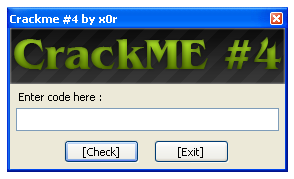
**Решение CrackMe #04 by x0r**

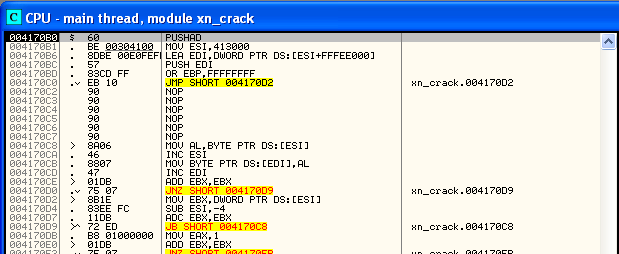
**Уровень:** Для начинающих  
  
В этой статье мы рассмотрим решение **[CrackMe #4 by x0r](http://forum.reverse4you.org/showthread.php?t=1220" \t "_blank)** предложенное [**[KSDR]**](http://forum.reverse4you.org/member.php?u=96). Скачать его можно [**здесь**](http://www.reverse4you.org/web_files/h-school/Cracking/Crackmes/Crackmes/xn_crackme.rar).  
  
**1. Осмотр местности**  
  
Скачиваем крякми, распаковываем архив и запускаем (не под отладчиком).

   
Рис. 1

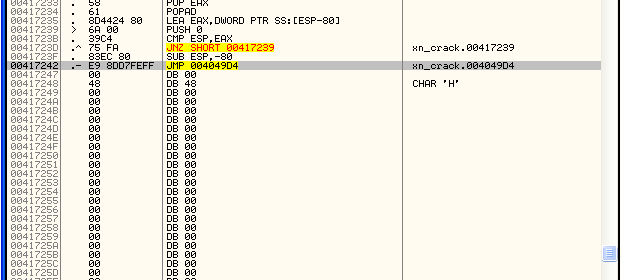
Перед нами появилось окно, с единственным полем ввода, в которое нам нужно ввести секретную фразу (далее пароль).   
  
Первое, что сейчас нужно сделать, это определить, как реагирует данный крякми на ввод какого-нибудь текста. Давайте введем туда что-нибудь и посмотрим, что же произойдет.

  
Рис. 2

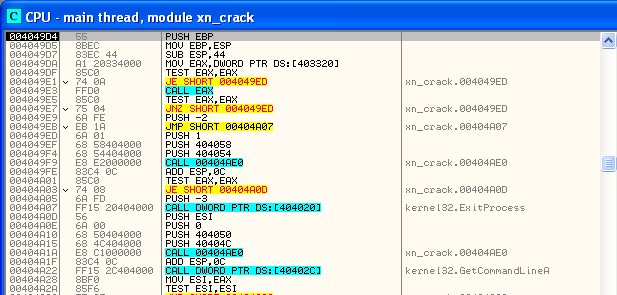
Итак, я ввел **«r0 Crew Forever»**, затем нажал **«Check»** и… И ничего не произошло. Облом =( Ладно, мы еще можем получить какую-нибудь зацепку посмотрев на функции и строки используемые в нутрии крякми, поэтому продолжим осмотр пациента уже в отладчике…  
  
Отпиваем глоток чайку, запускаем OllyDbg, затем загружаем в нее наш крякми **«File => Open => xn\_crackme.exe»** и жмем **«F9»** (Run). Сразу после этого появляется окошко, сигнализирующее о том, что возможно исследуемая программа упакована. Нажмите в нем **«Yes»** и мы окажемся на точке входа в программу:

  
   
Рис. 3

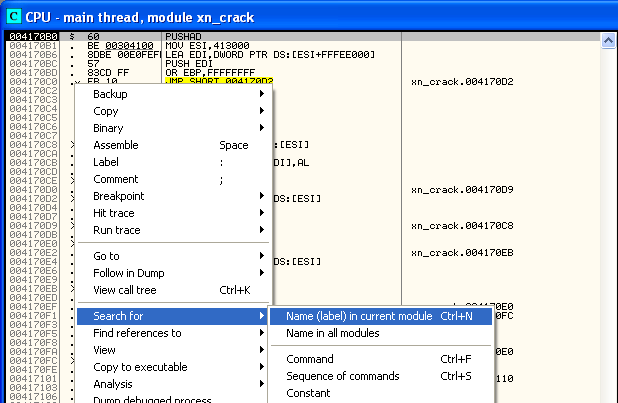
К сожалению, сразу проанализировать функции и строки у нас не получится. При загрузке крякми в отладчик, нам сообщили, что возможно крякми чем-то упакован. Из условия к крякми мы знаем, что он упакован с помощью UPX. А это значит, что для продолжения нашего анализа, нам нужно попасть на [**OEP**](http://ru.und3rgr0und.org/wiki/OEP).   
  
В нашем случае, найти OEP можно несколькими способами. Мы воспользуемся наиболее легким. Для этого прокрутите экран отладчика в низ, до тех пор, пока не увидите длинную вереницу нулей. У ее начала вы увидите инструкцию **JMP**, которая и ведет на **OEP**.

  
   
Рис. 4

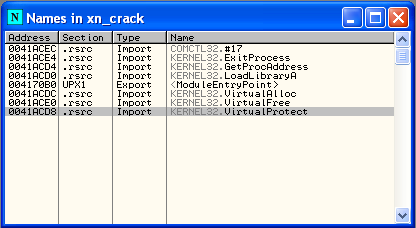
Поставьте на нее брейкпойнт, затем нажмите **«F9»** (Run) и потом **«F8»**. Все мы на OEP (Рис. 4.1).

  
   
Рис. 4.1

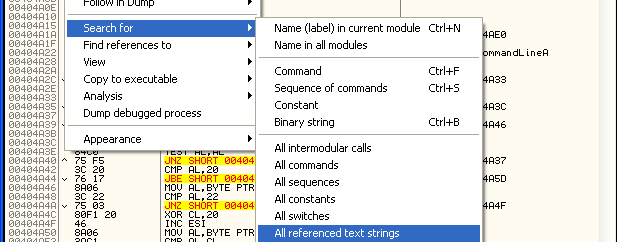
Теперь, с помощью OllyDbg, мы можем посмотреть на используемые «функции» и возможно найдем какие-нибудь интересные «строки», которые помогут нам в нашем задаче.   
  
  
***1.1. Смотрим функции***  
  
Для просмотра используемых программой функций нажмите **«Ctrl+N»**, либо, в главном окне отладчика, нажмите правую кнопку мыши, затем в выпадающем меню выберите **«Search for => Name (label) in current module»** (Рис. 5).

  
   
Рис. 5

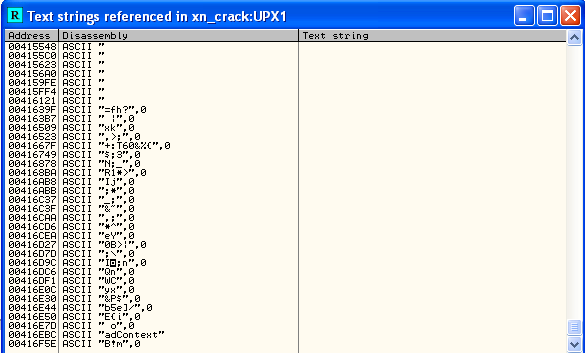
После этого появится окно, где будут перечислены функции, которые используются нашей программой (Рис. 6):

   
Рис. 6

Ну, что ж… Смотря на это окно мы понимаем, что и тут нас обломали. Все функции будут загружаться динамически, на это указывает практически полное отсутствие каких-либо функций + наличие **GetProcAddress** и **LoadLibrarryA**. Однако тут сразу бросаются в глаза 3-ри довольно интересные функции (**VirtualAlloc**, **VirtualFree** и **VirtualProtect**), отвечающие за работу с памятью. Возьмем на заметку, возможно, позже это пригодится нам.   
  
***1.2. Смотрим строки***  
  
Для просмотра имеющихся строк нажмите правую кнопку мыши (в главном окне отладчика), затем в выпадающем меню выберите **«Search for => All referenced text string»** (Рис. 7).

