техника отладки приложений без исходных кодов

крис касперски ака мыщъх, no-email

практически все знают, что программы взламываются отладчиком, но не все знают как. на самом деле, ничего сложного в этом нет. достаточно выучить несколько простых приемов и уже можно начинать ломать.

введение в отладку

Отладчик — невероятно мощный инструмент в руках взломщика, однако, к нему нужен свой подход. Большинство начинающих хакеров начинает отлаживать программу с точки входа и… умирает в цикле выборки сообщений. Пошаговое исполнение программы (так же называемое трассировкой) слишком трудоемкий и крайне неэффективный процесс. Событийно-ориентированные приложения, (а к ним относятся практически все Windows-приложения), так не отлаживаются. Допустим, мы трассируем MFC-приложение, доходим до вызова AfxWinMain и… оказываемся глубоко внутри MFC42.DLL, откуда уже и вызывается весь пользовательский код, но прежде чем трассировка доберется до него, мы успеем состариться!

Но ведь отлаживать всю программу целиком совершенно необязательно! Опытные хакеры трассируют только отдельные части защитного кода. Но как же мы найдем его в миллионах машинных инструкций исполняемого файла? Существует множество методик: точки останова, раскрутка стека, перекрестные ссылки, условная трассировка, прямой поиск паролей/серийных номеров в памяти и т. д. Расскажем обо всем этом поподробнее.

Мы будем курочить программу Drive LED от компании O&O Software, 30-дневую демонстрационную версию которой можно скачать с сайта: http://www.oo-software.com/en/download/index.shtml;

дизассемблер и отладчик в одной упряжке

Дизассемблер содержится в каждом отладчике (мы же ведь не собираемся отлаживать программу непосредственно в машинном коде, верно?), но тот дизассемблер, что находится внутри soft-ice или OllyDbg, слишком примитивен и ненагляден. IDA PRO намного мощнее. Она автоматически распознает имена библиотечных функций, определяет типы локальных переменных и делает множество других полезных вещей, в частности, позволяет комментировать листинг и назначать символьные метки для инструкций и данных. Исследовать защищенные программы с ее помощью — настоящее удовольствие. Однако, вызовы типа call [ebx+64h] приводят хакеров в бешенство, особенно если функция вызывается из различных мест с различным ebx. На выяснение значения ebx в дизассемблере можно ухлопать целый день, а в отладчике просто "подсмотрел" его и все!

Или вот вызывается что-то по адресу 77E92B8D, лежащим где-то внутри операционной системы (при дизассемблировании дампов памяти такие адреса встречаются сплошь и рядом). В отладчике достаточно просто дать команду "u 77E92B8D" и мы тут же увидим, что это CreateFileA.

Бессмысленно спорить, кто круче отладчик или дизассемблер. Эти инструменты взаимно дополняют друг друга. Реконструкцию алгоритмов лучше поручить дизассемблеру, а все непонятные места уточнять в отладчике.

Загрузка символов в отладчик осуществляется довольно противоестественным образом, на котором спотыкаются многие начинающие.

