противодействие дизассемблеру во сне и наяву или семь советов начинающему программисту

крис касперски ака мыщъх, no-email

чтобы выжить в этом агрессивном мире и защитить свою программу от взлома, нужно сразить отладчик и дизассемблер наповал. А как это сделать? Вот об этом мы и поговорим!

введение

Все, что можно запустить, можно и взломать. Это только вопрос времени, стимула и усилий. Против лома нет приема, а против хакера — тем более. Поэтому, защищаться надо так, чтобы простые пользователи не страдали. Использовать ненадежные приемы и навесные протекторы в стиле Extreme Protector или Armadillo недопустимо, поскольку они создают гораздо больше проблем, чем решают. Защищенная программа становится неуклюжей, тормозной, конфликтной и нестабильной. Появляются многочисленные критические ошибки приложений и голубые экраны смерти, в результате чего мы теряем клиента. Ну и кому это надо? Никаких недокументированных возможностей! Никакой привязки к операционной системе! Никаких приемов нетрадиционного программирования! Защита должна быть простой и надежной как индуистский слон! Минимум усилий, максимум эффективности!

совет N1 – шифруйтесь!

Шифровка — простой, но весьма эффективный способ борьбы с дизассемблером. Естественно, она должна быть динамической: крохотные порции кода/данных расшифровываются по мере необходимости, а после употребления зашифровываются вновь. Статические шифровальщики, расшифровывающие все тело программы за один раз, уже неактуальны. Достаточно снять дамп с работающей программы и в-у-а-л-я! Многие протекторы гробят таблицу импорта, корежат атрибуты секций, в общем пакостят по всему мясокомбинату, в результате чего снятый дамп оказывается неработоспособен, но вот для дизассемблирования он подходит вполне и такие меры защиты ни от чего не спасают!

В реализации динамического шифровщика есть множество тонкостей. Если реализовать его в виде автономной процедуры типа crypt(void *p, int N), хакер сможет расшифровать любой требуемый фрагмент простым вызовом crypt с соответствующими аргументами. Чтобы воспрепятствовать этому, различные части программы должны расшифровываться различными расшифровщиками. Некоторые "эксперты по безопасности" предлагают использовать несимметричную криптографию, полагая, что это убережет программу от модификации (то есть хака). Действительно, зашифровать модифицированную программу назад уже не получится, для этого нужно знать ключ, который хранится у разработчика и отсутствует в самой программе. Однако, ничего не стоит дописать к концу распаковщика несколько машинных команд, хачащих код налету прямо в памяти.

Шифровать можно как код, так и данные, причем, код шифруется намного сложнее. Во-первых, приходится предварительно манипулировать с атрибутами страниц, выдавая разрешение на запись, а, во-вторых, учитывать такую штуку как перемещаемые элементы — специальные ячейки памяти, в которые операционная система в процессе загрузки файла прописывает фактические адреса. Впрочем, в большинстве случаев шифровки данных оказывается вполне достаточно. Главное — не дать хакеру обнаружить в дампе текстовые строки с ругательными сообщениями (типа "trial expired"), на которые легко поставить точку останова или подсмотреть перекрестную ссылку.

Покажем как осуществляется шифровка текстовых строк на практике. Рассмотрим простейшую программу, считывающую из командной строки пароль и выводящую "password ok" или "wrong password". Один из вариантов реализации выглядит так:

```
#define _CRC_ 0x98                    // контрольная сумма пароля nezumi

main(int c, char **v)
{
        int a;
        int CRC=0;
        char *goods="password ok";
```

```
        char *wrong="wrong password";

        if (c>1)
        {
                for (a=0; a<strlen(v[1]); a++) CRC=(CRC+v[1][a]) & 0xFF;
                // для отладки (чтобы подсмотреть правильный пароль)
                // printf("%x\n",CRC);

                // проверка CRC и вывод текстовых строк на экран
                //-----------------------------------------------
                if (CRC-_CRC_) goods=wrong;printf("%s\n",goods);
                return 0;
        }
        printf("USAGE:crypt.exe password\n");
}
```

Листинг 1 поверка пароля с незашифрованными текстовыми строками

Для усиления защиты сравнивается не сам пароль, а его контрольная сумма (CRC). Эталонный пароль нигде не хранится и хакер при всем своем желании не может его подсмотреть. Оторвать мыщъх'у хвост если это не так! Но как же мы узнаем CRC эталонного пароля? Да очень просто! Достаточно внедрить в отладочную версию программы строку printf("%x",CRC), распечатывающую контрольную сумму введенного пароля, и ввести эталонный пароль. Например, CRC слова "nezumi" равна 98h.