  
   
Рис. 7

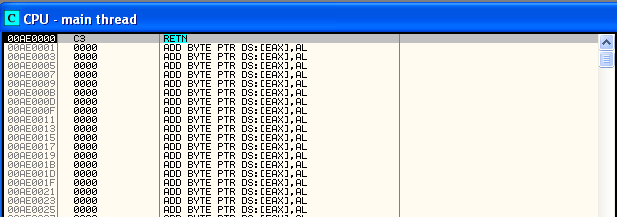
После этого появится окно, где будут перечислены строки, которые используются нашей программой (Рис. 8):

   
Рис. 8

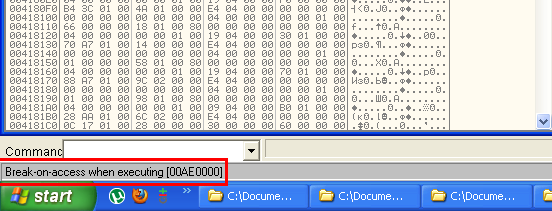
Очередной облом, ничего интересного не наблюдается…  
  
**2. Разведка боем**  
  
Так как внешний смотр пациента не дал никаких зацепок, то перейдем к активным действиям. Давайте запустим крякми под отладчиком и посмотрим, как он себя поведет.   
  
Сейчас вы должны находиться на **OEP**. Нажмите **«F9»** (Run), чтобы запустить программу, затем введите какой-нибудь текст.

   
Рис. 9

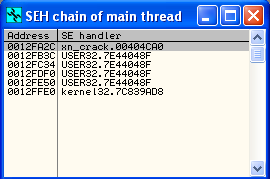
Я снова ввел **«r0 Crew Forever»**, нажал **«Check»** и… И на этот раз оказался где-то в космосе (вы должны быть тут же):

  
   
Рис. 10

Разбираемся с ситуацией… Если мы продолжим выполнение по **«F9»**, то крякми завершит свою работу (об этом позже)! Но почему? Ведь, когда мы его запускали не под отладчиком, все работало!?  
  
Ответ очевиден, мы попались на анти-отладку. Про наличие анти-отладки также говорится в условии к крякми **«немного анти-отладки»**.  
  
Я надеюсь, что вы еще не сдвинулись с адреса **0x00AE0000**? Хорошо. Тогда давайте осмотримся вокруг.  
  
Вначале обратите внимание на **«Status Bar»** он находится в самом низу, под основным окном (Рис. 11.1).

   
Рис. 11.1

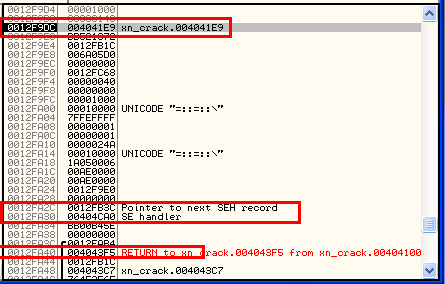
Что мы тут видим? А видим мы следующее, Оля говорит, что сработал брейкпойнт на доступ к памяти, во время выполнения инструкции, расположенной по адресу **0x00AE0000**. Но постойте-ка, какой брейкпойнт, какая память? Что за бред? Единственный брейкпойнт, который мы установили был на **OEP**! Да, вы абсолютно правы, но раз не мы, то кто? Тут-то наше предположение про **«анти-отладку»** и подтверждается. Если бы крякми выполнялся не под отладчиком, то это исключение передалось бы в **«обработчик исключений»**.  
  
Давайте посмотрим на имеющиеся обработчики **«View => SEH chain»** (Рис. 11.2):

  
Рис. 11.2

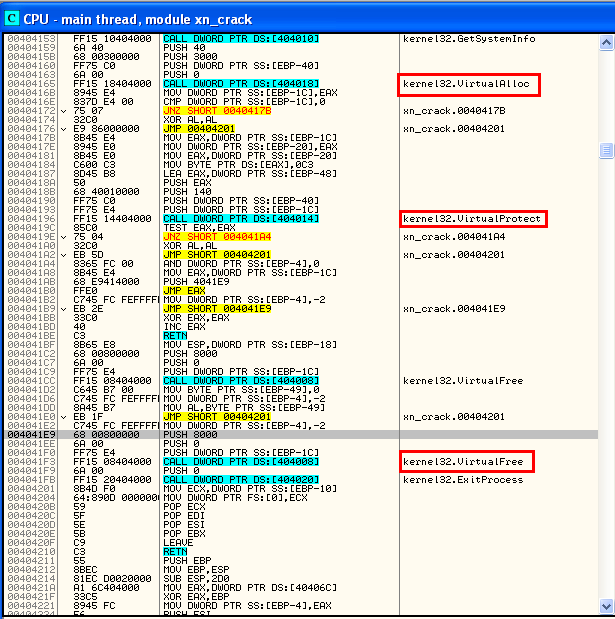
Как мы и предполагали, обработчик имеется, а это значит, что дальнейшее выполнение крякми должно идти через него. И если вы еще ничего не трогали, то мы можем перейти на [**SEH**](http://ru.und3rgr0und.org/wiki/SEH) и продолжить дальнейший анализ уже от туда. Но давайте не будем спешить.  
  
Возможно это не первый анти-отладочный трюк и до того, как мы попали на **0x00AE0000**, нас уже успели обнаружить и направить по неверному маршруту! А если это так, то дальнейший анализ заведет нас в тупик.  
  
Сейчас у нас два пути:

* Двинуться дальше, переключившись на SEH.
* Перезагрузить крякми и пройтись от OEP до 0x00AE0000, чтобы убедиться, что никакой анти-отладки до срабатывания «фальшивого» брейкопйнта не было и мы находимся на верном пути, а не движемся в болото.

В принципе мы можем положиться на удачу и выбрать первый путь. Забегая вперед, скажу, что конкретно в этом крякми удача вас не подведет ;) Но так как мы не ищем легких путей и хотим разобраться с принципом работы, а не способом взлома, то выберем второй путь. Ибо познавший суть вещей становится мастером.  
  
  
***2.1. Анализ ситуации***  
  
Сейчас нам нужно найти и исследовать код, который предшествовал переходу на адрес 0x00AE0000. Прежде чем это сделать, давайте немного раскрутим стэк. Чтобы хоть примерно понять, что происходило до перехода на 0x00AE0000.  
  
***2.1.1. Анализируем стэк***  
  
Давайте обратим внимание на (Рис. 12). Что происходило в стэке до того, как мы оказались там, где оказались?

  
Рис. 12

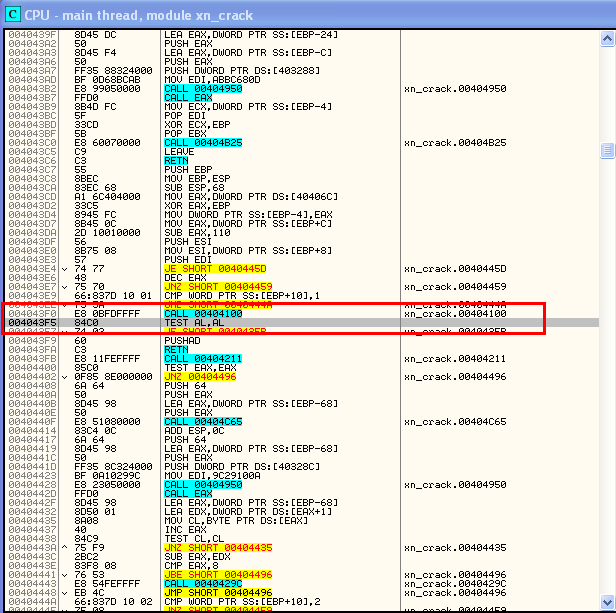
На данный момент **ESP == 12F9DC** и указывает на адрес **0x004041E9**. На него мы перейдем, если продолжим **«нормальное» выполнение** в отладчике. Давайте посмотрим, что нас ожидает, перейдя по этому адресу (Рис. 13). Жмем **«Ctrl+G»** и вводим **«4041E9»**:

  
   
Рис. 13

А ожидает нас следующее. Сразу после перехода осуществляется очистка памяти по адресу **0x00AE0000** (**VritualFree**), после чего крякми завершает свою работу (**ExitProcess**).   
  