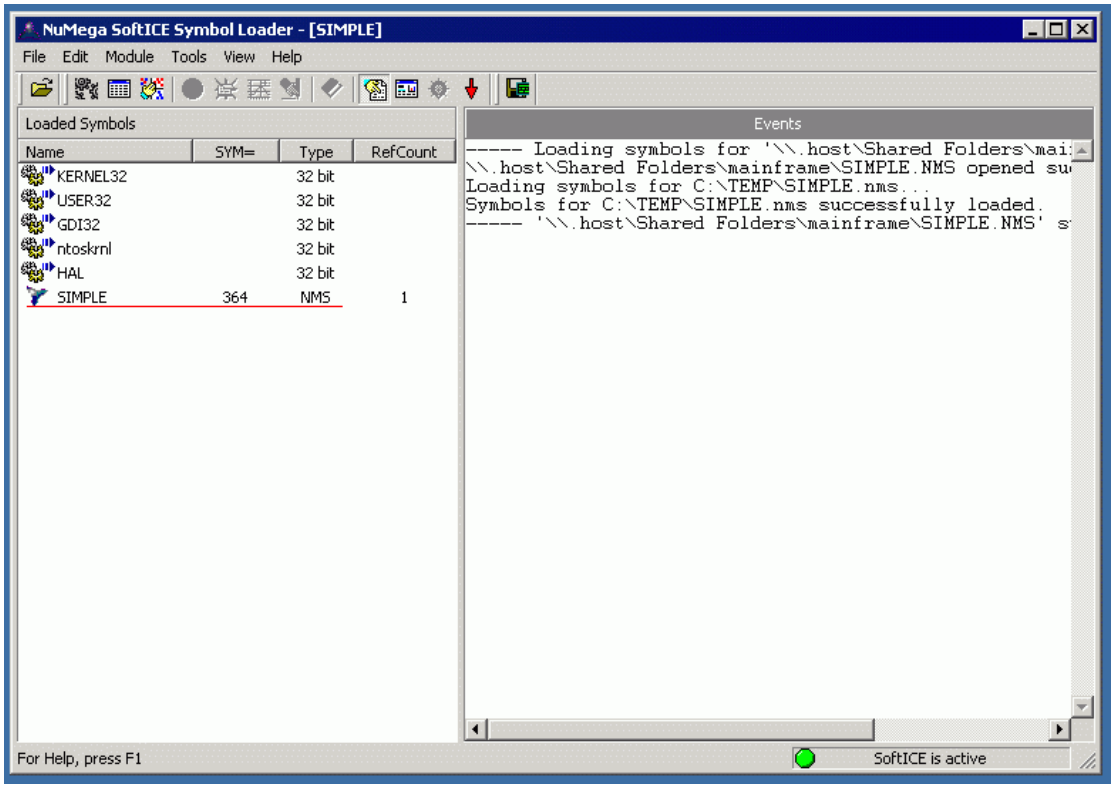

Рисунок 1 загрузка символьной информации в loader32

Сначала исследуемый файл пропускается через ИДУ. Затем, в меню "File" выбирается пункт "Produce output file → Produce MAP file" (причем, имя MAP-файла должно совпадать с именем самого дизассемблируемого файла). В появившемся диалоговом окне взводим все три галочки: Segmentation information (информация о сегментах), Autogenerated names (автогенерируемые имена) и Demangle names (размагленные имена). Полученный MAP-файл скармливается утилите idasym (которую можно скачать с сайта www.idapro.com) и конвертируется в sym-формат. Под воздействием утилиты nmsym, входящий в комплект поставки soft-ice, sym-файл преобразуется в nms. Уф! Половина работы сделана, теперь, пока запускается NuMega Symbol Loader можно и передохнуть. В меню File выбираем пункт Open, открываем nms-файл и говорим Module → Load (загрузить). Появившаяся надпись "Symbols for C:\TEMP\SIMPLE.NMS successfully loaded" говорит о том, что все прошло успешно. Теперь можно открыть и сам исполняемый файл (File → Open и Module → Load).

Сравните, как выглядит экран отладчика с символами и без:

```
EAX=08930005   EBX=7FFDF000   ECX=7FFDF000   EDX=00000893   ESI=00000000
EDI=00000000   EBP=0012FFC0   ESP=0012FF94   EIP=00401291   o d I s z a P c
CS=001B   DS=0023   SS=0023   ES=0023   FS=0038   GS=0000
──────────────────────────────────────────────────────────────────────PROT32─
001B:00401025  PUSH      00406030
001B:0040102A  CALL      00401234
001B:0040102F  ADD       ESP,04
001B:00401032  PUSH      00406070
001B:00401037  PUSH      64
001B:00401039  LEA       ECX,[EBP-68]
001B:0040103C  PUSH      ECX
001B:0040103D  CALL      00401124
001B:00401042  ADD       ESP,0C
001B:00401045  PUSH      00406040
001B:0040104A  LEA       EDX,[EBP-68]
001B:0040104D  PUSH      EDX
001B:0040104E  CALL      004010A0
001B:00401053  ADD       ESP,08
001B:00401056  TEST      EAX,EAX
001B:00401058  JZ        00401069
001B:0040105A  PUSH      00406050
(PASSIVE)-KTEB(85050760)-TID(0078)──simple!.text+0025──────────────────────
NTICE: Load32 START=77E10000  SIZE=65000  KPEB=85050020  MOD=user32
NTICE: Load32 START=77F40000  SIZE=3C000  KPEB=85050020  MOD=gdi32
NTICE: Load32 START=77DB0000  SIZE=5A000  KPEB=85050020  MOD=advapi32
NTICE: Load32 START=77D40000  SIZE=6F000  KPEB=85050020  MOD=rpcrt4
NTICE: Load32 START=78000000  SIZE=46000  KPEB=85050020  MOD=msvcrt
NTICE: Load32 START=400000  SIZE=8000  KPEB=850509E0  MOD=simple
NTICE: Load32 START=77F80000  SIZE=7A000  KPEB=850509E0  MOD=ntdll
NTICE: Load32 START=77E80000  SIZE=B7000  KPEB=850509E0  MOD=kernel32
Break due to BPX KERNEL32!GetVersion  (ET=964.26 milliseconds)
:p ret
:u 401000
:
     Enter a command (H for help)                                  simple
```

