TOP10 ошибок защитников программ

аргентинский болотный бобер nezumu el raton ака толстый жирный нутряк ибн мыщъх

> **создать качественную защиту от взлома в общем-то несложно, для этого даже не обязательно знать ассемблер и быть с операционной системой на "ты". почему же тогда программы ломаются косяками? во всем виноваты мелкие (и крупные) ошибки разработчиков, которых очень легко избежать, если, конечно, заранее знать где сало, а где капкан**

введение

Несмотря на разнообразие трюков и приемов, используемых создателями защит, большинство программ ломаются по одному и тому же набору стандартных шаблонов. Ошибки разработчиков удручающее однообразны — никакой тебе тяги к творчеству, никакого морального удовлетворения от взлома и вместо интеллектуальной игры, вместо смертельного поединка с защитой, хакерам приходится ковыряться в чем-то очень сильно неаппетитном, напоминающем чей-то наполовину разложившийся труп. Труп мертворожденных идей, надерганных программистами из древних мануалов, которые уже давно неактуальны.

После выхода из очередной депрессии, спровоцированной чрезмерной дозой феназепама, мыщъх закинулся пирацетамом и решительным движением хвоста набросал TOP "излюбленных" ошибок, показывающий *как не нужно защищать программы*, чтобы они ломались не сразу, а только под пыткой в застенках soft-ice типа "Гестапо".

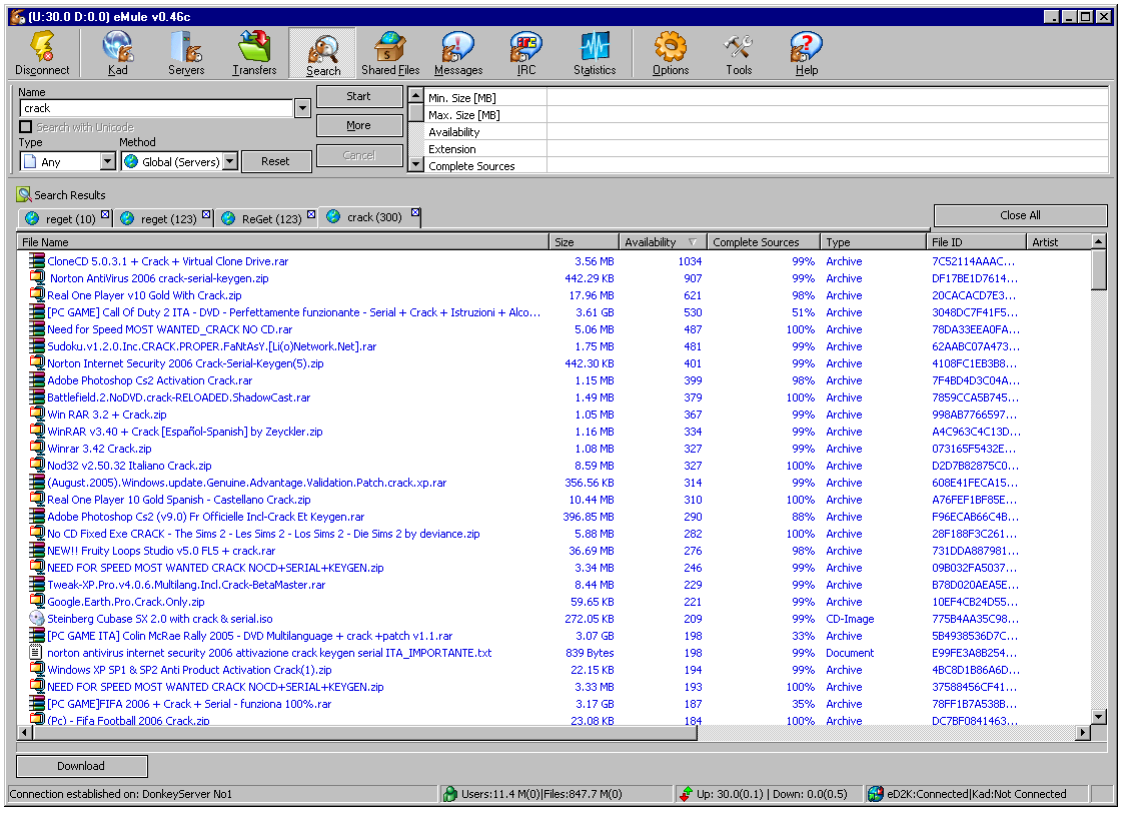


Рисунок 1 ломается все! практически для любой программы можно найти готовый crack всего за несколько секунд

ошибки стратегического типа — разбор полетов

Начнем с концептуальных ошибок, "благодаря" которым программу может взломать любой начинающий хакер или даже продвинутый пользователь. Мыщъх не только показывает как ломаются программы, но и поясняет, что нужно сделать, чтобы этого избежать.

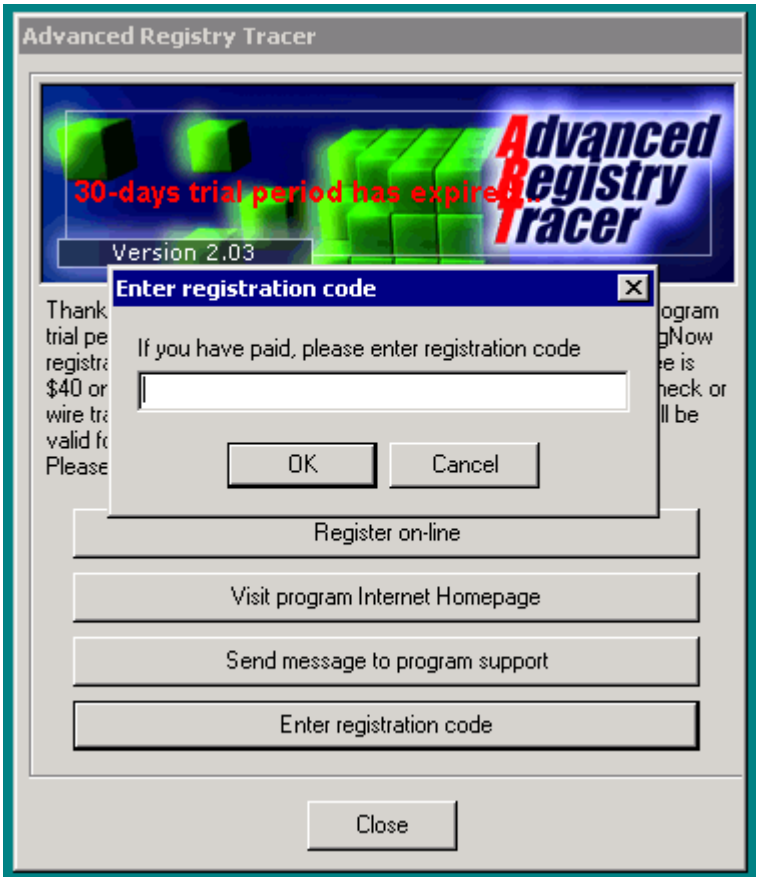
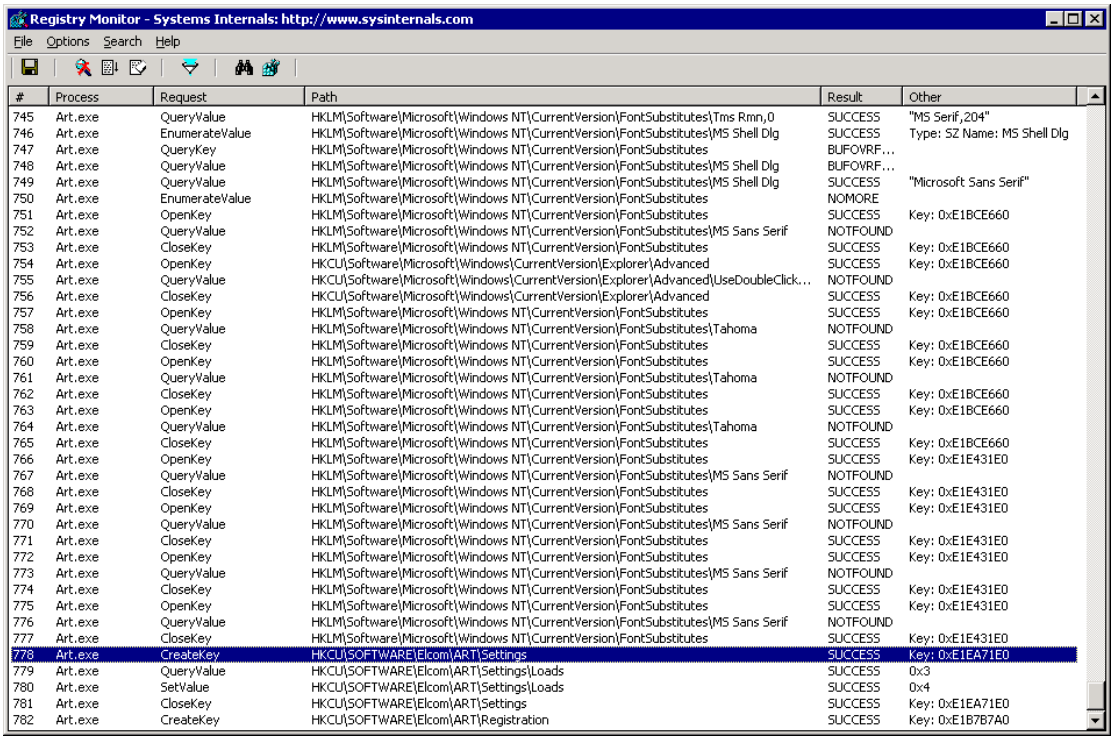