Откомпилируем полученную программу (естественно, предварительно убрав отладочную печать из финальной версии) и пропустим ее через дизассемблер:

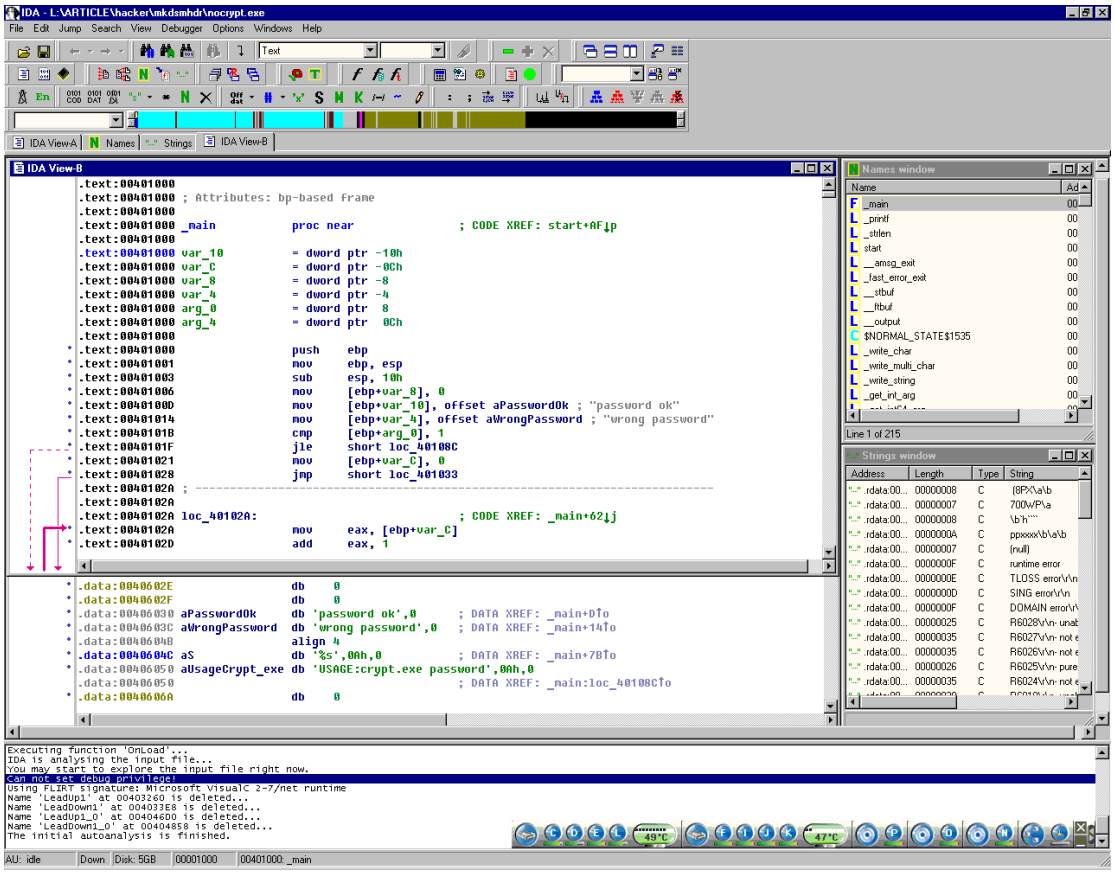

Рисунок 1 текстовые строки, хранящиеся открытым текстом, и перекрестные ссылки, ведущие к ним

Текстовые строки "password ok" и "wrong password" хранятся открытым текстом и легко обнаруживаются даже при беглом просмотре листинга (обычно для этого используются программы-фильтры, отсеивающие все читабельные текстовые последовательности). Что сделает хакер? Установив точку останова на начало "wrong password", он легко перехватит код, выводящий эту строку на экран, после чего ему останется найти тот условный переход, который его выводит. Весь взлом не займет и десяти минут, а осуществить его сможет даже ребенок!

Поэтому, текстовые строки необходимо зашифровать, расшифровывая их непосредственно перед выводом на экран. Это можно сделать, например, так:

```
#define _CRC_ 0x98                    // контрольная сумма пароля nezumi
#define _KEY_ 0xFF                    // ключ шифрования

main(int c, char **v)
{
        int a;
        int CRC=0;
        char buf[1024];

        // зашифрованные текстовые строки
        char *goods="\x8F\x9E\x8C\x8C\x88\x90\x8D\x9B\xDF\x90\x94"; //"password ok";
        char *wrong="\x88\x8D\x90\x91\x98\xDF\x8F\x9E\x8C\x8C\x88\x90\x8D\x9B"; //wrong

        if (c>1)
        {
                for (a=0; a<strlen(v[1]); a++) CRC=(CRC+v[1][a]) & 0xFF;

                // проверка CRC и расшифровка текстовых строк
                //----------------------------------------
                if (CRC-_CRC_) // пароль ок
                        for (a=0;a<strlen(wrong);a++) buf[a]=wrong[a]^_KEY_;
                else           // пароль не ок
                        for (a=0;a<strlen(goods);a++) buf[a]=~goods[a];

                // формирование завершающего нуля и вывод строки на экран
                buf[a]=0; printf("%s\n",buf);
                return 0;
        }
        printf("USAGE:crypt.exe password\n");
}
```

Листинг 2 проверка пароля с зашифрованными строками

Обратим внимание, что строки расшифровываются не по месту хранения, а помещаются в специальный буфер, чтобы затруднить взлом. В противном случае, хакер сможет установить точку останова на функцию printf (которая эту строку и выводит), что позволит ему определить смещение строки в секции данных, и проанализировать перекрестные ссылки которые ведут к защитному коду.