Рисунок 2 отладка файла без символьной информации

```
EAX=00300A60   EBX=7FFDF000   ECX=004060E0   EDX=00000003   ESI=00000000
EDI=00000002   EBP=0012FFC0   ESP=0012FF84   EIP=00401000   o d I s z a P c
CS=001B   DS=0023   SS=0023   ES=0023   FS=0038   GS=0000
────loc_401025──────────────────────────────────────────────────────PROT32─
loc_401025
001B:00401025  PUSH      aEnterPassword
001B:0040102A  CALL      _printf
001B:0040102F  ADD       ESP,04
001B:00401032  PUSH      off_406070
001B:00401037  PUSH      64
001B:00401039  LEA       ECX,[EBP-68]
001B:0040103C  PUSH      ECX
001B:0040103D  CALL      _fgets
001B:00401042  ADD       ESP,0C
001B:00401045  PUSH      aMygoodpassword
001B:0040104A  LEA       EDX,[EBP-68]
001B:0040104D  PUSH      EDX
001B:0040104E  CALL      _strcmp
001B:00401053  ADD       ESP,08
001B:00401056  TEST      EAX,EAX
001B:00401058  JZ        loc_401069
(PASSIVE)-KTEB(811F0DA0)-TID(0304)──simple!.text+0025──────────────────────
      001B:004048B8 LeadDown1_0
      001B:00404318 LeadDown2
      001B:004048D8 LeadDown2_0
      001B:00404340 LeadDown3
      001B:00404900 LeadDown3_0
      001B:0040419C LeadUp2
      001B:0040475C LeadUp2_0
      001B:004041C0 LeadUp3
      001B:00404780 LeadUp3_0
      001B:00404F60 RtlUnwind
:u 401000
:
     Enter a command (H for help)                                  simple
```

Рисунок 3 отладка файла с символьной информацией, автоматически сгенерированной IDA PRO

Без символьной информации назначение функций 401234h и 401124h совсем не очевидно и на них отладку можно угробить несколько часов лучших лет своей жизни, а с символами все ясно и так. К тому же, символьные имена можно использовать в точках останова, например: "bpx _fgets" (установить точку останова на функцию чтения пароля) или "bmp aMygoodpassword" (установить точку останова на код, обращающийся к эталонному паролю).

точки останова на API-функции

Точки останова (они же break point'ы или просто бряки) — основное оружие хакера в борьбе с защитными механизмами. Наибольшей популярностью пользуются точки останова на API-функции. Чтение содержимого окна часто (но не всегда) осуществляется API-функцией GetWindowTextA, открытие файла — CreateFileA, загрузка динамической библиотеки — LoadLibraryA и т. д. Установка точки останова на эти функции заставить отладчик "всплывать" всякий раз, когда защита пытается сделать что-то нехорошее. Этим она демаскирует свое расположение в исследуемом коде, выводя на след.

Проблема в том, что API-функций очень много и угадать каким именно способом защита манипулирует с окном, не так-то просто. Обычно используется либо "тупой" перебор всех возможных API-функций одна за другой, либо API-шпионы, показывание что происходит под капотом отлаживаемой программы (см. статью "взлом архиватора WinRAR").

Для установки точки останова на API-функцию достаточно нажать <Ctrl-D>, и дождавшись появления отладчика на экране, написать "bpx имя_функции–". В soft-ice точки останова носят глобальный характер. Если мы устанавливаем бряк на функцию CreateFileA, отладчик всплывает при каждом открытии/создании какого бы то ни было файла. Вот радость! Чтобы ограничить пыл отладчика, необходимо использовать условные точки останова. Допустим, мы хотим всплывать на "keyfile.key". Открываем MSDN и смотрим прототип функции CreateFile. Видим, что lpFileName передается в крайнем левом аргументе, а поскольку аргументы API функций заносятся в стек справа налево, указатель на имя открываемого файла окажется на вершине стека и выше него будет только адрес возврата.

Таким образом, в момент вызова CrateFile, lpFileName будет лежать по смещению 4, относительно ESP и условная точка останова будет выглядеть так: "`bpx CreateFileA if (*(esp->4)=='keyf')`". Имя файла, заключенное в кавычки, автоматически преобразуется отладчиком в 32-разроядную константу и потому его длина не должна превышать 4х байт, причем, отладчик чувствителен к регистру ('keyf' и 'Keyf' для него не одно и тоже), а вот файловая система — нет. В большинстве случаев частичного сравнения имени оказывается вполне достаточно. Если же нет, можно прибегнуть к оператору AND и сравнивать несколько 4х битных подстрок за раз. Синтаксис условных точек останова подробно описан в документации на soft-ice, так что не будем на этом останавливаться.

Многие защиты противостоят точкам останова, например, начиная выполнение API-функции не с первого байта, и в таких случаях приходится прибегать к установке бряков на native-API — своеобразному фундаменту операционной системы, ниже которого находятся только порты ввода/вывода и драйвера. Описание native-API функций можно найти в Interrupt List'e Ralf'a Brown'a или "The Undocumented Functions Microsoft Windows NT/2000" от Tomas'a Nowak'a. В частности, NtCreatrFile используется для создания/открытия файлов.