Рисунок 2 типичная реакция программы на окончание испытательного срока, и хотя на первый взгляд защита кажется неприступной, справится с ней может даже ламер, совершенно не разбирающийся в ассемблере, но вооруженный "правильными" утилитами

1: смывайте воду и выключайте свет

Для программ, защищенных триальным сроком, характерна **проблема реинсталляции**. Когда испытательный период заканчивается и программа говорит "мяу" (см. рис. 2), среднестатистический пользователь вместо того, чтобы зарегистрироваться, просто удаляет ее с компьютера и тут же устанавливает вновь, надеясь, что она заработает как новая. Специально для таких козлов, инсталлятор оставляют на компьютере секретный скрытый знак, не удаляемый деинсталлятором. Обнаружив, что программа уже была ранее установлена на этом компьютере, защита блокирует запуск и говорят "мяу" еще раз. На первый взгляд, защита кажется неприступной, но… обнаружить и удалить скрытый знак может даже ламер!

Это делается так: перед установкой программы с компьютера снимается полный дамп (антивирусные ревизоры помогают сформировать список файлов, а утилиты "принудительной деинсталляции" типа Advanced Registry Tracer создают "слепок" реестра). После установки программы создается еще один дамп, который сравнивается с первым. Все тайное становится явным! Если же первый дамп по каким-то причинам не был сделал (юзер спохватился только *после* окончания триального срока), не беда — запускаем файловый монитор вместе с монитором реестра Марка Руссиновича (www.sysinternals.com) и смотрим, что именно "не нравится" защите, то есть к каким именно потайным уголкам она обращается (см. рис 3).



#	Process	Request	Path	Result	Other
745	Art.exe	QueryValue	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes\Tms Rmn,0	SUCCESS	"MS Serif,204"
746	Art.exe	EnumerateValue	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes\MS Shell Dlg	SUCCESS	Type: SZ Name: MS Shell Dlg
747	Art.exe	QueryKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes	BUFOVRF...	
748	Art.exe	QueryValue	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes\MS Shell Dlg	BUFOVRF...	
749	Art.exe	QueryValue	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes\MS Shell Dlg	SUCCESS	"Microsoft Sans Serif"
750	Art.exe	EnumerateValue	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes	NOMORE	
751	Art.exe	OpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes	SUCCESS	Key: 0xE1BCE660
752	Art.exe	QueryValue	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes\MS Sans Serif	NOTFOUND	
753	Art.exe	CloseKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes	SUCCESS	Key: 0xE1BCE660
754	Art.exe	OpenKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	Key: 0xE1BCE660
755	Art.exe	QueryValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced\UseDoubleClick...	NOTFOUND	
756	Art.exe	CloseKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	Key: 0xE1BCE660
757	Art.exe	OpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes	SUCCESS	Key: 0xE1BCE660
758	Art.exe	QueryValue	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes\Tahoma	NOTFOUND	
759	Art.exe	CloseKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes	SUCCESS	Key: 0xE1BCE660
760	Art.exe	OpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes	SUCCESS	Key: 0xE1BCE660
761	Art.exe	QueryValue	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes\Tahoma	NOTFOUND	
762	Art.exe	CloseKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes	SUCCESS	Key: 0xE1BCE660
763	Art.exe	OpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes	SUCCESS	Key: 0xE1BCE660
764	Art.exe	QueryValue	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes\Tahoma	NOTFOUND	
765	Art.exe	CloseKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes	SUCCESS	Key: 0xE1BCE660
766	Art.exe	OpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes	SUCCESS	Key: 0xE1E431E0
767	Art.exe	QueryValue	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes\MS Sans Serif	NOTFOUND	
768	Art.exe	CloseKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes	SUCCESS	Key: 0xE1E431E0
769	Art.exe	OpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes	SUCCESS	Key: 0xE1E431E0
770	Art.exe	QueryValue	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes\MS Sans Serif	NOTFOUND	
771	Art.exe	CloseKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes	SUCCESS	Key: 0xE1E431E0
772	Art.exe	OpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes	SUCCESS	Key: 0xE1E431E0
773	Art.exe	QueryValue	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes\MS Sans Serif	NOTFOUND	
774	Art.exe	CloseKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes	SUCCESS	Key: 0xE1E431E0
775	Art.exe	OpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes	SUCCESS	Key: 0xE1E431E0
776	Art.exe	QueryValue	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes\MS Sans Serif	NOTFOUND	
777	Art.exe	CloseKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes	SUCCESS	Key: 0xE1E431E0
778	Art.exe	CreateKey	HKCU\SOFTWARE\Elcom\ART\Settings	SUCCESS	Key: 0xE1EA71E0
779	Art.exe	QueryValue	HKCU\SOFTWARE\Elcom\ART\Settings\Loads	SUCCESS	0x3
780	Art.exe	SetValue	HKCU\SOFTWARE\Elcom\ART\Settings\Loads	SUCCESS	0x4
781	Art.exe	CloseKey	HKCU\SOFTWARE\Elcom\ART\Settings	SUCCESS	Key: 0xE1EA71E0
782	Art.exe	CreateKey	HKCU\SOFTWARE\Elcom\ART\Registration	SUCCESS	Key: 0xE1B7B7A0

Рисунок 3 монитор реестра позволяет отслеживать скрытые знаки, к которым обращается защищенная программа

Исход сражения с защитой можно предугадать заранее, но вот можно ли его предотвратить? Первое (и самое глупое), что только можно предложить — гадить в реестре и в файловой системе, оставляя целую навозную кучу "следов", которую пользователь запариться разгребать. Только некрасиво это. Какому пользователю такая программа понравится? Мотивация честной регистрации падает ниже абсолютного нуля и… даже честный зарубежный Джон обкладывает разработчика матом и не регистрируется.