Для шифровки необязательно использовать криптостойкие алгоритмы, такие как RC4 или DES, сойдет и обычный XOR. Необходимо только убедиться, что ни один символ шифруемой строки не обращается в ноль, ведь Си трактует ноль как завершитель строки. Выражение x XOR y == 0 становится истинным тогда и только тогда, когда x == y, т. е. шифоключ совпадает с одним из символов шифруемой строки. Значение FFh ни разу ни встречается ни в одной из двух наших строк, поэтому это подходящий ключ, но лучше использовать несколько независимых шифровщиков, чтобы было сложнее ломать.

Единственная проблема — как зашифровать строки? В принципе, это можно сделать и после компиляции, воспользовавшись любым hex-редактором (например, hiew'ом), однако, при каждом ребилде эту процедуру придется повторять вновь и вновь, что очень за… ну, в смысле, достает.

Мы напишем для этой цели специальный шифратор, захватывающий исходную строку и выплевывающий зашифрованную последовательность, оформленную по всем правилам языка Си, после чего нам останется только вставить ее в исходный код.

Макет шифратора может выглядеть так:

```
main(int c, char **v)
{
        int a;

        printf("char *var_name=\"");
        for(a=0;a<strlen(v[1]);a++)          // шифровка по XOR
                printf("\\x%02X",v[1][a] ^ atol(v[2])); printf("\";");
}
```

Листинг 3 макет простейшего шифратора

Текстовые строки "password ok/wrong password" волшебным образом исчезают из откомпилированной программы (см. рис 2)! Теперь, для взлома защиты хакеру придется

потратить намного больше времени и усилий. В данном случае, разница не так уж заметна, но в программах, состоящих из десятков тысяч строк, все будет торчком!

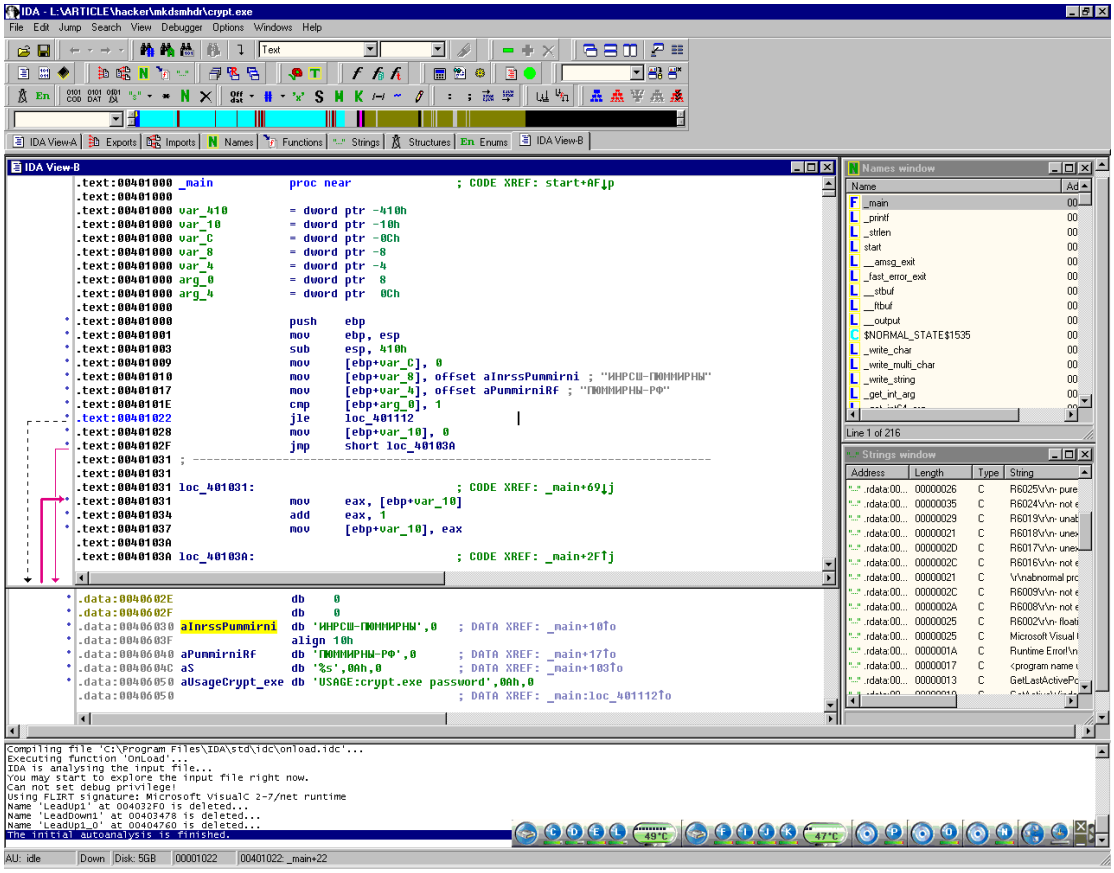

Рисунок 2 зашифрованные строки уже не бросаются в глаза

совет N2 – не давайте переменным говорящих имен

Ни в коем случае не назначайте защитным компонентам никаких осмысленных имен, особенно при программировании в DELPHI и BUILDER, поскольку, они попадают в исполняемый файл. Вроде бы очевидный совет (меня даже высмеяли за него пару раз), но сколько программистов в него вляпывается!