Не могут не бросаться в глаза **«подозрительные» функции** (спасибо Оле), которые были обнаружены еще в начале нашего исследования, во время осмотра используемых функций (Рис. 6).  
  
Продолжим осмотр стэка. Нас интересуют все записи (значения), которые соответствуют пределу от **0x00401000** до **0x00413000**, т.е. находятся в секции кода (**UPX0**) исследуемого приложения (см. Alt+M). Среди, ближайших, из таких значений мы находим следующие два:

* **0x00404CA0** – SEH-обработчик на который должно перейти управление, если бы не было отладчика (см. View => SEH chain)
* **0x004043F5** – Ближайшая процедура, из которой мы вышли на адрес 0x00AE0000.

Давайте перейдем на адрес **0x004043F**5 (Ctrl+G и ввести 4043F5) и посмотрим, что она собой представляет (Рис. 14).

  
   
Рис. 14

Что мы видим? В принципе, ничего особенного. Единственное, что мы можем сейчас предположить, так это то, что переход на **0x00AE0000** находится внутри процедуры, расположенной по адресу **0x00404100** и вызываемой по адресу **0x004043F0**. Если вы зайдете в эту процедуру и посмотрите на нее, то вы заметите, что **Рис. 13** является ее частью ;) Так же мы можем сказать, что адрес **0x004043F0** находится внутри процедуры начинающейся с адреса **0x004043C7**. Этот вывод сделан на основании стандартного пролога функции (PUSH EBP/MOV EBP, ESP) выше.  
  
Итак, после анализа стэка мы имеем небольшую цепочку функций вызываемых **до срабатывания «фальшивого» брейкпойнта**. Выглядит она следующим образом:

Code:

Функция (0x004043C7) =>

Функция (0x00404100) =>

Фальшивый брейкпойнт (?) =>

RETN => 0x004041E9 => Exit

SEH => 0x00404CA0 => ?

Сейчас нам нужно определить, что происходило до вызова функции **0x004043C7**. Определять это мы будем идя от **OEP**.  
  
***2.1.2. Анализируем код начиная с OEP***  
  
**Перегружаем крякми** и **становимся на OEP**. Сейчас нам нужно определить план действий.   
  
Вначале давайте решим, что считать началом выполнения программы? Наш крякми довольно маленький, поэтому очевидно, что началом его работы можно считать нажатие на кнопку **«[Check]»**. Отсюда следует, что нам нужно найти **«обработчик кнопки [Check]»**.   
  
Как это сделать? Так как крякми является **«оконным приложением»**, то в первую очередь нужно искать **«обработчик сообщений Windows»** и уже в нем искать нить, ведущую к **«обработчику кнопки [Check]»**.  
  
После того, как мы найдем обработчик кнопки, нам нужно дойти до функции **0x004043C7** и убедиться, что до ее запуска не было никаких анти-отладочных проверок.  
  
Ну, раз план ясен – действуем! Для того, чтобы найти **«обработчик сообщений Windows»** нам нужно найти кусок кода, который отвечает за создание окна. Как его найти? Ну, можно покопаться в сером веществе или в гугле (кто как предпочитает) и вспомнить все способы задания **WndProc**. После чего установить брейкпойнты на соответствующие функции и надеяться, что вы чего-то не забыли.   
  
Мы же будем делать по-другому, например, лень вспоминать/искать – нужное подчеркнуть. Вместо установок брейкпойнтов, воспользуемся способом, который опирается на то, что сразу после вызова функции создания окна – мы теряем управление, т.е:

Code:

PUSH чего-то там

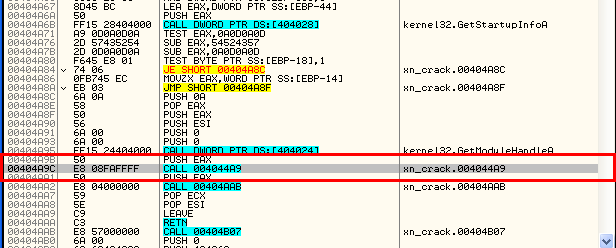
CALL CreateWindow <= После этого мы окажемся

CALL Exit <= не тут

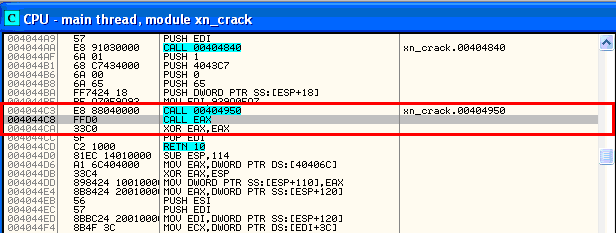
…

Proc WndProc <= A ТУТ!

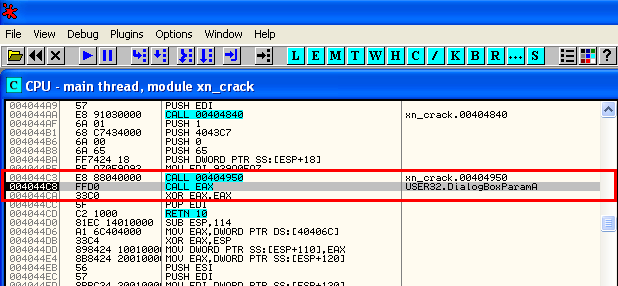
Итак, поехали! Стоя на **OEP** зажимаем кнопку **«F8»** (в некоторых места желательно использовать **«F4»**, буду надеяться, что догадаетесь сами, где и как) и смотрим, где теряется управление (Рис. 15).

  
   
Рис. 15

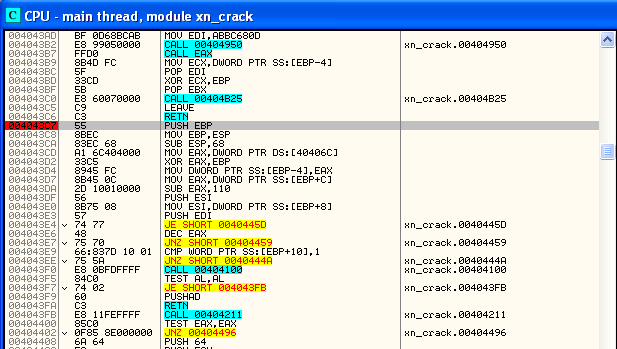
Первый раз мы теряем управление по адресу **0x00404A9C**. Перегружаем крякми, повторяем свои действия и дойдя до этого адреса нажимаем **«F7»**. Далее снова зажимаем **«F8»** (Рис. 16).

  
   
Рис. 16

Второй раз мы теряем управление по адресу **0x004044C8**. Самые внимательные могли заметить, что этот вызов соответствует функции **DialogBoxParamA** (Рис. 17):

  
   
Рис. 17

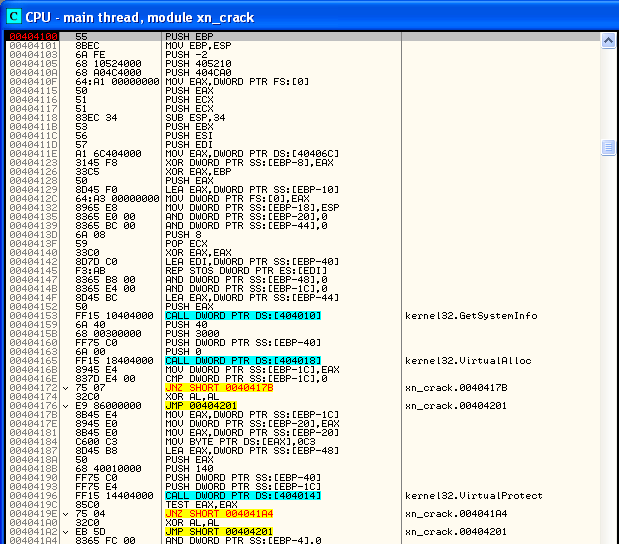
Как раз то, что мы искали! Если вы посмотрите в стэк и сравните переданные функции параметры, то увидите, что **WndProc** соответствует адресу **0x004043С7**. Вам это ничего не напоминает? Верно! Это имена та функция, дойдя до которой, мы собирались убедиться в том, что до срабатывания **«фальшивого» брейкпойнта** не было никакой **другой** анти-отладки. Что же, наша работа немного сократилась =)  
  
Что касается функции **0x00404950**, то она возвращает адрес функции **DialogBoxParamA**.  
  