Отладчик OllyDbg поддерживает намного более мощный механизм условных точек, позволяющий отслеживать практически любые ситуации. Например, EAX == "mypswd" — всплывать, когда регистр EAX указывает на строку с паролем/серийным номером который мы ввели при регистрации. Это универсальный способ взлома, подходящий практически ко всем защитам. Каким бы образом программа не извлекала содержимое окна редактирования, в какой-то момент она неизбежно засунет указатель в регистр. Вот тут-то мы и всплывем! Процедура проверки соответствия пароля будет где-то неподалеку. Конечно, этим регистром не обязательно должен быть EAX. Вполне вероятно, что компилятор задействует EBX, ESI или что-то еще. Документация на OllyDbg заявляет о поддержке выражения R32 == "mypswd", где R32 – любой регистр общего назначения, однако, в текущих версиях отладчика эта конструкция не работает и все регистры приходится перебирать вручную (благо, можно написать свой плагин, автоматизирующий этот процесс).

Поимо точек останова на API, можно брякать библиотечные функции. В приложениях, написанных на DELPHI/BUILDER/MFC/Visual BASIC прямые вызовы API используются редко. И хотя никакое дело без API-функций, конечно же не обходится, их анализ мало что дает, особенно, если используется динамический обмен данных с окном (DDX) и другие

навороченные технологии, обмазывающие API-функциями несколькими мегабайтами кривого кода. Это же сдохнуть можно! Но мы не будем! Библиотечные функции легко опознаются ИДОЙ и брякаются как обычные API-функции только с той разницей, что точка останова носит локальный характер, воздействующий только на отлаживаемое приложение. А это значит, что после нажатия <Ctrl-D> мы должны переключить контекст управления, чтобы попасть в адресное пространство отлаживаемого приложения. Это осуществляется либо командой "ADDR имя_процесса", либо установкой точки останова на любую API-функцию, вызываемую отлаживаемым приложением. Например, SendMessageA. Жмем, <Ctrl-D>, пишем "bpx MeggsageBoxA", выходим из sof-ice, дожидаемся пока он всплывет (если не всплывает, можно дернуть мышью или щелкнуть по отлаживаемому окну), если в правом нижнем углу отладчика находится имя нашего процесса — все ок, в противном случае выходим из отладчика и ждем его всплытия опять.

точки останова на сообщения

Допустим, у нас есть окно с несколькими элементами управления (меню, флажок или кнопка) нажатия на которые мы хотим отследить (см. рис. 1). Как это сделать? Очень просто! Установить точку останова на сообщение! В Windows весь интерфейс построен на сообщениях (об этом хорошо написал Петзолд в "Программировании для Windows 95"). В частности, при нажатии на элемент управления (или изменении окна редактирования) окну посылается сообщение WM_COMMAND. Вот на него-то мы и поставим точку останова, но прежде определим дескрпитор (handle) окна.