Вот так, например, выглядит результат декомпиляции программы Etlin HTTP Proxy:

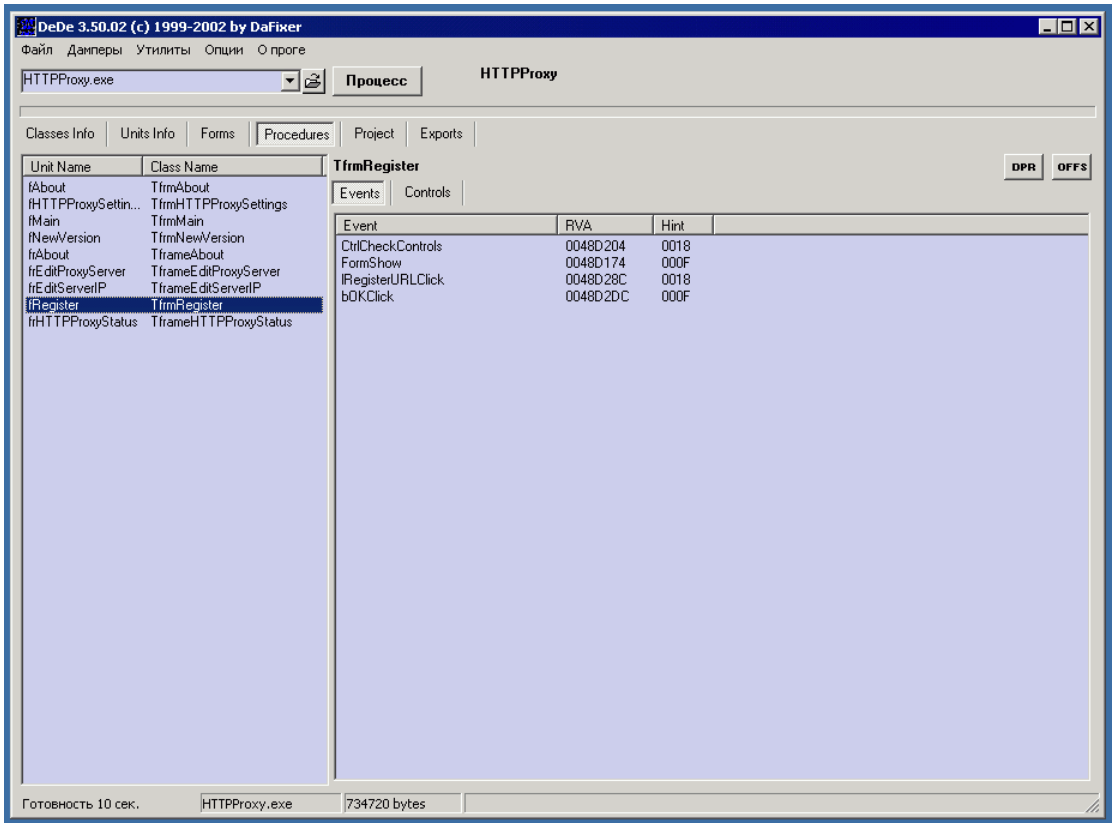

Рисунок 3 декомилятор DEDE, исследующий программу Etlin HTTP Proxy

Хакер с ходу видит юнит fRegister с процедурой bOkClick, обрабатывающей нажатие кнопки "OK" и расположенной по адресу 48D2DCh. Все! Защитный механизм успешно локализован! Самая сложная часть взлома позади!

Просто поразительно сколько информации можно извлечь, просматривая текстовые строки, открытом текстом лежащие в рядовой программе. Если бы программисты использовали бессмысленную абракадабру, взлом занимал бы гораздо больше времени!

совет N3 — используйте виртуальные функции

Активное использование виртуальных функций существенно затрудняет дизассемблирование. Виртуальные функции вызываются по косвенным ссылкам и над вычислением эффективных адресов приходится основательно потрудится.

Вот пример:

```
class Base{
public: virtual void demo_1(void){printf("BASE DEMO 2\n");};
        virtual void demo_2(void) = 0;};

class Derived:public Base{
public: virtual void demo_1(void){printf("DERIVED\n");};
        virtual void demo_2(void){printf("DERIVED DEMO 2\n");};
};

main(){Base *p = new Derived; p->demo(); p->demo_2();printf("non-virtual\n");}
```

Листинг 4 виртуальные и не виртуальные функции

А вот его дизассемблерный фрагмент:

```
.text:0040101B        mov     eax, [esi]
.text:0040101D        mov     ecx, esi
.text:0040101F        call    dword ptr [eax]       ; вызов виртуальной функции 1
.text:00401021        mov     edx, [esi]
.text:00401023        mov     ecx, esi
.text:00401025        call    dword ptr [edx+4]     ; вызов виртуальной функции 2
.text:00401028        push    offset aNonvirtual;
```

```
.text:0040102D        call    sub_4010B0            ; вызов не виртуальной функции
```

Листинг 5 вызов виртуальных и не виртуальных функций

Что мы видим? Вызов не виртуальной функции осуществляется по непосредственному значению (константе), равной в данном случае 4010B0h. А вот с виртуальными функциями все намного сложнее. Команда call dword prt [eax], передает управление по адресу, хранящемуся в двойном слове, на которое указывает регистр EAX, который в свою очередь загружается из двойного слова, на которое указывает регистр ESI, а сам ESI… И такие цепочки могут продолжаться долго, очень долго, причем исходный указатель инициализируется совсем в другом месте (как правило пра-пра-пра-материнской функции). В общем, с виртуальными функциями современные дизассемблеры еще не справляются (к DELHI и BUILDER это не относится, для них существует множество отличных декомпиляторов).