Теперь, когда мы определили адрес **WndProc**, давайте установим на нее брейкпойнт и нажмем **«F9»**. После чего мы окажемся тут (Рис. 18):

  
   
Рис. 18

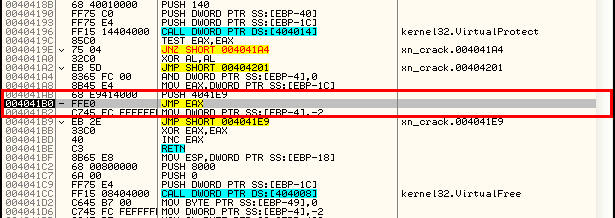
Это место должно быть уже вам знакомо. Чтобы освежить память посмотрите на **Рис. 14**. Проанализировав функцию, я определил, что обработчик кнопки располагается по адресу **0x004043F0** и срабатывает он во время наступления события **WM\_COMMAND** с **wParam = 1**(единица в нашем случае обозначает идентификатор кнопки).  
  
Снимите брейкпойнт с **0x004043C7**. Поставьте брейкпойнт по адресу **0x00404100** и нажимаем **«F9»**. Крякми продолжит свою работу. Затем введите какой-нибудь текст в поле ввода (Рис. 19) и нажмите **«[Check]»**.

   
Рис. 19

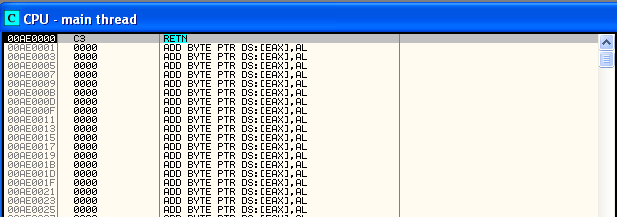
После чего мы окажемся тут (Рис. 20).

  
   
Рис. 20

Эта функция вам так же должна быть знакома. Единственное, что нам тут было неизвестно это то, для чего тут нужны были функции **VirtualAlloc**, **VirtualProtect**, **VirtualFree** и где собственно происходил переход на **0x00AE0000**. Теперь мы знаем, что с помощью этих функций осуществлялся один из известных анти-отладочных трюков, который был **[описанн](http://www.openrce.org/reference_library/anti_reversing_view/20/OllyDbg%20Memory%20Breakpoint%20Detection/" \t "_blank)** еще в 2006 году. Сам же переход происходит по адресу **0x4041B0** (Рис. 21):

  
   
Рис. 21

Снимаем брейкпойнт с **0x004043C7** и жмем **«F9»** (Рис. 22).

  
   
Рис. 22

И мы оказываемся там же, куда мы уже попадали еще на **Рис. 10**. Зачем же мы все это проделали? Если вы забыли, то напомню, «что бы не зайти в тупик». Если бы до первого анти-отладочного приема имелся еще какой-нибудь, то он бы мог направить нас по неправильному маршруту и в итоге завести в какое-нибудь болото.  
  
Сейчас мы имеем следующую цепочку вызовов:

Code:

WndProc (0x004043C7) =>

Обработчик кнопки (0x00404100) =>

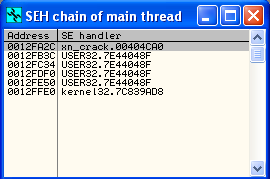
Фальшивый брейкпойнт (4041B0) =>

Анти-отладка (00AE0000):

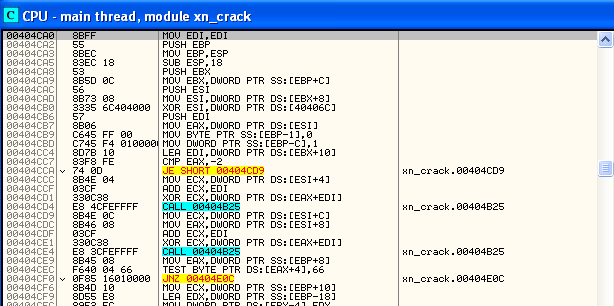
RETN => 0x004041E9 => Exit

SEH => 0x00404CA0 => Продолжение работы

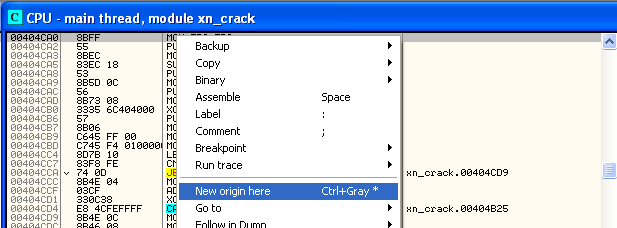
Из которой следует, что дальнейшее выполнение работы программы должно идти через **SEH**. Для этого идем на вкладку **«View => SEH chain»** (Рис. 23):

   
Рис. 23

Далее делаем **двойной щелчок** по самому верхнему обработчику. После чего мы окажемся тут (Рис. 24):

  
   
Рис. 24

Далее устанавливаем курсор на адрес **0x00404CA0**, жмем правой кнопкой мыши и выбираем **«New origin here»**. Это действие переместит указатель команд (регистр EIP) на адрес **0x00404CA0**, что в свою очередь означает, что следующая команда начнет свое выполнения, начиная с указанного адреса (Рис. 25)

  
   
Рис. 25

***2.2. Основной механизм крякми***  
  
Теперь, когда мы выяснили причину возникновения **«фальшивого» брейкпойнта**, можно продолжать дальнейшее исследование крякми.   
  
Сейчас вы должны находиться в SEH. И нам нужно ответить на следующие вопросы.

1. Как крякми получает доступ к строке?
2. Где происходит сравнение введенной строки?
3. Где выводится сообщение, сообщающее о правильном решении?

На эти вопросы мы постараемся ответить в последующих трех разделах.

**ПРИМЕЧАНИЕ:** Начиная с этого места, я не буду подробно описывать свои действия, иначе решение этого крякми затянется на долго. Если вам не будет понятно, почему я пропустил, тот или иной кусок кода, вы всегда можете исследовать и понять его сами, там ничего сложного нет.

***2.2.1. Определение доступа к строке***  
  
Определить код, отвечающий за получение введенной строки, можно поставив брейкпойнты на определенные функции (например, **GetWindowText/GetDlgItemText**). Беря в учет то, что исследуемый крякми (как в принципе и большинство) имеет сравнительно небольшой размер, я пошел другим путем и просто напросто прошел его вдоль и в поперек по **«F7»**.  
  
Результат оказался следующим:

Code:

SEH =>

Функция 404F35 =>

4041СС: VirtualFree – 0x00AE000

404204: Удаляем установленный SEH

Функция 404211 =>

Очистка памяти 404С65 => (с 12F76C до 12FA38)

Очистка памяти 404С65 => (с 12FA4C до 12FAB0)

Функция 404950 =>

Вернуть указатель на функцию 404680 => (eax = GetWindowTextA)

GetWindowTextA => (Edit, буфер=12FA4C)

Функция 40429С =>

Бинго!

12FA18: Great Work!\_

12FA30: Yes!\_

12FA24: gjebaZinw\_

Функция 404950 =>

Вернуть указатель 404680 => (eax = GetForegroundWindow)

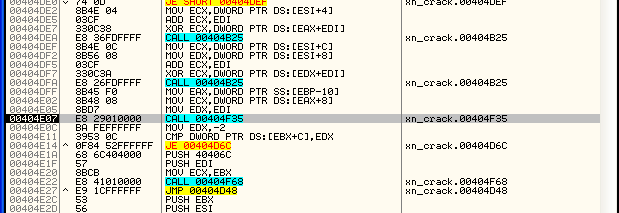
GetForegroundWindow => (eax = текущее окно (Olly))

Функция 404950 =>

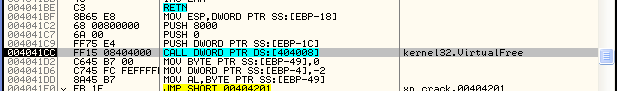
Вернуть указатель 404680 => (eax = GetDlgItemTextA)

GetDlgItemTextA(Olly, Edit, буфер=12F9B4)

Разберемся с полученным результатом более подробно.  
  