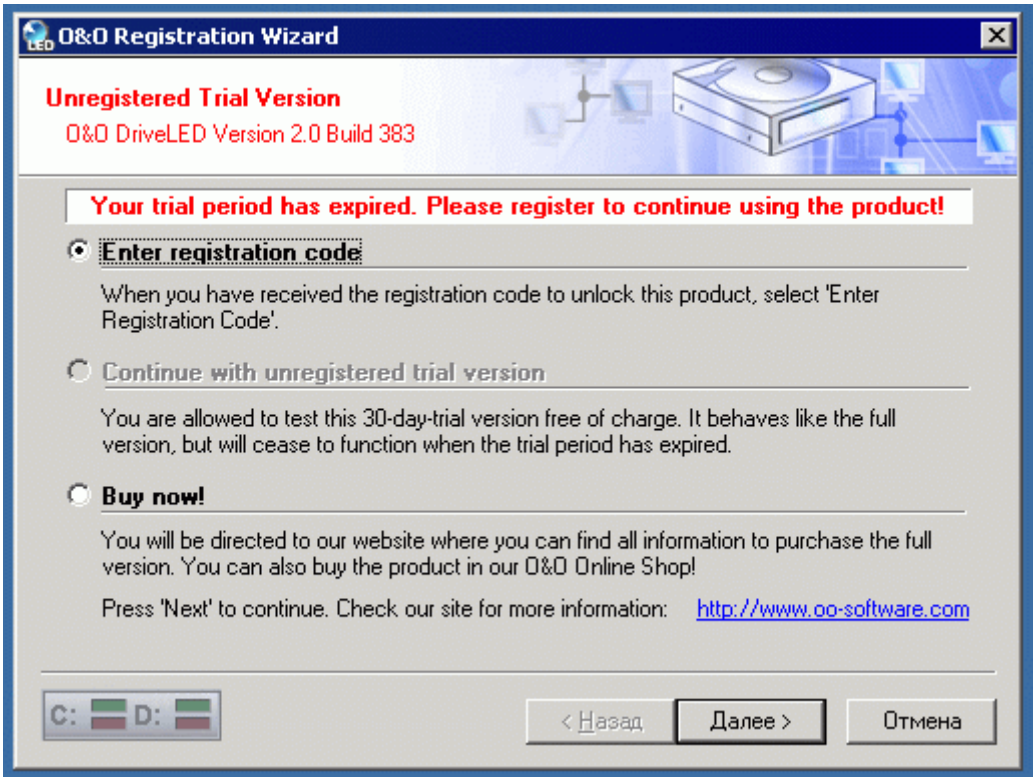

Рисунок 4 диалоговое окно, на которое мы поставим бряк

Это можно сделать либо любым Windows-шпионом (например, Spyxx, входящим в состав Microsoft Visual Studio), либо средствами самого soft-ice, а конкретно командой "HWND", выводящий список всех оконных элементов. Если в ответ на "HWND", soft-ice выплюнет: "Unable to find a desktop window", необходимо переключить контекст командой "ADDR".

Левая колонка содержит дескрипторы оконных элементов, правая — имена модулей, которым эти элементы принадлежат. Имя модулей не всегда совпадают с именами процессов, если окно принадлежит динамической библиотеке, то soft-ice пишет имя DLL, а не основного процесса. В данном случае, диалог обрабатывается библиотекой oodlrwrs, о чем можно узнать с помощью команды MOD, а фрагмент отчета выглядит так:

Handle	Class	WinProc	TID	Module
010098	VMDropTargetClass	00403810	138	VMwareUser
010096	VMDropTargetClass	00403810	138	VMwareUser
010094	VMDropTargetClass	00403810	138	VMwareUser
010090	VMDropTargetClass	00403810	138	VMwareUser
01001C	NDDEAgnt	0100BC04	F8	winlogon
120124	#32770 (Dialog)	00F7BC5E	2BC	comctl32
220132	#32770 (Dialog	00F7BC5E	2BC	oodlrwrs
1F00FE	**Button**	**00F7BC5E**	**2BC**	**oodlrwrs**
200102	**Button**	**00F7BC5E**	**2BC**	**oodlrwrs**
1B00F0	**Button**	**00F7BC5E**	**2BC**	**oodlrwrs**
320130	Static	00F7BC5E	2BC	oodlrwrs
210138	Static	77E19AA4	2BC	oodlrwrs
230116	Static	77E19AA4	2BC	oodlrwrs
24014C	Static	77E19AA4	2BC	oodlrwrs
1700F8	Static	00F7BC5E	2BC	oodlrwrs
20013A	Static	77E19AA4	2BC	oodlrwrs
1F0122	Static	77E19AA4	2BC	oodlrwrs

Листинг 1 определение дескрипторов окон и элементов управления

Мы видим, что три наших кнопки принадлежат диалогу #32770 с дескриптором 220132. В принципе, можно поставить точку останова и на 120124 — адрес оконной процедуры (WinProc) у них одинаков. Говорим: "BMSG 220132 WM_COMMAND" и выходим из soft-ice. Нажимаем на кнопку "далее >" и… отладчик послушно всплывает! Остается только немного протрассировать оконную процедуру в поисках кода, обрабатывающего это нажатие.

точки останова на данные

Чаще всего бывает так, что ключевой файл/регистрационные данные извлекаются в одном месте, а обрабатываются совсем в другом. Установив точку останова на GetWindowTextA мы перехватим код, считывающий введенный нами регистрационный номер, но как найти то место, где он сравнивается с оригиналом? Это легко!

Открываем MSDN, смотрим прототип функции GetWindowText, ага: указатель на возвращаемую строку находится во втором аргументе слева, значит, на момент вызова GetWindowTextA он будет располагаться по адресу ESP + 8 (четыре байта на hWnd и еще четыре на адрес возврата).

Говорим: "bpx GetWindowTextA", выходим из отладчика, вводим серийный номер в окно редактирования, нажимаем "OK" — отладчик всплывает (ну будем считать, что всплывает, в действительности он может и не всплыть, все зависит от того какую API-функцию использовал программист, так что тут возможны варианты). Даем команду "d esp->8" (если окно дампа отключено, перед этим необходимо дать команду "wd"), а затем "p ret" — в окне появляется введенная нами строка (см рис. 5)