Только необходимо помнить, что оптимизирующие компиляторы при первой же возможности заменят виртуальную функцию на статическую. Просто объявить функцию как виртуальную недостаточно, нужно *использовать* ее как виртуальную.

совет N4 – ослепляйте FLIRT

Типичная программа наполовину состоит из библиотечных функций, анализ которых занимает огромное количество времени и усилий (особенно, если это интерфейсный компонент какой). IDA PRO поддерживает шикарную технологию FLIRT (Fast Library Identification and Recognition Technology), автоматически распознающую функции большинства популярных библиотек, в результате чего задача хакера существенно упрощается.

Вот фрагмент защитного механизма, считывающий имя пользователя и серийный номер из окна редактирования функцией TControl::GetText:

```
CODE:0048D2F7  mov     eax, [ebx+328h]
CODE:0048D2FD  call    @TControl@GetText$qqrv ; TControl::GetText(void)
...
CODE:0048D309  mov     eax, [ebx+320h]
CODE:0048D30F  call    @TControl@GetText$qqrv ; TControl::GetText(void)
```

Листинг 6 имена библиотечных функций, автоматические распознанные ИДОЙ

Размер защитного кода несопоставим с размером библиотечных функций, которые он использует (библиотечные функции на порядок "жирнее"). Если бы не FLIRT, взлом затянулся бы надолго. А так пришел, увидел, отломил. Тем более, что большинство хакеров предпочитает ставить точки останова не на GetWindowTextA, а на @TControl@GetText$qqrv, что намного удобнее. Как этому помешать? Оказывается, чтобы ослепить ИДУ достаточно изменить всего несколько байтов в начале каждой библиотечной функции.

Вот, например, @TControl@GetText$qqrv:

```
CODE:004410B8  push    ebx
CODE:004410B9  push    esi
CODE:004410BA  push    edi
...
CODE:004410E3  pop     edi
CODE:004410E4  pop     esi
CODE:004410E5  pop     ebx
CODE:004410E6  retn
```

Листинг 7 фрагмент дизассемблерного листинга библиотечной функции @TControl@GetText$qqrv

Если заменить push ebx/push esi//pop esi/pop ebx на push esi/push ebx//pop ebx/pop esi, IDA не сможет узнать эту функцию и хакеру придется основательно попыхтеть над реконструкцией алгоритма защитного механизма.

```
CODE:0048D2F7  mov     eax, [ebx+328h]
CODE:0048D2FD  call    sub_4410B8
...
CODE:0048D309  mov     eax, [ebx+320h]
CODE:0048D30F  call    sub_4410B8
```

Листинг 8 исправленные функции уже не опознаются ИДОЙ

Конечно, править все функции вручную — медленный и крайне неперспективный пусть, однако, знатоки ассемблера за несколько вечеров напишут автоматический пермутатор или возьмут уже готовый (благо полиморфных движков в сети предостаточно). Тем более, что это достаточно сделать всего один раз — в библиотеке, а потом просто линковать эту библиотеку к защищенным программам и все! Кстати говоря, исходные тексты большинства стандартных библиотек открыты, а это значит, что перекомпилировав их с другими ключами оптимизации (или другим компилятором) мы ослепим FLIRT на все 100%