Механизм защиты крякми (если так можно сказать) сосредоточен в функции **0x00404F35**, которая вызывается из **SEH** (Рис. 26).

  
   
Рис. 26

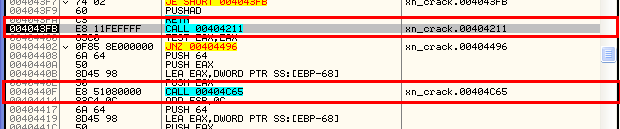
Вначале она осуществляет очистку памяти, ранее выделенную под «фальшивый» брейкпойнт (Рис. 27)

  
   
Рис. 27

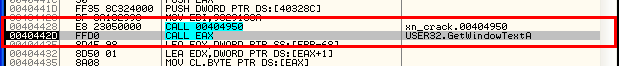
Затем удаляет установленный SEH (Рис. 28)

  
http://www.reverse4you.org/web_files/h-school/Cracking/Crackmes/Manuals/Crackme_04/26.png   
Рис. 28

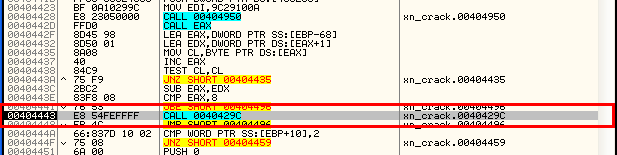
После этого очищает память (с **12F76C** до **12FA38**) и (с **12FA4C** до **12FAB0**) для дальнейшего использования (Рис. 29)

  
   
Рис. 29

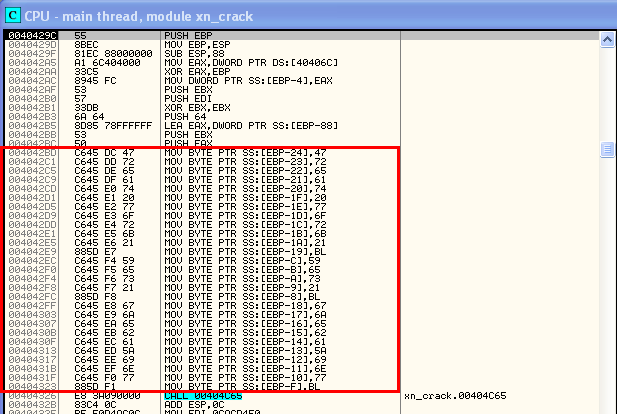
Далее получает указатель на **GetWindowTextA** и вызывает ее с **«дескриптором поля вода»** и указателем на буфер **0x0012FA4C** (Рис. 30):

  
   
Рис. 30

Как выяснится позже – это был **«фейковый» доступ** к введенной строке!  
  
Затем вызывается **«Функция 40429С»** в которой сосредоточен **«основной механизм защиты»** крякми (Рис. 31).

  
   
Рис. 31

Сразу после входа в нее по **«F7»**, ваше внимание должна привлечь следующая последовательность инструкций (Рис. 32):

  
   
Рис. 32

Им соответствует запись очень интересных строк, по следующим адресам:

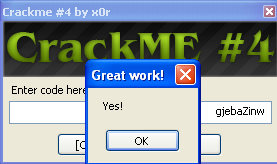
Code:

12FA18: Great Work!\_

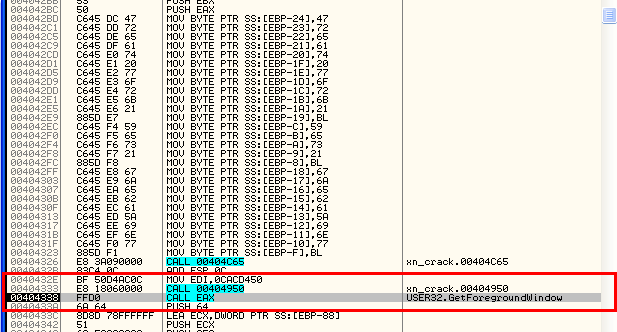
12FA30: Yes!\_

12FA24: gjebaZinw\_

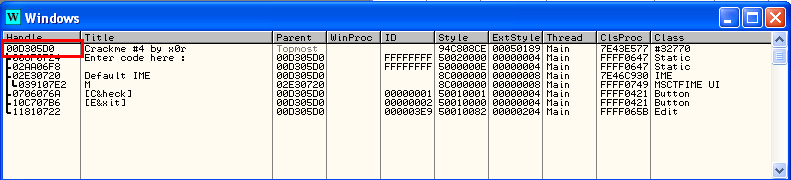
Очевидно, что первые две строки будут использоваться при выводе об успешном вводе пароля. А вот третья строка, возможно и является тем самым паролем!  
  
Запустите крякми (не под отладчиком) и введите **«gjebaZinw»**. Вы увидите следующее (Рис. 33)

   
Рис. 33

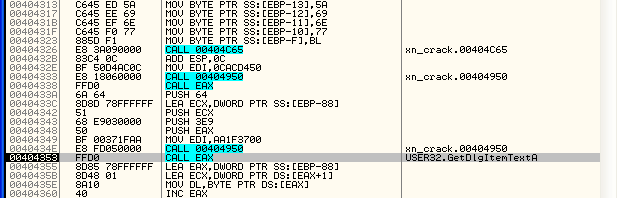
Не спешите закрывать отладчик! Лучше сходите и приготовьте еще чашечку чая.   
  
На самом деле автор этого крякми сильно облажался. На что он надеялся я собственно не понимаю. Как и не понимаю того, за чем был нужен фейковый вызов GetWindowTextA, если сразу за ним идет функция в нутрии которой находятся подобные строки (Рис. 32). Я бы как минимум перенес запись этих строк за вызов **GetForegroundWindow**.  
  
Давайте попробуем оправдать надежды автора! Перегрузите крякми, но уже **под отладчиком**. Введите найденный пароль **«gjebaZinw»** и нажмите **[Check]**.  
  
Ну как, что-то получилось? Пароль сработал? Нет? У меня тоже. Оказывается, это еще не конец. В программе присутствует еще как минимум один метод анти-отладки. И что самое интересное, автор не учел того момента, что единицы будут проверять найденный пароль **«gjebaZinw»**, во время активной сессии отладки, непосредственно в самом отладчике! Если такие вообще будут… А значит, на его трюк практически никто не попадется (тех кто получил пароль на халяву, в расчет не берем).  
  
Проблема, почему крякми отказывается работать под отладчиком, раскрывается **в следующих четырех вызовах**. **Первый** получает указатель на функцию **GetForegroundWindow**, **второй**вызывает ее (Рис. 34)

  
   
Рис. 34

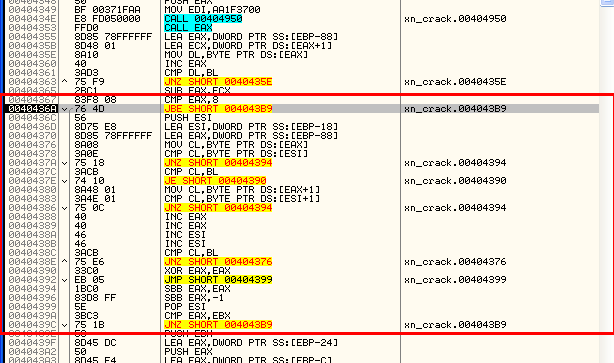
Функция **GetForegroundWindow** возвращает дескриптор приоритетного окна (окна, с которым пользователь в настоящее время работает). Под отладчиком этим окном является окно самого отладчика. Результат своей работы эта функция возвращает в регистре **EAX**.  
  
Собственно таким макром нас и определяют. Чтобы обойти этот фокус, сразу после выполнения **GetForegroundWindow**, измените регистр **EAX**, вписав правильный дескриптор окна. Узнать дескриптор окна крякми можно с помощью **«View => Windows»** (Рис. 35)

  
   
Рис. 35

Далее после вызова **GetForegroundWindow**, идет вызов процедуры, которая возвращает адрес **GetDlgItemTextA**. Затем происходит вызов **GetDlgItemTextA** со следующими параметрами [дескриптор окна, поле ввода=3E9, буфер=12F9B4] (Рис. 36). Здесь **«дескриптор окна» == дескриптору возвращаемому GetForegroundWindow**.