Гораздо элегантнее будет оставить едва различимый и совершенно неочевидный след, например, изменить дату создания папки %windows%, поместив в поле десятых долей секунд свое "магическое" число. Да, конечно, мониторы успешно отследят эту нехитрую махинацию, но учитывая размер их логов, пользователь с высокой степенью вероятности просто не обратит на эту мелочь внимания (правда, возникает потенциальный конфликт с другими защитами).

А вот еще один трюк: создаем файл, делаем seek на весь размер свободного пространства, как бы "втягивая" его внутрь себя, а затем сканируем полученный файл на предмет наличия "своего" содержимого. hint: при удалении файлов с диска, они продолжают "догнивать" в свободных секторах довольно длительное время, поэтому защита может легко и прозрачно обнаружить была ли она установлена на данный диск или нет (к сканированию на уровне секторов для этого прибегать совершенно необязательно! достаточно просто сделать seek, ведь при выделении кластеров операционная система их не очищает, что является огромной дырой в безопасности). Конечно, хакер без труда обнаружит и отломает такую проверку, но простого пользователя она поставит в тупик, ну разве что он не воспользоваться специальными утилитами для физического удаления файлов, затирающих их содержимое, но утилит для физического удаления веток реестра нет! (это намек).

Самое надежное — "зашить" дату ограничения триального срока в саму программу еще на стадии компиляции. Поскольку, программы не выкладываются на сервер каждый день, длительность демонстрационного периода будет тем короче, чем позднее пользователь скачает программу, поэтому, испытательный срок лучше удлинить до 60 дней (вам, что жалко?) и обновлять программу на сервере не реже раза в месяц. Как бороться с повторными скачками? Ну… во-первых, если программа тяжелая, громоздкая и большая, далеко не каждому пользователю будет в радость каждый месяц перекачивать мегабайты данных по своему каналу (скоростной Интернет есть далеко не у всех), во-вторых, можно отдавать программу только после предварительной регистрации, тогда бедному пользователю придется каждый раз выдумывать себе разные адреса, менять ящики и т. д., что сильно напрягает и высаживает на жуткую измену, склоняющую пользователя к регистрации.

Как вариант можно сделать так, чтобы при первом запуске инсталлятор (*не содержащий тела в себе основного тела программы!*) собирал информацию о конфигурации и отправлял ее серверу. Сервер сверял ее со своей базой и затем либо отдавал программу, либо не отдавал ("сетевой инсталлятор" писать совершенно необязательно, лучше просто дать ссылку на временный линк, автоматически удаляющийся через несколько дней, что очень просто реализуется и решает проблемы "докачки"). Взломать такую защиту пользователю (даже очень и очень продвинутому) будет уже не под силу, да и хакеров она напряжет изрядно.

2: хронометраж обратного отсчета времени

Никогда не полагайтесь на системное время — его очень легко перевести назад! К тому же, существует множество утилит типа TrialFreezer, перехватывающих вызов API-функции семейства GetLocalTime и подсовывающих *отдельно взятой программе* подложную информацию, что намного удобнее, чем работать с переведенным временем, поскольку при этом страдают все приложения.

Что может сделать защита? Сбегать в Интернет за Атомным временем? А если юзер поставит брандмауэр? (А он его наверняка поставит, если только не лось). Вести счетчик запусков — прекрасная идея, только его очень легко обнаружить с помощью сравнения двух "соседних" дампов.

Надежнее всего — сканировать диск на предмет поиска самых разных файлов и смотреть на дату их создания, причем брать не только дату создания/последней модификации самого файла, но так же извлекать "штамп времени" из заголовков PE-файлов и динамических библиотек, которые можно обнаружить... в своем адресном пространстве без всякого обращения к файловой системе! Ведь скачивает же пользователь новые версии различных разделяемых библиотек, а многие антивирусы и другие "сторожевые" программы устанавливают модули, проецируемые на все процессы сразу. Конечно, данная методика определения времени не очень точна и пригодна лишь для *грубой оценки верхней границы времени использования*, однако, учитывая наличие службы "windows update" и довольно частный выход новых фиксов, точность определения вплотную приближается к одному-двум месяцам, что для триальных защит вполне достаточно.

3: сравнение различных версий одной и той же программы

Разработчик защиты должен считаться с тем, что у взломщика наверняка окажется несколько различных версий одной и той же программы. Что это значит в практическом плане? А то, что сравнивая их между собой, хакер быстро найдет где хранится жестко прошитая дата истечения испытательного срока, серийный номер и эталонный ключ (если каждая версия отпираются "своим" ключом) и т. д.

Возьмем к примеру популярный текстовой редактор TSE Pro, часть защиты которого реализована на его собственном интерпретируемом языке, скомпилированным в байт-код, не поддающийся дизассемблированию. А готовых декомиляторов, увы, нет. Тем не менее защита снимается за считанные секунды простым сравнением двух версий, установленных в различное время на различные машины (вообще-то, в данном случае, редактор достаточно установить в разные каталоги, поскольку никаких проверок на скрытые знаки в нем нет).