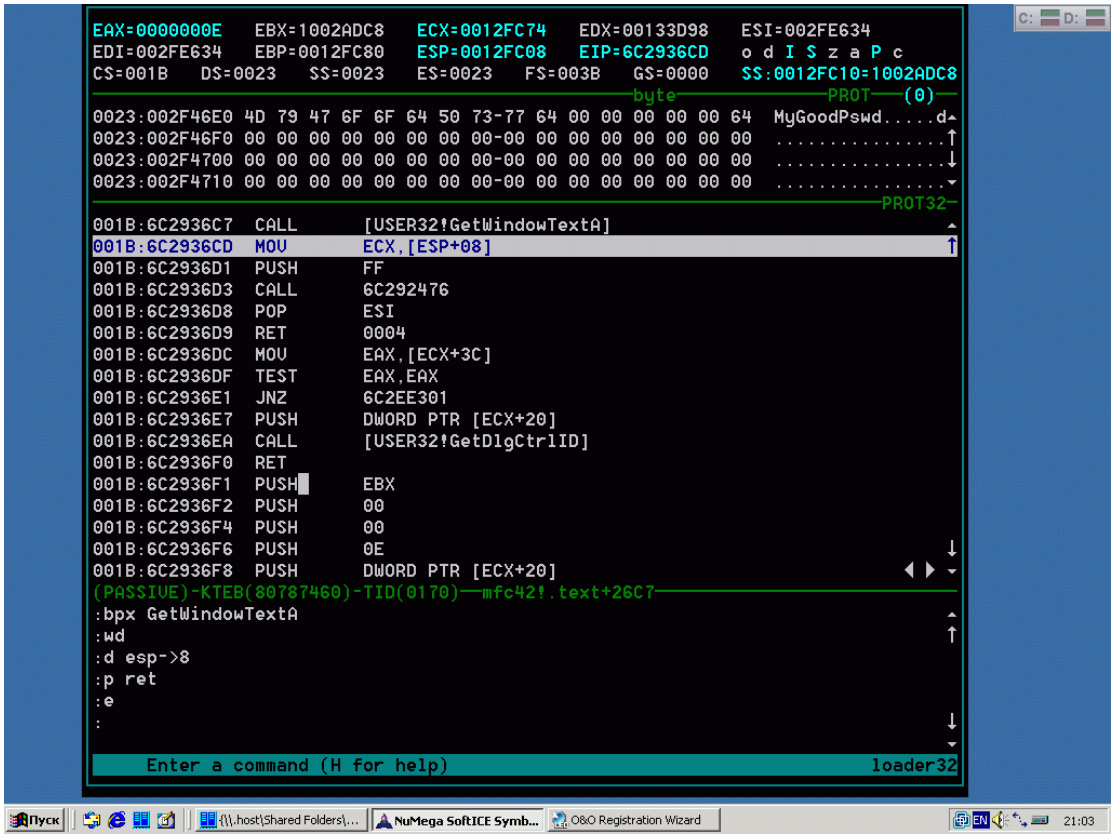

Рисунок 5 определение адреса по которому записывается считанный пароль

Все, что нам нужно — это ее адрес, равным в данном случае 2F46E0. Логично, чтобы сравнить пароль с оригиналом, защита должна его считать из памяти. И в этом момент из кусов появляется мы (в смысле мыщъх и отладчик). Команда "bpm 2F46E0" устанавливает точку останова на адрес 2F46E0, заставляя soft-ice всплывать при каждом чтении/записи этой ячейки. Звучит прекрасно, но на практике срабатывает далеко не всегда. Вовсе не факт, что в первое же всплытие отладчика выведет нас к защитному коду. Скорее всего здесь будет библиотечная функция, копирующая пароль в локальный буфер, передаваемый по цепочке другим функциям. И хорошо если по ссылке! Зачастую буфер передается по значению, т. е. копируется в другой буфер целиком. На каждый из таких буферов приходится ставить точку останова, а количество точек останова равно четырем. Это не ограничение отладчика, это просто архитектура у Пня такая.

Отсюда еще не следует, что точки останова на данные бесполезны, просто они сильны совсем в другой области. Вот, например, мы выяснили, что переменной x содержится флаг регистрации. Как именно выяснили не суть важно. Допустим встретили код типа: cmp [x],0/jz nag_screen (если переменная x равна нулю вывести ругательный диалог). Как определить где именно этот x инициализируется? В большинстве случав, перекрестные ссылки автоматически восстанавливаются ИДОЙ, однако, разработчик защитного механизма может легко ослепить ее, но едва ли он справится с командой "bpm x" (установить точку останова на доступ к переменой x). А вот другой вариант: изменили мы пару байтиков в программе, а она, обнаружив факт своего взлома, отказалась работать. Чтобы найти процедуру проверки целостности кода достаточно установить одну или несколько точек останова на модифицированные ячейки. Да много чего можно придумать, главное — фантазию иметь!