  
   
Рис. 36

В итоге получается, что введенный нами пароль считывается два раза, и что самое интересное двумя разными функциями (дальше разберемся).  
  
С получением строк разобрались – переходим к сравнению.  
  
***2.2.2. Поиск сравнения введенного пароля***  
  
Код сравнения введенной строки располагается сразу за вызовом функции **GetDlgItemTextA**, начиная с адреса **0x00404367** по **0x0040439C** (Рис. 37). Тут же становится ясно, что используется буфер, который заполнила функция **GetDlgItemTextA**, а не **GetWindowTextA**. Отсюда следует, **GetWindowTextA** – это **фейковый вызов**.

  
   
Рис. 37

Цепочка вызовов теперь имеет следующий вид:

Code:

SEH =>

Функция 404F35 =>

4041СС: VirtualFree – 0x00AE000

404204: Удаляем установленный SEH

Функция 404211 =>

Очистка памяти 404С65 => (с 12F76C до 12FA38)

Очистка памяти 404С65 => (с 12FA4C до 12FAB0)

Функция 404950 =>

Вернуть указатель на функцию 404680 => (eax = GetWindowTextA)

GetWindowTextA => (Edit, буфер=12FA4C)

Функция 40429С =>

Бинго!

12FA18: Great Work!\_

12FA30: Yes!\_

12FA24: gjebaZinw\_

Функция 404950 =>

Вернуть указатель 404680 => (eax = GetForegroundWindow)

GetForegroundWindow => (eax = текущее окно (Olly))

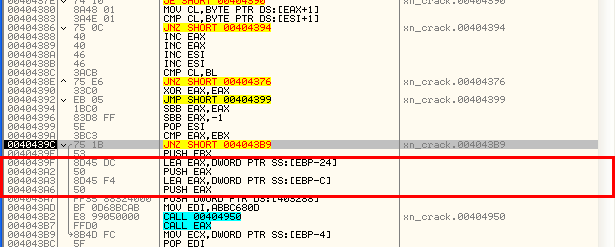
Функция 404950 =>

Вернуть указатель 404680 => (eax = GetDlgItemTextA)

GetDlgItemTextA(Olly, буфер=12F9B4)

+ Проверка (0x00404367 – 40439C)

Переходим к поиску окна выводящего сообщение об успехе.  
  
***2.2.3. Поиск окна выводящего сообщение об успехе***  
  
Сразу после проверки происходит подготовка данных для окна (Рис. 38)

  
   
Рис. 38

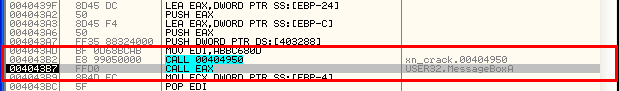
Тут происходит сохранение в стэке следующих строк:

Code:

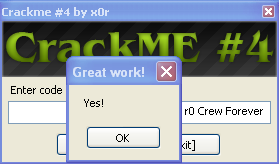
Great Work!

Yes!

Сразу после этого происходит вызов функции, которая подготавливает указатель на **MessageBoxA**. И собственно непосредственный вызов самой **MessageBoxA** (Рис. 39).

  
   
Рис. 39

В результате получаем следующее окно (Рис. 40).

   
Рис. 40

У вас на **Рис. 40** в место **«r0 Crew Forever»** должен быть пароль **«gjebaZinw»**, я просто не перегружал отладчик, как просил сделать вас после **Рис. 33**, а продолжил идти с теми данными, которые уже были.  
  
Теперь можете допить свой чай (если он конечно у вас еще остался) – исследование крякми завершено! Все последующие функции, которые последуют за **MessageBoxA**, просто раскрутят и очистят стек от предыдущих вызовов, после чего будет передано управление в **«обработчик сообщений Windows»** (WndProc).  
  
Окончательная цепочка вызовов будет иметь, примерно, следующий вид:

Code:

SEH =>

Функция 404F35 =>

4041СС: VirtualFree – 0x00AE000

404204: Удаляем установленный SEH

Функция 404211 =>

Очистка памяти 404С65 => (с 12F76C до 12FA38)

Очистка памяти 404С65 => (с 12FA4C до 12FAB0)

Функция 404950 =>

Вернуть указатель на функцию 404680 => (eax = GetWindowTextA)

GetWindowTextA => (Edit, буфер=12FA4C)

Функция 40429С =>

Бинго!

12FA18: Great Work!\_

12FA30: Yes!\_

12FA24: gjebaZinw\_

Функция 404950 =>

Вернуть указатель 404680 => (eax = GetForegroundWindow)

GetForegroundWindow => (eax = текущее окно (Olly))

Функция 404950 =>

Вернуть указатель 404680 => (eax = GetDlgItemTextA)

GetDlgItemTextA(Olly, буфер=12F9B4)

Проверка (0x00404367 – 40439C)

Подготовка строк для MessageBox:

Great Work!

Yes!

Функция 404950 =>

Вернуть указатель 404680 => (eax = ?)

Функция 404880 =>

Формирует указатель на MessageBoxA 4044D0 =>

(eax = MessageBoxA)

RETN

Функция 4046A0 =>

Выход из всех ранее вызванных функций.

MessageBoxA

RETN

RETN

RETN => Windows => WndProc

Успехов =)

*Last edited by root; 08-10-2012 at 17:01.*

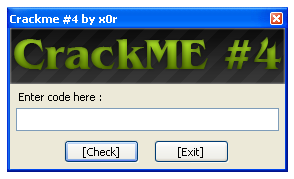
Дорогу осилит идущий. (К. Касперск

**Решение CrackMe #04 by x0r**

**Уровень:** Для начинающих  
  
В этой статье мы рассмотрим решение **[CrackMe #4 by x0r](http://forum.reverse4you.org/showthread.php?t=1220" \t "_blank)** предложенное [**[KSDR]**](http://forum.reverse4you.org/member.php?u=96). Скачать его можно [**здесь**](http://www.reverse4you.org/web_files/h-school/Cracking/Crackmes/Crackmes/xn_crackme.rar).  
  
Для осуществления нашей цели нам понадобятся:

* Руки
* OllyDbg
* Пару чашек чая

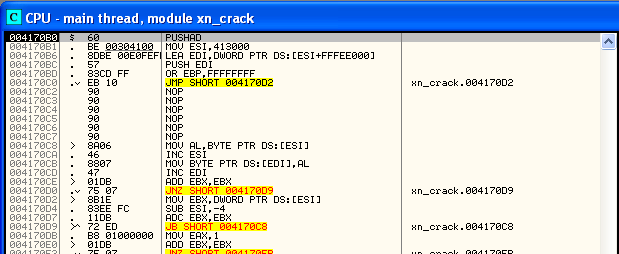
Если все необходимые инструменты имеются, значит можно приступать.  
  
  
**1. Осмотр местности**  
  
Скачиваем крякми, распаковываем архив и запускаем (не под отладчиком).

   
Рис. 1

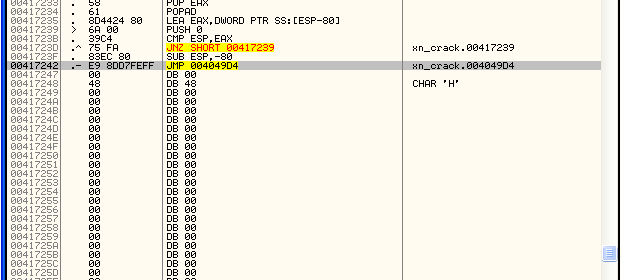
Перед нами появилось окно, с единственным полем ввода, в которое нам нужно ввести секретную фразу (далее пароль).   
  
Первое, что сейчас нужно сделать, это определить, как реагирует данный крякми на ввод какого-нибудь текста. Давайте введем туда что-нибудь и посмотрим, что же произойдет.