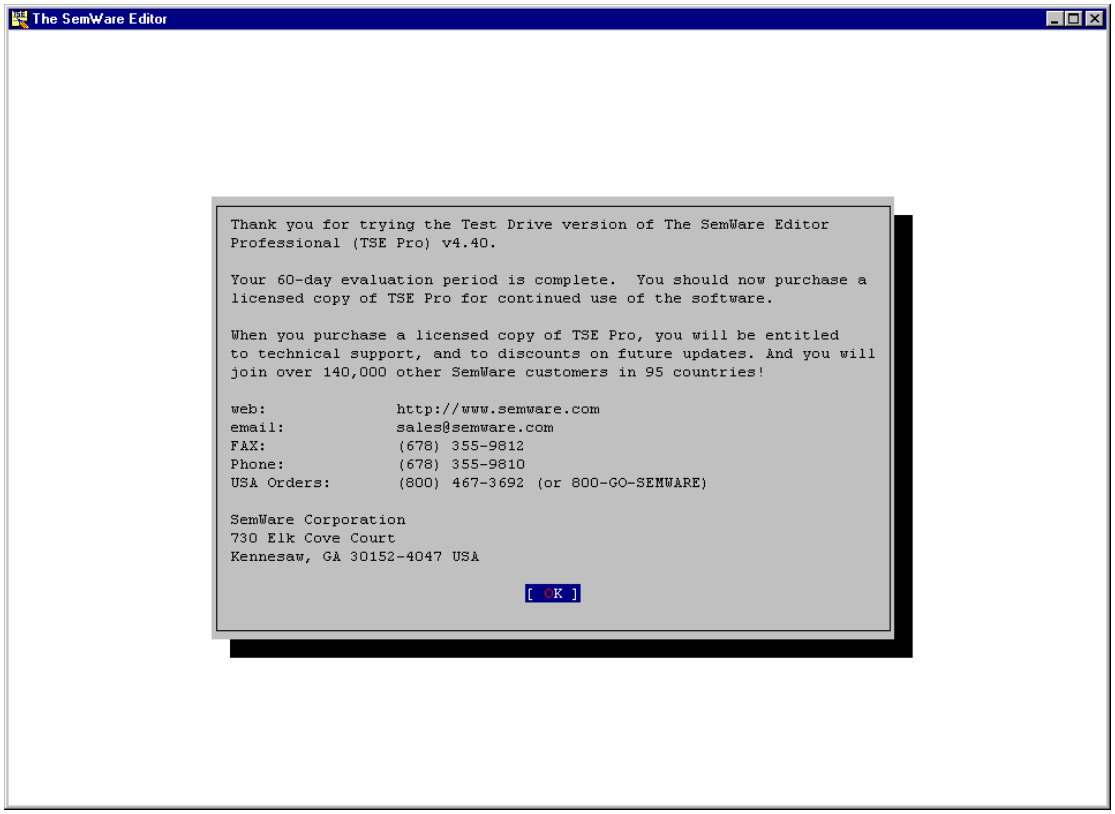


Рисунок 4 редактор TSE Pro отказывается запускаться, мотивируя это тем, что 60-дневный испытательный период уже истек

Утилита fc.exe из штатной поставки Windows, показывает, что время окончания испытательного срока "прошито" в файлах e32.mac и g32.exe:

```
$fc /b e32.mac e32.mac.old
Сравнение файлов e32.ma_ и E32.MAC.OLD
00000065: 06 05
00000066: D5 DD
00000067: C8 D4
```

Листинг 1 дата истечения демонстрационного перерода, прошитая внутри одного из файлов редактора TSE Pro, была мгновенно найдена с помощью утилиты fc

Нас обложили со всех сторон — сохранять время первого запуска на компьютере пользователя нельзя (найдет и удалит), жестко прошивать его в теле программы тоже (сравнит две версии и "переведет" дату вперед в hiew'е). Что же делать? **Скремблировать данные и код** — вот что! Попросту говоря, шифровать разные версии программы различными ключами, тогда прямое сравнение ничего не даст, если конечно, хакер не "распакует" программу, удалив распаковщик к едреням, но борьба с распаковщиками и пути противостояния ей — тема для отдельной статьи.

4: когда криптография бесполезна

Последнее время распространилась мода на несимметричную криптографию, цифровые подписи и всякие прочие сертификаты. Именно таким образом защищен The Bat. Создать генератор ключей, располагая только той информацией, которая заключена в защищенной программе, действительно ***невозможно***. Для этого требуется секретный ключ, а он есть только у разработчика защиты и ни у кого еще. Что делать? Атаковать локальную сеть компании-разработчика? Так ведь посадят на х!

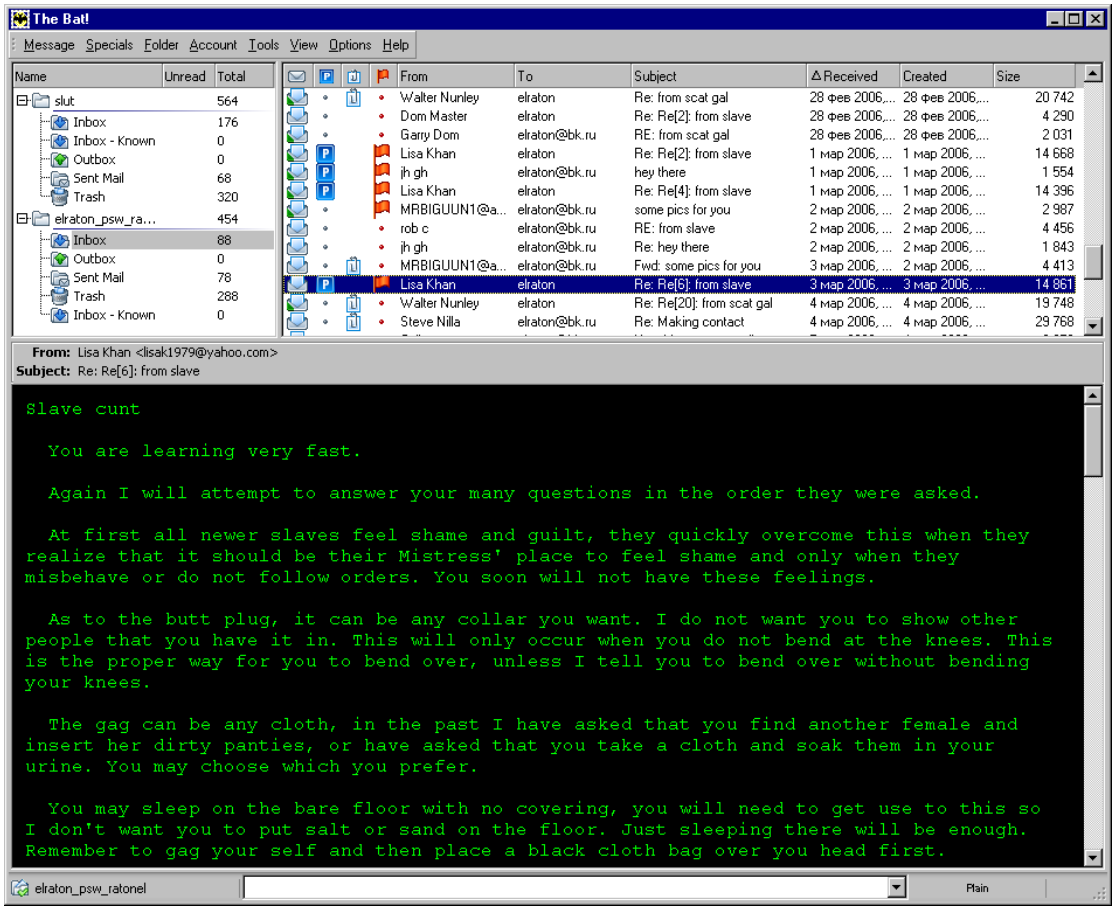


Рисунок 5 почтовый клиент The Bat, защищенный несимметричной криптографией

Хакеры поступают проще: "отламывают" защитный код или модифицирую открытый ключ, хранящийся в теле программы, заменяя его своим собственным открытым ключом, для которого существует известный секретный ключ. "Кряки" для Bat'a так и работают. И ведь работают же!!! *Даже самая навороченная криптографическая система в отсутствии механизмов контроля целостности программы — бесполезна, а контроль целостности легко найти и отломать*.

Исключение составляет тот случай, когда криптография используется для расшифровки критических фрагментов программы, без которых она неработоспособна, однако, неработоспособная программа никому не нужна, поэтому для триальных защиты такая методика не подходит. К тому же, если у хакера имеется хотя бы один-единственный рабочий экземпляр программы с валидным ключом, нейтрализация защиты — дело техники.

Вывод: ***несимметричную криптографию можно и нужно использовать только с тщательно проработанным механизмом проверки собственной целостности, со множеством проверок в разных местах.***

5: константы, говорящие сами за себя

Какой криптографический алгоритм лучше использовать — "стандартный" или "самопальный"? Большинство разработчиков склоняются в пользу первого, вызывая у хакеров бурное ликование.