раскрутка стека

Внешние проявления защитного механизма засечь очень легко. Как правило, это либо окошко с надписью "trial expired", либо форма для ввода серийного номера. Установить точку останова на WM_COMMAND легко, но что это дает? Мы окажемся внутри оконной процедуры, в глубоких недрах которой зарыт защитный код. Можно, конечно, и потрассировать, но это же сколько времени уйдет! Вот бы узнать какие команды исполнялись до этого! Обратить выполнение программы вспять и посмотреть какой именно код определят факт регистрации

программы. Некоторые отладчики поддерживают механизм обратной трассировки (back trace), запоминая все выполняемые команды и складывая их в специальный буфер, однако, это сильно замедляет выполнение программы и выводит антиотладочные приемы на оперативный простор. Мы поступим иначе. soft-ice поддерживает шикарную команду "STACK", раскручивающую стек и выводящую адреса всех материнских функций. Не совсем равноценная замена обратной трассировки, но для большинства случаев ее вполне хватает.

В нашем случае, ответ отладчика выглядит так:

```
:STACK
0012E138 77E155B5      oorwiz!.text+0001AC5E
0012E168 77E15A3B      USER32!DefWindowProcW+0105
0012E188 77E1FB52      USER32!SendMessageW+0043
0012E214 77E1E6C3      USER32!WINNLSGetIMEHotkey+0E15
0012E254 77E1E561      USER32!EditWndProc+0075
0012E278 77E198DF      USER32!ScrollWindow+0096
0012E29C 77E13EB0      USER32!ShowCursor+0057
0012E2BC 77E16469      USER32!SetTimer+0435
0012E2E0 77E164E5      USER32!SetRect+0065
0012E300 00F7A1B6      USER32!CallWindowProcW+0019
0012E320 00F7A403      oorwiz!.text+000191B6
0012E33C 00F7BC02      oorwiz!.text+00019403 ; _AfxPostInitDialog
0012E39C 00F7BC92      oorwiz!.text+0001AC02 ; AfxWndProc
0012E3BC 77E13EB0      oorwiz!.text+0001AC92 ;
0012E3DC 77E1591B      USER32!SetTimer+0435
```

Листинг 2 раскрутка стека в soft-ice

Десять первых вызовов относятся к библиотеке USER32.DLL и не представляют для нас никакого интереса (soft-ice неправильно определил принадлежность вызова 12E138h, приписав его к oorwiz, но oorwiz не может располагаться по адресам 77E155B5 – эта зона принадлежит USER32). А вот одиннадцатый вызов 12E320, ведущий к адресу F7A403 весьма интересен. Заглянув сюда дизассемблером, мы обнаружим следующий код:

```
.text:1001A3E6         call   dword ptr [eax+10Ch]
.text:1001A3EC         test   eax, eax
.text:1001A3EE         jnz    short loc_1001A406
.text:1001A3F0         push   [ebp+arg_8]
.text:1001A3F3         mov    eax, [esi]
.text:1001A3F5         push   [ebp+arg_4]
.text:1001A3F8         mov    ecx, esi
.text:1001A3FA         push   [ebp+arg_0]
.text:1001A3FD         call   dword ptr [eax+110h]
.text:1001A403         mov    [ebp+var_4], eax ; адрес возврата
.text:1001A406
.text:1001A406 loc_1001A406:                   ; CODE XREF: CWnd::WindowProc()+24↑j
.text:1001A406         mov    eax, [ebp+var_4]
.text:1001A409         pop    esi
.text:1001A40A leave
.text:1001A40B         retn   0Ch
```

Листинг 3 фрагмент дизассемблерного листинга, с подозрительным условным переходом

Функция 1001A3FD:call dword ptr [eax+110h] — та самая, к которой ведет адрес возврата. Именно она и выводит противный регистрационный диалог. Прокручивая экран дизассемблера вверх, легко найти условный переход, расположенный по адресу 101AEE, который прыгает за диалог возврата. Изменив jnz на jmp short мы навсегда уберем диалог с экрана. Конечно, такая мера еще не зарегистрирует программу, но это все-таки кое-что!

отладка динамических библиотек

Loader32 (символьный загрузчик soft-ice) как будто позволяет загружать динамические библиотеки, но отлаживать их в автономном режиме не позволяет, что, собственного говоря, и не удивительно, ведь всякая такая библиотека — просто набор функций, вызываемых из основного процесса. Возьмем библиотеку oorwiz.dll, экспортирующую тройку функций с заманчивыми именами: RegWiz_InitReadOnly, RegWiz_InitTrial, RegWiz_InitLicMgr. Как их отладить?

Заходим в Loader32, выбираем пункт File → Load Export, указываем имя библиотеки (oorwiz.dll). В списке "Loader Symbols" немедленно появляется новое имя. Теперь, загружаем основной исполняемый файл (в данном случае oodled.exe) и устанавливаем точки останова на

интересующие нас функции ("bpx RegWiz_InitReadOnly", "bpx RegWiz_InitTrial", "bpx RegWiz_InitLicMgr"), заставляя отладчик всплывать при их вызове.