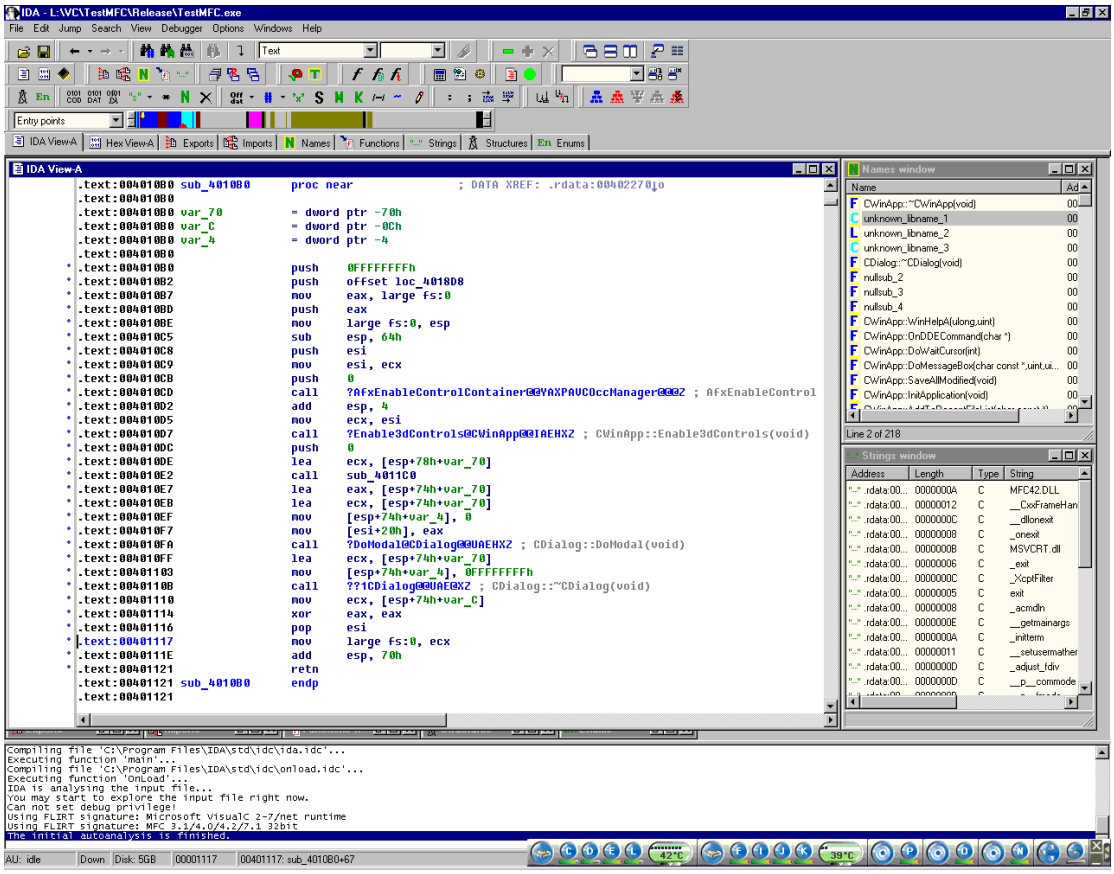

Рисунок 4 MFC-приложение, использующее готовые библиотеки

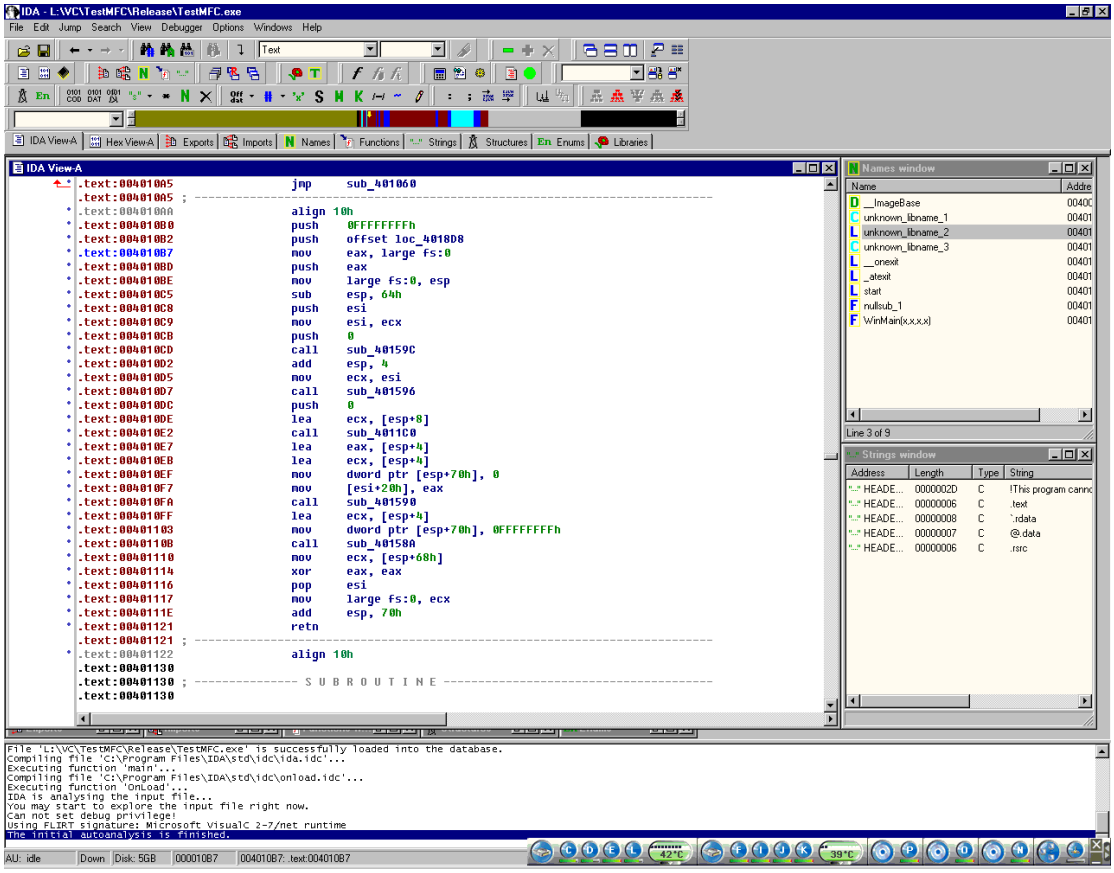

Рисунок 5 тоже самое приложение, использующие перекомпилированные MFC-библиотеки

совет N5 — программируйте на Visual Basic и Forth

Как это ни смешно, но программы, написанные на Visual Basic'e ломаются значительно труднее (особенно, если использовать трансляцию в p-код). Еще сложнее ломается Forth. Кто сталкивался — той поймет. Кто не сталкивался — еще не в психушке. Forth – это вообще-то интерпретатор, но довольно своеобразный, совсем не похожий на остальные языки. Чисто технически можно написать Forth-декомпилятор, или найти уже готовый, но для этого хакеру потребуется изучить и сам Forth, а это, значит, что взлом программы растянется надолго. Разумеется, никто не предлагает писать приложение на Forth'e целиком. Достаточно запрограммировать на нем несколько ключевых защитных процедур (генерация серийного номера, расшифровщик и т. д.)