Рассмотрим защитный механизм, контролирующий свою целостность с помощью надежного и хорошо апробированного CRC32. Как найти процедуру проверки среди десятков мегабайт постороннего кода? Очень просто — по стандартному полиному. Там, где есть CRC32, всегда присутствует и константа **EDB88320h**. Контекстный поиск обнаруживает стандартный полином за считанные секунды, ну а дальше по перекрестным ссылкам нетрудно найти саму процедуру проверки и тот код, что ее вызывает.

Существует множество готовых программ, распознающих стандартные криптографические алгоритмы. Вот только одна из них:

http://www.hexblog.com/2006/01/findcrypt.html. Это плагин для IDA Pro распространяющийся в исходных текстах на бесплатной основе.

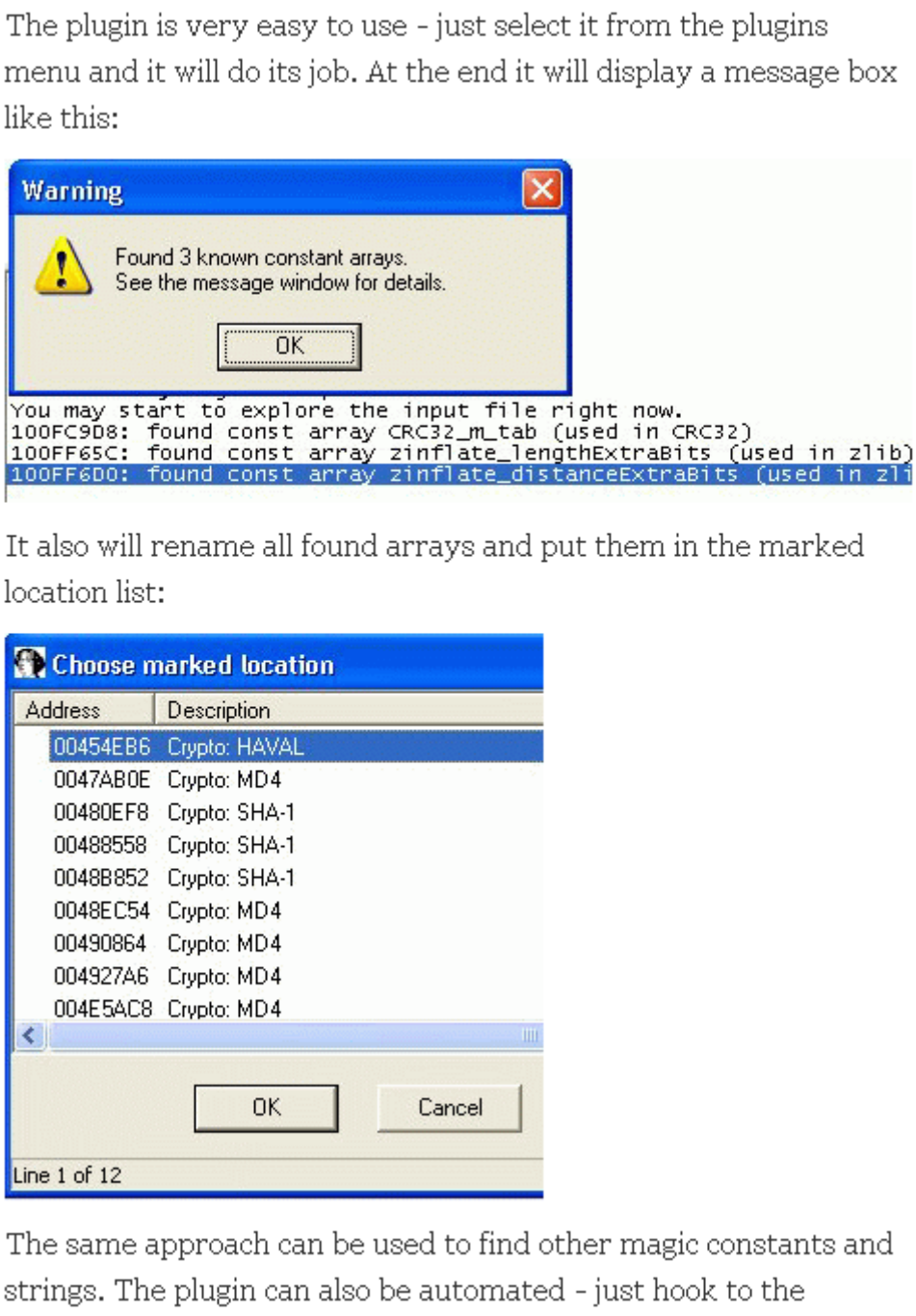


Рисунок 6 плагин к IDA Pro, распознающий стандартные криптографические алгоритмы

Используя стандартные алгоритмы, необходимо тщательно скрыть все легко узнаваемые полиномы и предвычисленные таблицы, по которым они могут быть легко локализованы в теле программы.

6: избыточная сложность, легко обнаруживаемая глазами

Процедура проверки серийного номера/ключа ни в коем случае не должна быть запутанной или чрезмерно сложной, иначе она будет существенно отличаться от всех остальных (обычных) процедур и опытный хакер распознает ее простым "визуальным" просмотром дизассемблерного листинга программы.

Позволю процитировать себе легендарного хакера Юрия Харона: *"...у меня, например, уже какой-то м-м-м овощной синдром (реплекс :) выработался — когда функция проверки "внешняя" по отношению к программе, она замечается при быстром пролистывании в ИДЕ, поскольку, выглядит "нехарактерно для кода" :)*

Короче, ищем код внешне отличающийся от всего остального. Лучше всего что бы этот код что-то долго и нудно вычислял. В нормальной программе _линейных_ фрагментов такого кода практически не встречается, а вот при создании защит все "перестраховываются" и пишут "очень сложные" свертки, etc. Вот по этой "навороченности" они достаточно легко находятся просто глазами" :). Но это, разумеется, может не сработать для сильно экзотического компилятора, и тогда мы ищем где нас просят ввести код/регистрацию или предупреждают о взломанном (самый частный случай взлома). На втором месте вышеописанное. Ну, а на третьем всё остальное :)"

Комментарии, как говориться, излишне. Господа программисты, если хотите защититься, ***не пишите слишком "навороченных" процедур***. Хакер все равно их расколет, пусть функция растянется хоть на тысячу строк, только локализовать ее будет не в пример легче.

7: несколько серийных номеров в одном

Как обычно ломаются программы? Ищется процедура, сравнивающая введенный серийный номер с эталонным, затем либо правится код, либо пишется генератор серийных номеров. Если же разные части программы в различное время будут проверять различные части одного и того же ключа, — вот тогда хакеру придется очень сильно поднапрячься, прежде чем довести взлом до ума.

Допустим, программа спрашивает серийник на запуске и пока он не будет введен, никуда дальше этого не пускает. ОК, хакер быстро "отламывает" защитный код (пишет генератор серийных номеров) и программа как будто бы запускается, но при расчете таблицы (попытке записи файла на диск) проверяет другую часть серийного номера с помощью дополнительной защитной функции, которую хакер на первой стадии взлома благополучно "проморгал". ОК, хакер вновь берет отладчик в руки и дорабатывает свой генератор (отламывает вторую проверочную процедуру). И программа работает уже в полный рост, только вот при выводе на печать… Ну в общем, вы поняли. Если хакер ломает программу "для себя" он будет долго материться и в конце концов ему это дело так надоест, что ее все-таки купит (доломает из спортивного интереса). А вот если программа ломается "на сторону" по спец заказу, то после первых двух-трех промахов клиент пошлет хакера на хрен и предпочтет заплатить, а не мучаться.

