизассемблером

Лучший из всех дизассемблеров IDA PRO. Это, действительно, очень мощный дизассемблер, который не так-то просто взять на испуг. С классическими защитными приемами (типа прыжка в середину команды) он справляется, даже не замечая, что тут было что-то защитное. Если на пути хакера встретиться стена зашифрованного/упакованного кода, он напишет короткий (длинный) скрипт и расшифрует все, что нужно даже не выходя из дизассемблера! Для p-кода будет написан отдельный плагин типа "докомпилятор".

Самым мощным анти-хакерским средством был и остается косвенный вызов функций по указателю, передаваемому в качестве аргумента. Хакер натыкается на что-то типа call eax и материться в бессильной злобе, пытаясь определить, что содержится в eax на данный момент. Если материнская функция вызывается обычным способом, взломщику достаточно просто перейти по перекрестной ссылке, заботливо сгенерированной IDA и подсмотреть что передается функции, однако, если указатель инициализируется далеко от места вызова, хакеру придется раскрутить всю цепочку вызовов, что нелегко, а если функция, вызывающую косвенную функцию, сама вызывается косвенным путем, то это вообще кратны! Остается только запускать отладчик, устанавливать точку останова на call eax и смотреть ее значение в живую, однако, точкам останова легко противостоять (и мы об этом уже говорили), к тому же в различные моменты времени eax может указывать на разные функции и тогда простое подглядывание eax ничего не даст! Дизассемблер ослепнет и оглохнет!

Рассмотрим следующий пример:

```
sub_sub_demo(int a, void *p, void *d)
{
        // printf("sub_sub_demo\n");
        if (--a) return ((int(*)(int, void*, void*))p)(a, p, d);
        return 0;
}

sub_demo(int a, void *p, void *d)
{
        // printf("sub_demo\n");
        if (--a) return ((int(*)(int, void*, void*))d)(a, p, d);
        return 0;
}

demo(int a, void *p, void *d)
{
        // printf("demo\n");
        ((int(*)(int, void*, void *))p)(a, p, d);

}
main()
{
        demo(0x69,sub_demo, sub_sub_demo);
}
```

Листинг 8 пример программы, вызывающий функции по указателю

В исходном тесте все понятно. Функция main вызывает функцию demo, передавая ей указатели на sub_demo и sub_demo, которые поочередно вызывают друг друга, каждый раз уменьшая счетчик на единицу. Короче, мы имеем цикл. Но какой! Вы только посмотрите на его дизассемблерный код! Хвост отсохнет!

```
.text:00401000 loc_401000:                             ; DATA XREF: _maino
.text:00401000         mov     ecx, [esp+4]
.text:00401004         dec     ecx
.text:00401005         jz      short loc_401018
.text:00401007         mov     eax, [esp+0Ch]
.text:0040100B         push    eax
.text:0040100C         mov     eax, [esp+0Ch]
.text:00401010         push    eax
.text:00401011         push    ecx
.text:00401012         call    eax
.text:00401014         add     esp, 0Ch
.text:00401017         retn
.text:00401020
.text:00401020 loc_401020:                             ; DATA XREF: _main+5o
.text:00401020         mov     ecx, [esp+4]
.text:00401024         dec     ecx
.text:00401025         jz      short loc_401038
.text:00401027         mov     eax, [esp+0Ch]
.text:0040102B         mov     edx, [esp+8]
.text:0040102F         push    eax
.text:00401030         push    edx
.text:00401031         push    ecx
.text:00401032         call    eax
.text:00401034         add     esp, 0Ch
.text:00401037         retn
.text:00401040
```

```
.text:00401060 _main   proc near                    ; CODE XREF: start+AFp
.text:00401060         push    offset loc_401000
.text:00401065         push    offset loc_401020
.text:0040106A         push    69h
.text:0040106C         call    sub_401040
.text:00401071         add     esp, 0Ch
.text:00401074         retn
.text:00401074 _main   endp
```

Листинг 9 дизассемблерный листинг, демонстрирующий всю мощь косвенного вызова функций

IDA Pro не смогла распознать функции и хотя восстановила перекрестные ссылки на main, они еще ни о чем не говорят! Ведь функции sub_demo и sub_sub_demo вызываются совсем не оттуда. Возьмем "функцию" loc_401000. Она принимает указатель в качестве аргумента и тут же его вызывает. А что это за указатель? Да хвост его знает! Ведь перекрестной ссылки на материнскую функцию нет! Единственный способ установить истину — проанализировать все цепочку вызовов с самого начала программы — с функции main, но слишком трудоемко, так что даже не обсуждается (особенно, если ломать реальную программу), но даже в этом случае приходится постоянно следить за указателями, которые пляшут как кони и ни хрена не понятно, что находится в них в каждый конкретный момент!

Защиту можно значительно усилить, если реализовать модель Маркова — функцию, возвращающую указатель на функцию. Такой прием программирования не слишком популярен, поскольку непривычен и лишен языковой поддержки (язык Си вообще не позволяет объявлять функции, возвращающие указатели на функции, поскольку такие определения рекурсивны, и программисту приходится возиться с постоянным преобразованием типов, что не слишком украшает программу и является потенциальным рассадником ошибок). Тем не менее, на автоматическое дизассемблирование моделей Маркова не способен ни один дизассемблер, включая IDA PRO.

боремся с мониторами

В заключении рассмотрим пару способов борьбы с файловыми мониторами и мониторами реестра. Самое простое, что только можно придумать, через FindWindow находить главное окно монитора и затем либо закрывать его на хрен, либо посылкой специальных Window-сообщений, удалять из списка "свои" обращения. Естественно, если хакер переименует окно (что можно сделать через FindWindow/SetWindowText), защита обломается по полной программе, так что этот прием слишком ненадежен.

А что надежно? Хранить флаг регистрации вместе с настойками в одном ключе реестра в двоичном виде, тогда его мониторинг ничего не даст. Хакер видит, что из такой ветви читается куча байтов, но вот какой из них за что отвечает — ему неведомо и выяснить это можно только путем отладки/дизассемблирования защищенной программы, чему несложно противостоять (см. все вышеописанные методики).

заключение

Мы рассмотрели множество эффективных и при этом системно-независимых защитных техник, работающих на прикладном уровне и не сильно усложняющих реализацию. Как видно, даже в таких жестких условиях можно многое сделать!