Еще можно использовать Microsoft Visual Studio .NET — она тоже позволяет генерировать p-код, правда для него уже появилось множество декомпиляторов, да и сам формат p-кода стандартизован и тщательно специфицирован. Лучше откопать какой-нибудь редкоземельный интерпретатор, например, Haskel или LISP. Во-первых, знакомство с новыми языками расширяет кругозор, а, во-вторых, кругозор большинства начинающих хакеров не выходит за пределы C/Pascal/ASM и тот же LISP они с ходу не взломают. Скорее всего, исследуемая программа будет заброшена на полку до лучших времен (т. е. навсегда).

```
: IS 0 DO I . LOOP ;
: AS 0 DO CR 1 - DUP IS LOOP ;
```

Листинг 9 простейшая программа на языке Forth…

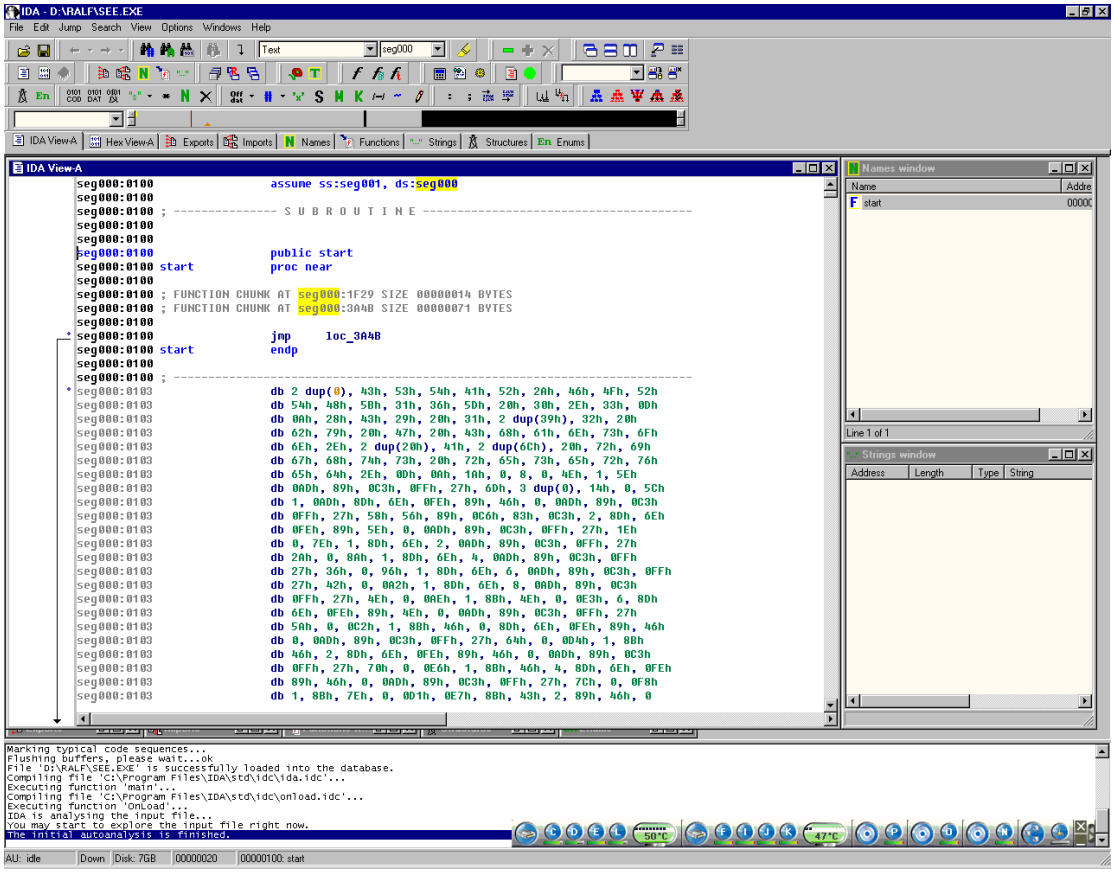

Рисунок 6 …и ее дизассемблерный текст

совет N6 – используйте глобальные переменные

Большинство руководств по программированию имеют явное предубеждение насчет глобальных переменных и рассказывают множество ужасных историй, как пара придурков гоняла бага целый день (месяц, неделю), а все из-за глобальных переменных! Это верно! Обращение к глобальным переменным может происходить из разных мест и кто угодно может затереть чужое значение со всеми отсюда вытекающими последствиями. Локальные переменные в этом отношении намного нагляднее и проще. Их любят все — и программисты, и хакеры.