Один момент — ***серийный номер ни в коем случае не должен храниться в секции данных как глобальная переменная***, иначе перекрестные ссылки и аппаратные точки останова выдадут функции проверки с головой. Всегда передавайте серийный номер по цепочке локальных переменных тысячам посторонних функций программы! Тогда хакер никак не сможет отследить какие именно функции реально проверяют серийный номер, а какие его только передают по транзиту.

8: детерминированная логика

Редкая программа ломается в один присест и хакеру для преодоления защиты приходится предпринимать серию последовательных наступлений, рокировок и отступлений, планомерно продвигающих его в глубь, все ближе и ближе к сердцу защиты, часто ассоциируемом с "логовом дракона". При этом ставятся точки останова, картографируется маршрут трассировки и с каждым прогоном хакер чувствует себя все увереннее и увереннее. Начинающие ломатели вообще ограничиваются тем, что планомерно хачят один условный переход за другим в надежде найти тот единственный, что им нужен (а, зачастую, он единственный и есть).

Ситуация значительно усложняется, если программист применяет ***оружие недетерминированной логики***, или, попросту говоря, вызывает различные проверочные функции в случайное время из произвольных мест, используя функцию rand() или другой генератор подобного типа. В этом случае хакер не сможет повторить однажды пройденный маршрут, поскольку при следующем запуске программа пойдет совсем другим путем. Допустим, в прошлый раз хакер дотрассировал программу до точки А и понял, что свернул не на том повороте (в смысле — проскочил условный переход), и что сворачивать нужно было гораздо раньше, а теперь защитная функция уже позади и дальше трассировать некуда. ОК, он

перезапускает отладчик и… с превеликим удивлением обнаруживает, что его занесло совсем в другие места, совершенно незнакомые ему…

Саму библиотечную функцию rand() для этой цели использовать, конечно же, не стоит, иначе перекрестные ссылки выдадут все ветвления на блюдечке с голубой каемочкой или хакер отпадчит функцию rand() так, чтобы она всегда выдавала один и тот же результат, заставляющий программу ходить одним маршрутом. Лучше исследовать исходный код rand() и переписать его самостоятельно, непосредственно вживив в тело программы, тогда ломать программу будет очень и очень нелегко.

Допустим, мы имеем десять ***различных*** никак не зависимых друг от друга защитных функций, часть из них вызывается при каждом запуске программы, часть — через раз, а часть — с вероятностью раз несколько недель. Если защитные функции не выявляются ни по каким косвенным признакам, то хакеру придется полностью проанализировать весь код программы целиком, что нереально.

9: присутствие регистрационных данных в памяти

Классический способ взлома, уходящий своими корнями в эпоху времен ZX-SPECTRUM, это — ***прямой поиск регистрационных данных в памяти***. Хакер вводит серийный номер от балды (подсовывает программе "левый" ключевой файл), а затем ищет его в памяти и, если защита не предпринимает никаких дополнительных усилий, он действительного его находит. Остается установить точку останова на эти данные и терпеливо ждать пока защитный код, обращающийся к ним, не угодит в капкан. Процедура, ответственная за сравнение данных, введенным пользователем, с "эталоном" будет локализована и… безжалостно взломана.

Хитрые программисты поступают так: они посимвольно считывают клавиатурный ввод и ***тут же его шифруют***! Таким образом, данных, введенных пользователем, в памяти уже не оказывается и контекстный поиск теперь не срабатывает, обламывая хакера по полной программе.

10: защита в ассемблерных вставках

Хорошо продуманная защита не нуждается в ассемблере и уж тем более в ассемблерных вставках, выдающих защитный код с головой. Хотите знать почему? Если в функции отсутствуют ассемблерные вставки, оптимизирующие компиляторы выбрасывают стандартный пролог, адресуя локальные переменные и аргументы непосредственно через регистр ESP, но как только в теле функции появится хоть одна ассемблерная вставка, интеллекта компилятора для "сквозной" адресации через ESP уже оказывается недостаточно и он возвращается к стандартному прологу.

Проведем простой эксперимент. Возьмем следующую программу и откомпилируем ее компилятором Microsoft Visual C++ с максимальным режимом оптимизации (ключ /Ox).

```
main()
{
        int a,b=0;
        for (a=0;a<10;a++) b+=a*2;
        printf("%x\n",b);
}
```

Листинг 2 исходный код функции без ассемблерных вставок

Дизассемблерный листинг при этом будет выглядеть так:

```
.text:00000000 _main          proc near
.text:00000000                xor     ecx, ecx
.text:00000002                xor     eax, eax
.text:00000004
.text:00000004 loc_4:                                 ; CODE XREF: _main+C↓j
.text:00000004                add     ecx, eax
.text:00000006                add     eax, 2
.text:00000009                cmp     eax, 14h
.text:0000000C                jl      short loc_4
.text:0000000E                push    ecx
.text:0000000F                push    offset $SG398
.text:00000014                call    _printf
.text:00000019                add     esp, 8
.text:0000001C                retn
```

```
.text:0000001C _main          endp
```

Листинг 3 дизассемблерный листинг функции без ассемблерных вставок (стандартный пролог выброшен компилятором)

Как мы видим, ничего похожего на пролог тут нет! Но стоит нам добавить хотя бы простейшую ассемблерную вставку типа: `__asm {mov a,eax }` и перекомпилировать программу, как все полетит кувырком!

```
.text:00000000 _main          proc near
.text:00000000
.text:00000000 var_4          = dword ptr -4
.text:00000000
.text:00000000                push    ebp
.text:00000001                mov     ebp, esp
.text:00000003                push    ecx
.text:00000004                xor     ecx, ecx
.text:00000006                xor     eax, eax
.text:00000008
.text:00000008 loc_8:                                 ; CODE XREF: _main+10↓j
.text:00000008                add     ecx, eax
.text:0000000A                add     eax, 2
.text:0000000D                cmp     eax, 14h
.text:00000010                jl      short loc_8
.text:00000012                mov     [ebp+var_4], eax
.text:00000015                push    ecx
.text:00000016                push    offset $SG398
.text:0000001B                call    _printf
.text:00000020                add     esp, 8
.text:00000023                mov     esp, ebp
.text:00000025                pop     ebp
.text:00000026                retn
.text:00000026 _main          endp
```

Листинг 4 дизассемблерный листинг той же самой функции с мелкой ассемблерной вставкой (стандартный пролог выделен полужирным шрифтом)

Вот он, стандартный пролог, легко обнаруживаемый контекстным поиском! Поэтому, ***либо вообще не используйте никакого ассемблера в своих программах, либо пишите на чистом ассемблере с последующей трансляцией в obj, либо предваряйте ассемблерные функции спецификатором "naked", в этом случае Microsoft Visual C++ ни пролога, ни эпилога вставлять не будет.***

мелкие промахи, ведущие к серьезным последствиям

1) **категорически недопустимо бороться с пассивными отладчиками**! многие системщики постоянно держат soft-ice в фоне и совсем не для хакерских целей! защиты они уже давно не ломают — нет времени, да и программирование приносит гораздо большие деньги, но когда необходимая им программа ругается на soft-ice, отказываясь запускаться, вот тогда-то они выседают на ярость и, тряхнув стариной, разносят защиту в пух и прах, очень часто при этом выкладывая crack на всеобщее обозрение;
2) **не нужно пытаться обнаружить виртуальные машины** — все равно не получится! их слишком много: VM Ware, VirtualPC, BOCHS, QEMU… к тому же многие пользователи и сетевые/журнальные обозреватели не желая "засирать" свою основную систему, "обкатывают" новые программы именно под виртуальными машинами и если те отказываются там запускаться, выбор делается в пользу конкурентной программы;