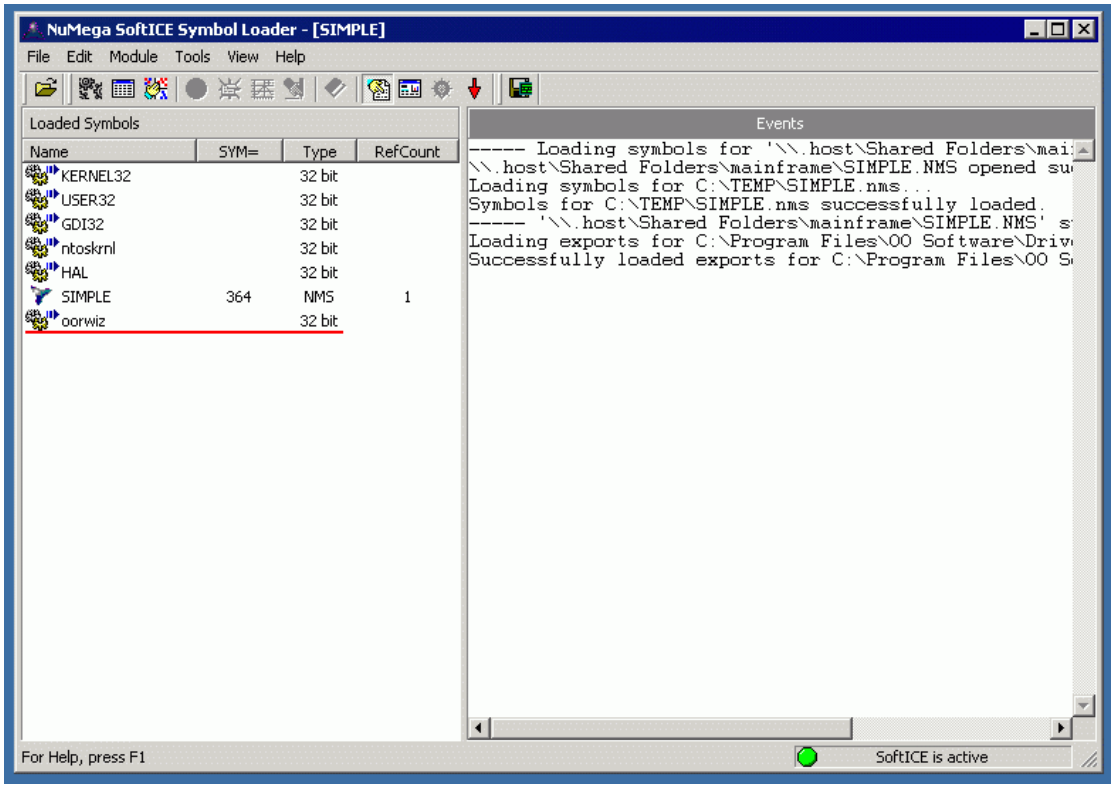

Рисунок 6 загрузка экспорта из динамических библиотек

Поскольку динамические библиотеки перемещаемы, адреса в дизассемблере могут не совпадать с отладчиком. Вот например, в oorwiz.dll IDA определяет адрес функции RegWiz_InitTrial как 10001D00h, а soft-ice как F60000. Ну и как с этим жить? А вот как: базовый адрес загрузки (Imagebase) равен 10000000h, в чем IDA честно признается в начале файла. Но загрузить по этому адресу библиотеку не получается и операционная система перемещает ее по адресу xxxx, о чем говорит команда "MOD" в soft-ice:

```
:mod
hMod     Base     Module Name File Name
80400000 804000C8 ntoskrnl    \WINNT\System32\ntoskrnl.exe
…
00400000 00400108 oodled      \Program Files\OO Software\DriveLED2\ood
00F30000 00F300B8 oodlrwrs    \Program Files\OO Software\DriveLED2\ood
00F60000 00F600F8 oorwiz      \Program Files\OO Software\DriveLED2\oor
10000000 100000C0 oodledrs    \Program Files\OO Software\DriveLED2\ood
```

Листинг 4 просмотр базовых адресов загрузки командой MOD

Разница базовых адресов составляет 10001000-F60000 == F0A1000, поэтому, чтобы перевести адрес из отладчика в дизассемблер к нему необходимо добавить F0A1000, а из дизассемблера в отладчик — отнять.

заключение

Рассмотренные приемы работают далеко не везде и не всегда. Разработчики далеко не идиоты и он взлома они все-таки защищаются. Лучше начинать с простых защит, постепенно переходя все к более сложным. Отладчик это сложный инструмент, который не осваивается за день. Исследование машинных кодов — настоящее искусство, которому учатся всю жизнь. Так что не нужно огорчаться, если что-то не получается. Чем хитрее защита и чем труднее взлом, тем более удовлетворение она приносит в конечном итоге! Кто-то сравнил это чувство с оргазмом. Ничего подобного! Хакерство намного круче!