  
Рис. 2

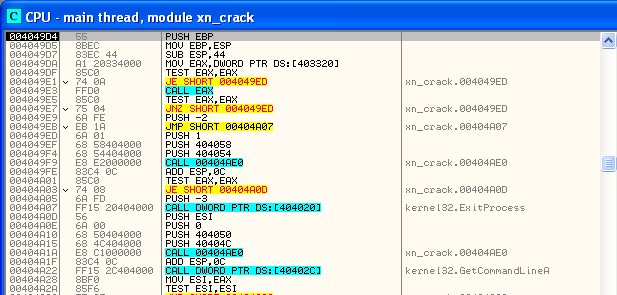
Итак, я ввел **«r0 Crew Forever»**, затем нажал **«Check»** и… И ничего не произошло. Облом =( Ладно, мы еще можем получить какую-нибудь зацепку посмотрев на функции и строки используемые в нутрии крякми, поэтому продолжим осмотр пациента уже в отладчике…  
  
Отпиваем глоток чайку, запускаем OllyDbg, затем загружаем в нее наш крякми **«File => Open => xn\_crackme.exe»** и жмем **«F9»** (Run). Сразу после этого появляется окошко, сигнализирующее о том, что возможно исследуемая программа упакована. Нажмите в нем **«Yes»** и мы окажемся на точке входа в программу:

  
   
Рис. 3

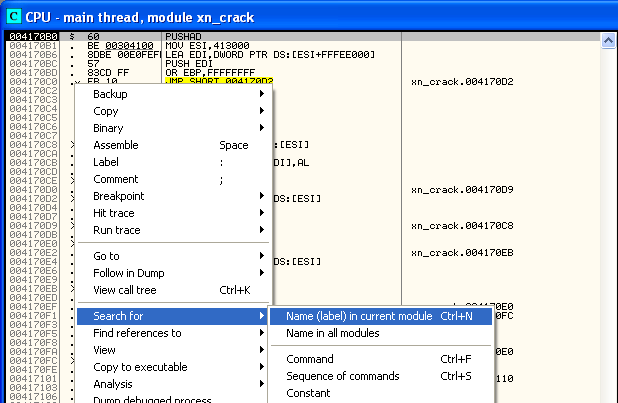
К сожалению, сразу проанализировать функции и строки у нас не получится. При загрузке крякми в отладчик, нам сообщили, что возможно крякми чем-то упакован. Из условия к крякми мы знаем, что он упакован с помощью UPX. А это значит, что для продолжения нашего анализа, нам нужно попасть на [**OEP**](http://ru.und3rgr0und.org/wiki/OEP).   
  
В нашем случае, найти OEP можно несколькими способами. Мы воспользуемся наиболее легким. Для этого прокрутите экран отладчика в низ, до тех пор, пока не увидите длинную вереницу нулей. У ее начала вы увидите инструкцию **JMP**, которая и ведет на **OEP**.

  
   
Рис. 4

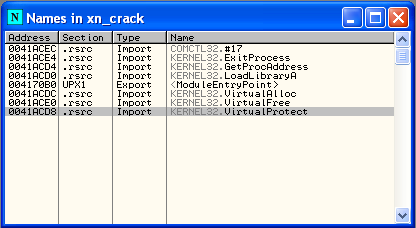
Поставьте на нее брейкпойнт, затем нажмите **«F9»** (Run) и потом **«F8»**. Все мы на OEP (Рис. 4.1).

  
   
Рис. 4.1

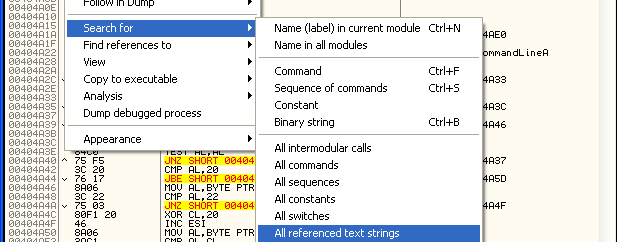
Теперь, с помощью OllyDbg, мы можем посмотреть на используемые «функции» и возможно найдем какие-нибудь интересные «строки», которые помогут нам в нашем задаче.   
  
  
***1.1. Смотрим функции***  
  
Для просмотра используемых программой функций нажмите **«Ctrl+N»**, либо, в главном окне отладчика, нажмите правую кнопку мыши, затем в выпадающем меню выберите **«Search for => Name (label) in current module»** (Рис. 5).

  
   
Рис. 5

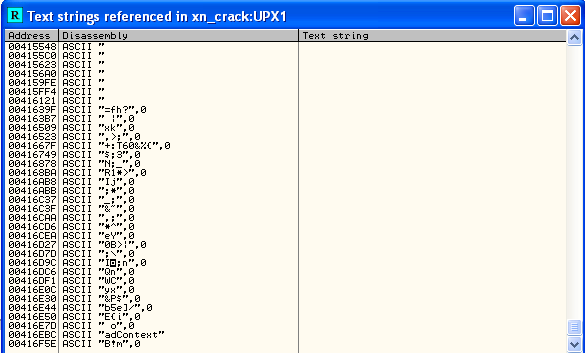
После этого появится окно, где будут перечислены функции, которые используются нашей программой (Рис. 6):

   
Рис. 6

Ну, что ж… Смотря на это окно мы понимаем, что и тут нас обломали. Все функции будут загружаться динамически, на это указывает практически полное отсутствие каких-либо функций + наличие **GetProcAddress** и **LoadLibrarryA**. Однако тут сразу бросаются в глаза 3-ри довольно интересные функции (**VirtualAlloc**, **VirtualFree** и **VirtualProtect**), отвечающие за работу с памятью. Возьмем на заметку, возможно, позже это пригодится нам.   
  
  
***1.2. Смотрим строки***  
  
Для просмотра имеющихся строк нажмите правую кнопку мыши (в главном окне отладчика), затем в выпадающем меню выберите **«Search for => All referenced text string»** (Рис. 7).

  
   
Рис. 7

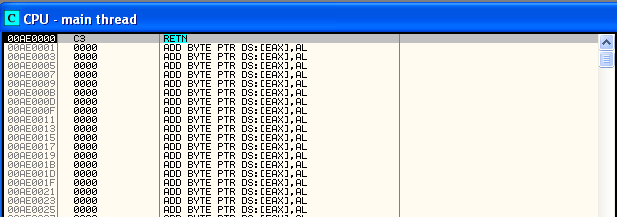
После этого появится окно, где будут перечислены строки, которые используются нашей программой (Рис. 8):

   
Рис. 8

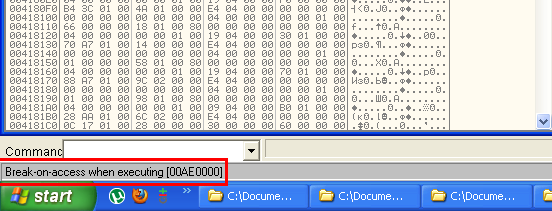
Очередной облом, ничего интересного не наблюдается…  
  
  
**2. Разведка боем**  
  
Так как внешний смотр пациента не дал никаких зацепок, то перейдем к активным действиям. Давайте запустим крякми под отладчиком и посмотрим, как он себя поведет.   
  
Сейчас вы должны находиться на **OEP**. Нажмите **«F9»** (Run), чтобы запустить программу, затем введите какой-нибудь текст.

   
Рис. 9

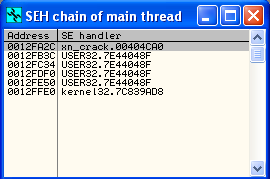
Я снова ввел **«r0 Crew Forever»**, нажал **«Check»** и… И на этот раз оказался где-то в космосе (вы должны быть тут же):

  
   
Рис. 10

Разбираемся с ситуацией… Если мы продолжим выполнение по **«F9»**, то крякми завершит свою работу (об этом позже)! Но почему? Ведь, когда мы его запускали не под отладчиком, все работало!?  
  
Ответ очевиден, мы попались на анти-отладку. Про наличие анти-отладки также говорится в условии к крякми **«немного анти-отладки»**.  
  
Я надеюсь, что вы еще не сдвинулись с адреса **0x00AE0000**? Хорошо. Тогда давайте осмотримся вокруг.  
  
Вначале обратите внимание на **«Status Bar»** он находится в самом низу, под основным окном (Рис. 11.1).