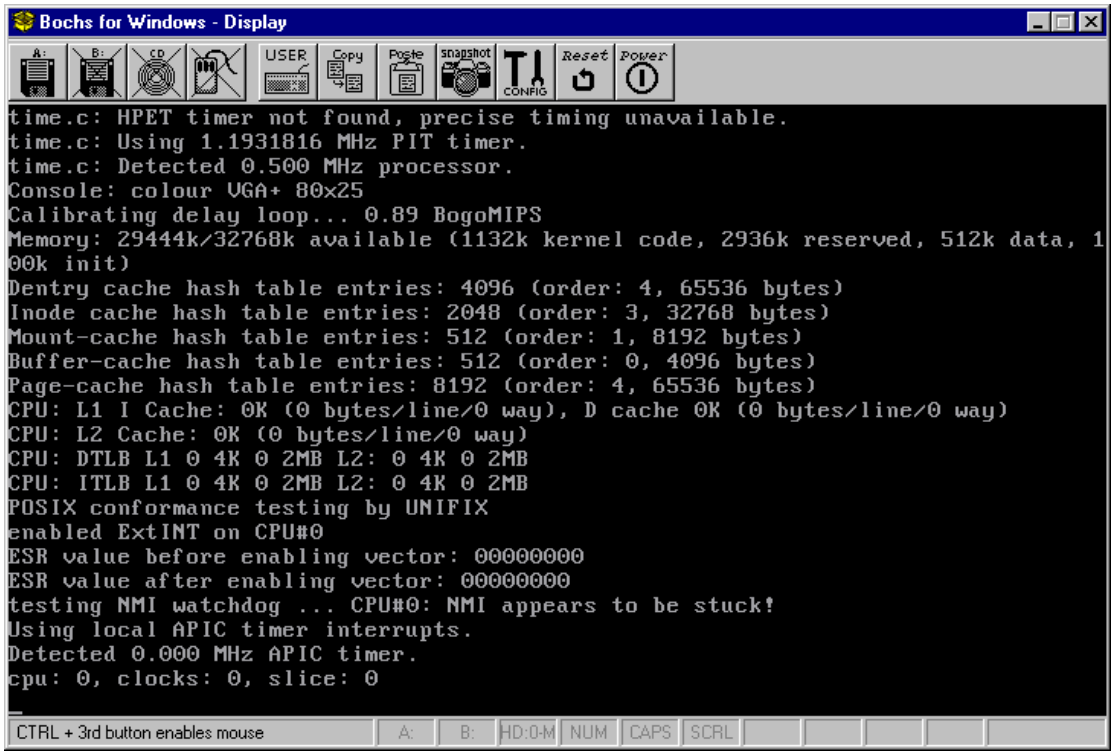


Рисунок 7 BOCHS – одна из многих виртуальных машин

3) **привязываться ни к чему нельзя**! пользователям очень не нравится когда программы привязываются к оборудованию, запрещая ее менять (а как же апгрейд?), тем более, что подобная привязка очень легко "отламывается", а если не отламывается то запускается под виртуальной машиной. к носителям информации и электронным ключам привязываться тоже нельзя — честным пользователем один геморрой (и реверанс в сторону конкурентов), нечестные все равно скопируют;
4) **не давайте хакеру явно понять, что программа еще не взломана**! выводите ругательство о взломе не сразу, а спустя некоторое время, например, через несколько дней :) или хотя бы используйте "отложенный" вызов "ругательных" процедур, посылая скрытому окну W сообщение типа "нас взломали". пусть окно W поставит его в очередь, обрабатываемую вместе с другими "нормальными" сообщениями, тогда прямая трассировка ни к чему не приведет и хакер утонет в коде;
5) **обнаружив взлом, не пытайтесь "мстить" пользователю нестабильной работой!** далеко не каждый потенциальный клиент догадается, что причина сбоев кроется в плохом краке, а не самой программе! столкнувшись с проблемами, он не побежит регистрироваться, а просто установит конкурентную программу и по своему будет очень логичен и прав;
6) **не используйте никаких недокументированных возможностей**! это ничуть не затрудняет взлом (хакеры знают все и обо всем), а вот работоспособность защищенной программы от этого сильно страдает и при установке очередного пакета обновления или при запуске под специфичной версией Windows она может отказать. так же не защищайте программу с использованием драйверов! во-первых, без многолетнего опыта очень сложно написать стабильно работающий драйвер, не завешивающий систему и не создающий новые дыры в системе безопасности, к тому же драйвера в силу их крошечного размера очень просто отломать. код, написанный на Visual Basic'e ломается не в пример сложнее;

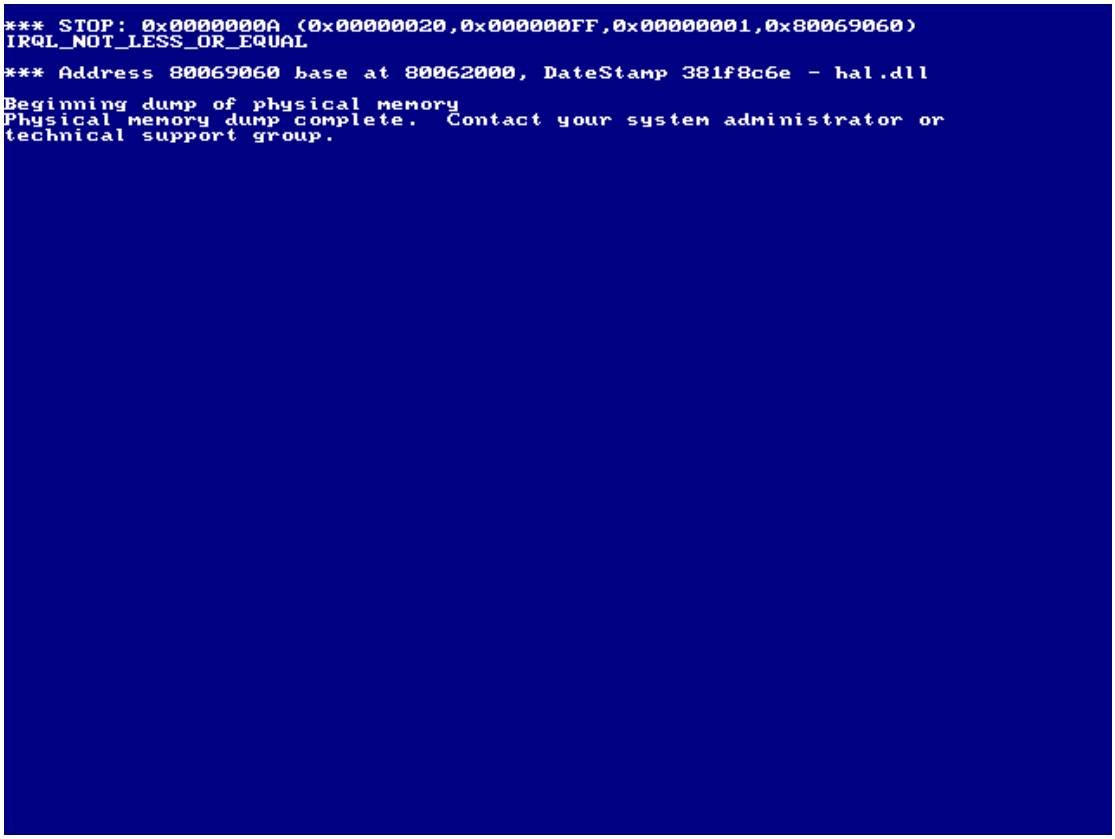


Рисунок 8 голубой экран смерти, вызванный ошибкой в драйвере защиты

7) **не используйте готовых защитных пакетов** (протекторов, упаковщиков): все готовые решения уже давно сломаны и чтобы научиться _правильно_ ими пользоваться необходимо затратить достаточно много времени, к тому же, к вашим собственным ошибкам добавляются баги протектора (а протекторов без багов не существует) и разобраться в этом глюкодроме будет очень нелегко;