С другой стороны, флаг регистрации, расположенной в глобальной переменной, ломается на ура, поскольку к нему ведут перекрестные ссылки, показывающие какой код к нему обращается и когда (см. рис 7). Как затруднить взлом? А вот как — использовать одну и ту же ячейку памяти для хранения нескольких типов данных сразу.

Допустим, на стадии инициализации некоторая ячейка используется как флаг регистрации, затем флаг регистрации временно копируется в другое место, а сюда помещается флаг ошибок, например. Через некоторое время процедура обмена повторяется вновь… и вновь! Конечно, это упрощенная схема, но общий смысл, думаю, понятен.

Хакер будет видеть множество перекрестных ссылок, ведущих в разные части программы. Проанализировав несколько из них, он, быстро найдет флаг ошибок, переименует переменную в f_error и тут же потеряет к ней всякий интерес. А если не потеряет, ему будет очень трудно разобраться в какой момент эта переменная флаг ошибок, а в какой — флаг регистрации.

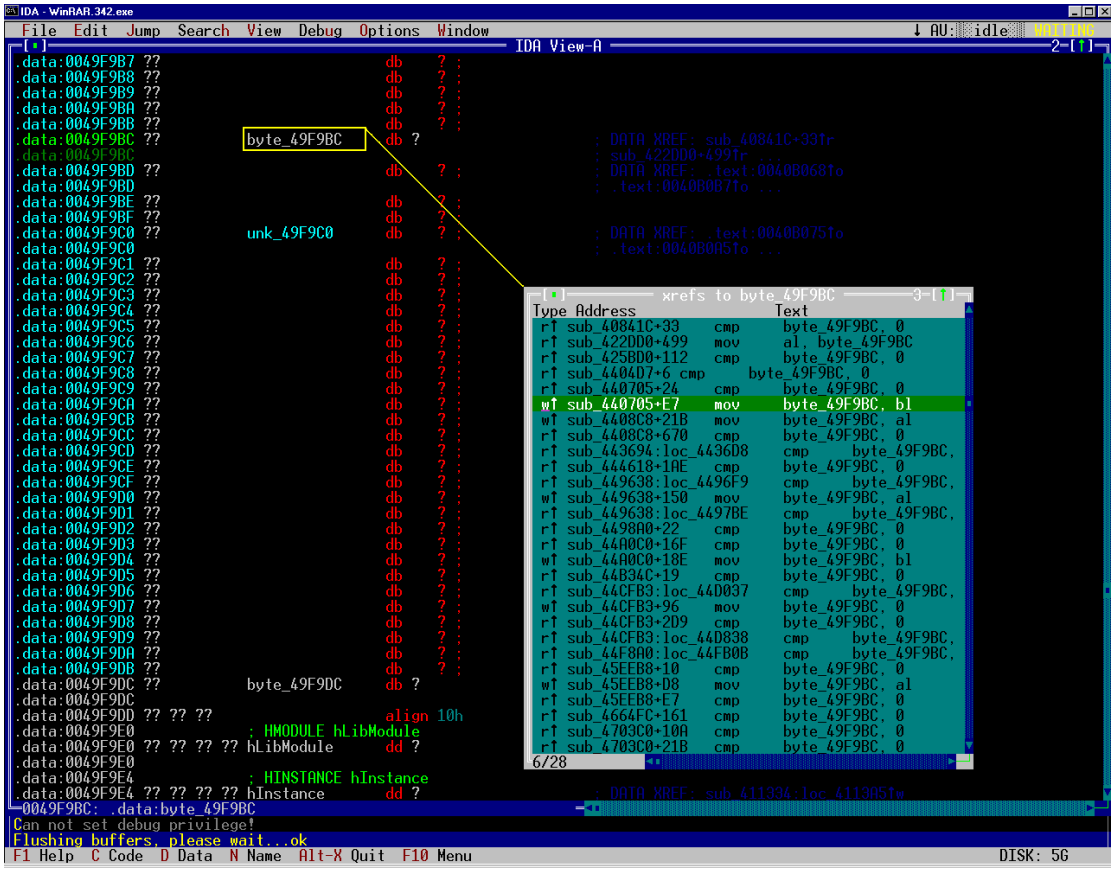

Рисунок 7 WinRAR — флаг регистрации и перекрестные ссылки, ведущие к нему

заключение

Рассмотренные приемы, не смотря на свою простоту, служат отличным оружием против хакеров. Конечно, взломать можно все, но… только со временем. А времени хакерам всегда недостает. В первую очередь ломаются простые программы. Сложные оставляются на потом. К тому же многие хакерские группы состязаются в количестве взломанных программ (причем, в расчет берется именно количество, а не сложность взлома) и хитроумные защиты портят им всю малину, в смысле — сбивают рейтинг. В общем, защититься от хакеров вполне реально, и теперь вы знаете как.