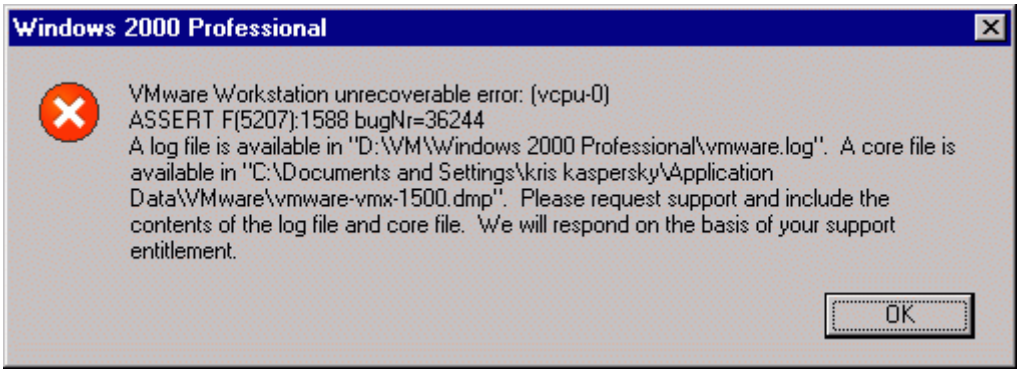


Рисунок 9 протектор, конфликтующий с VM Ware, подрывает доверие пользователей к программе

8) **никогда не доверяйте API-функциям** — их очень легко перехватить и подсунуть любой желаемый результат, тем более, не используйте прямых вызовов API-функций в программе, преимущественно использующей библиотечные функции и RTL, поскольку это сразу же демаскирует защиту, позволяя локализовать "логово дракона" во многомегабайтой чаще кода программы;
9) **ничего не проверяйте на ранней стадии инициализации**, иначе хакер доберется до защиты элементарной пошаговой трассировкой! чем позже выполняется проверка — тем лучше, причем проверке не должен предшествовать вызов никаких "очевидных" API-функций (таких, например, как CreateFile для открытия ключевого файла) — между

загрузкой ключевого файла и его проверкой должно пройти какое-то время (в смысле они должны быть разделены как можно большим объемом нелинейного кода);

10) **не защищайте программы**! все равно взломают! а если не взломают, то не купят из принципа! основной доход приносит категория честных пользователей, для которых достаточно тривиальной "защиты" из пары строк; как показывает практика, разработка более сложных защитных механизмов оказывается коммерчески неоправданной (исключение составляют специализированные программные комплексы, типа IDA PRO, PC 3000, продажи которых измеряются считанными тысячами штук). программы, ориентированные на массовый рынок, лучше распространять бесплатно, получая доход с рекламы, поддержки или других дополнительных сервисов (берите пример с Opera);

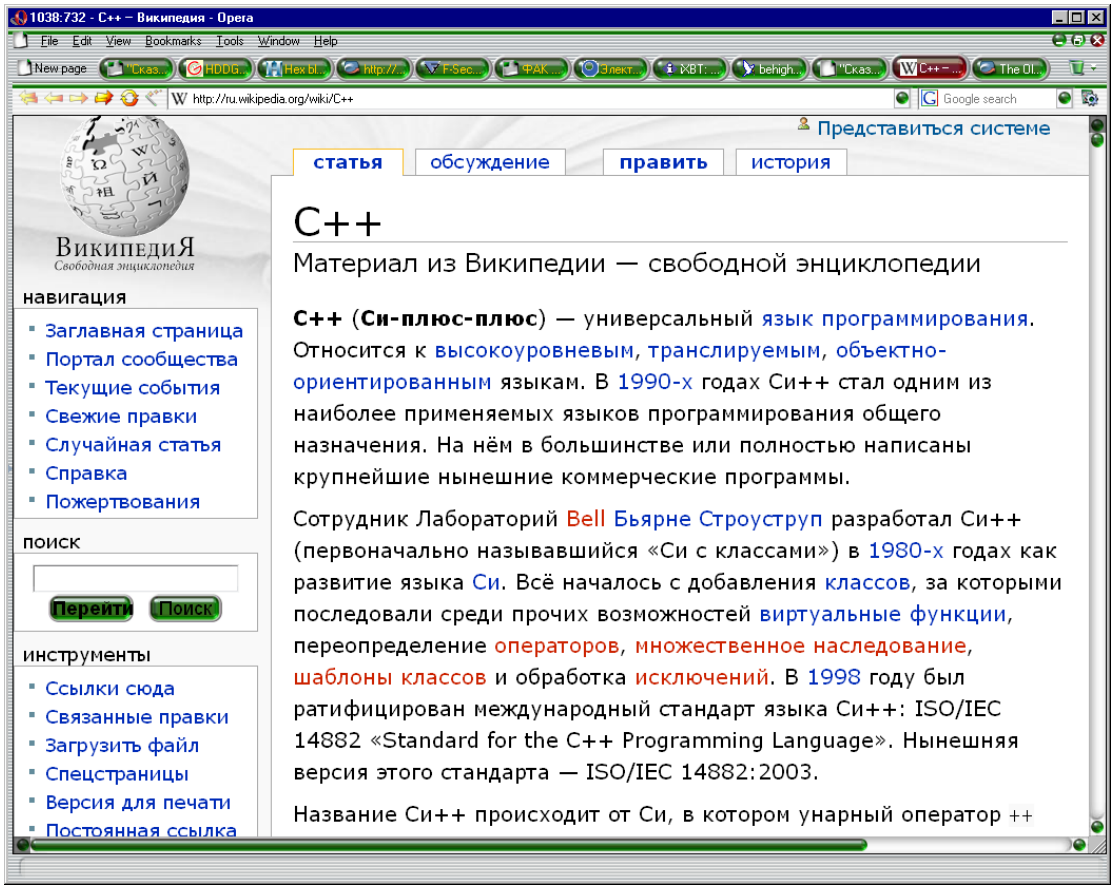


Рисунок 10 Opeara – бесплатно распространяемый браузер, извлекающий доход из сотрудничества с Google

заключение

Приведенные здесь советы в общем-то очевидны, но… подавляющее большинство программистов ими почему-то пренебрегает. А зря! Попробуйте, послушайтесь мыщъх'а — ваши защиты сразу же окрепнут, а доходы существенно возрастут, хотя совсем не в доходах дело, просто надоело барахтаться в луже непрофессионализма, в сотый раз ломая программу по одному и тому же сценарию, который надоел до слез.