   
Рис. 11.1

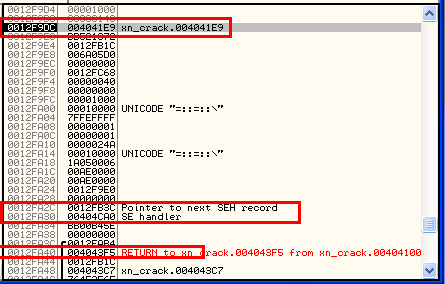
Что мы тут видим? А видим мы следующее, Оля говорит, что сработал брейкпойнт на доступ к памяти, во время выполнения инструкции, расположенной по адресу **0x00AE0000**. Но постойте-ка, какой брейкпойнт, какая память? Что за бред? Единственный брейкпойнт, который мы установили был на **OEP**! Да, вы абсолютно правы, но раз не мы, то кто? Тут-то наше предположение про **«анти-отладку»** и подтверждается. Если бы крякми выполнялся не под отладчиком, то это исключение передалось бы в **«обработчик исключений»**.  
  
Давайте посмотрим на имеющиеся обработчики **«View => SEH chain»** (Рис. 11.2):

  
Рис. 11.2

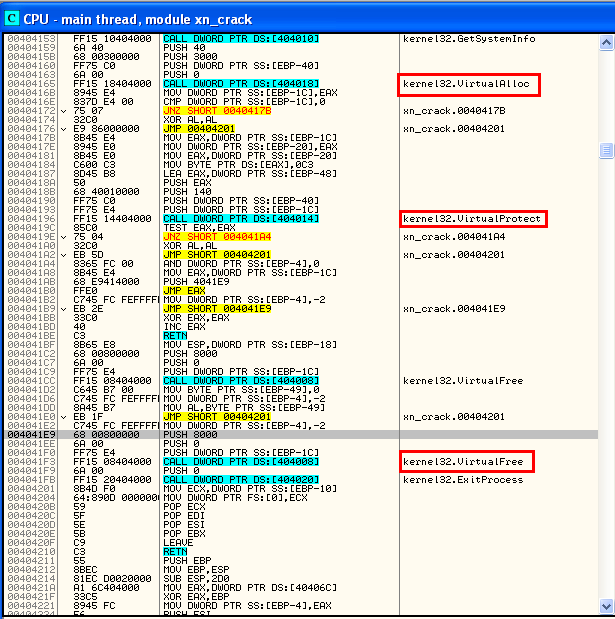
Как мы и предполагали, обработчик имеется, а это значит, что дальнейшее выполнение крякми должно идти через него. И если вы еще ничего не трогали, то мы можем перейти на [**SEH**](http://ru.und3rgr0und.org/wiki/SEH) и продолжить дальнейший анализ уже от туда. Но давайте не будем спешить.  
  
Возможно это не первый анти-отладочный трюк и до того, как мы попали на **0x00AE0000**, нас уже успели обнаружить и направить по неверному маршруту! А если это так, то дальнейший анализ заведет нас в тупик.  
  
Сейчас у нас два пути:

* Двинуться дальше, переключившись на SEH.
* Перезагрузить крякми и пройтись от OEP до 0x00AE0000, чтобы убедиться, что никакой анти-отладки до срабатывания «фальшивого» брейкопйнта не было и мы находимся на верном пути, а не движемся в болото.

В принципе мы можем положиться на удачу и выбрать первый путь. Забегая вперед, скажу, что конкретно в этом крякми удача вас не подведет ;) Но так как мы не ищем легких путей и хотим разобраться с принципом работы, а не способом взлома, то выберем второй путь. Ибо познавший суть вещей становится мастером.  
  
  
***2.1. Анализ ситуации***  
  
Сейчас нам нужно найти и исследовать код, который предшествовал переходу на адрес 0x00AE0000. Прежде чем это сделать, давайте немного раскрутим стэк. Чтобы хоть примерно понять, что происходило до перехода на 0x00AE0000.  
  
***2.1.1. Анализируем стэк***  
  
Давайте обратим внимание на (Рис. 12). Что происходило в стэке до того, как мы оказались там, где оказались?

  
Рис. 12

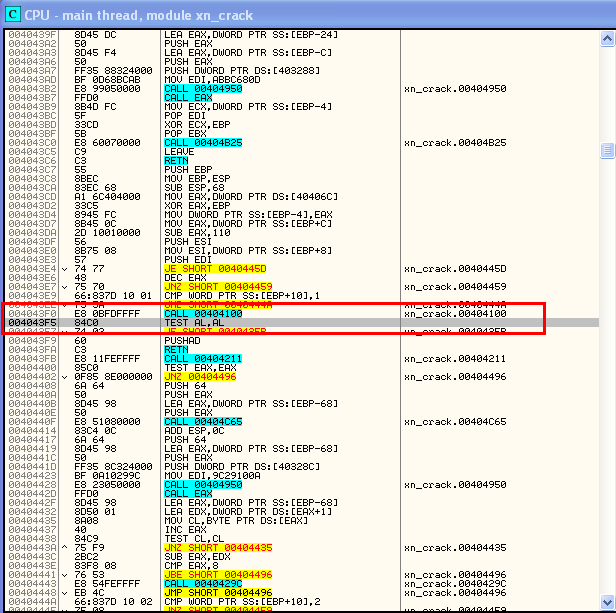
На данный момент **ESP == 12F9DC** и указывает на адрес **0x004041E9**. На него мы перейдем, если продолжим **«нормальное» выполнение** в отладчике. Давайте посмотрим, что нас ожидает, перейдя по этому адресу (Рис. 13). Жмем **«Ctrl+G»** и вводим **«4041E9»**:

  
   
Рис. 13

А ожидает нас следующее. Сразу после перехода осуществляется очистка памяти по адресу **0x00AE0000** (**VritualFree**), после чего крякми завершает свою работу (**ExitProcess**).   
  
Не могут не бросаться в глаза **«подозрительные» функции** (спасибо Оле), которые были обнаружены еще в начале нашего исследования, во время осмотра используемых функций (Рис. 6).  
  
Продолжим осмотр стэка. Нас интересуют все записи (значения), которые соответствуют пределу от **0x00401000** до **0x00413000**, т.е. находятся в секции кода (**UPX0**) исследуемого приложения (см. Alt+M). Среди, ближайших, из таких значений мы находим следующие два:

* **0x00404CA0** – SEH-обработчик на который должно перейти управление, если бы не было отладчика (см. View => SEH chain)
* **0x004043F5** – Ближайшая процедура, из которой мы вышли на адрес 0x00AE0000.

Давайте перейдем на адрес **0x004043F**5 (Ctrl+G и ввести 4043F5) и посмотрим, что она собой представляет (Рис. 14).

  
   
Рис. 14

Что мы видим? В принципе, ничего особенного. Единственное, что мы можем сейчас предположить, так это то, что переход на **0x00AE0000** находится внутри процедуры, расположенной по адресу **0x00404100** и вызываемой по адресу **0x004043F0**. Если вы зайдете в эту процедуру и посмотрите на нее, то вы заметите, что **Рис. 13** является ее частью ;) Так же мы можем сказать, что адрес **0x004043F0** находится внутри процедуры начинающейся с адреса **0x004043C7**. Этот вывод сделан на основании стандартного пролога функции (PUSH EBP/MOV EBP, ESP) выше.  
  
Итак, после анализа стэка мы имеем небольшую цепочку функций вызываемых **до срабатывания «фальшивого» брейкпойнта**. Выглядит она следующим образом:

Code:

Функция (0x004043C7) =>

Функция (0x00404100) =>

Фальшивый брейкпойнт (?) =>

RETN => 0x004041E9 => Exit

SEH => 0x00404CA0 => ?

Сейчас нам нужно определить, что происходило до вызова функции **0x004043C7**. Определять это мы будем идя от **OEP**.  
  
***2.1.2. Анализируем код начиная с OEP***  
  
**Перегружаем крякми** и **становимся на OEP**. Сейчас нам нужно определить план действий.   
  
Вначале давайте решим, что считать началом выполнения программы? Наш крякми довольно маленький, поэтому очевидно, что началом его работы можно считать нажатие на кнопку **«[Check]»**. Отсюда следует, что нам нужно найти **«обработчик кнопки [Check]»**.   
  
Как это сделать? Так как крякми является **«оконным приложением»**, то в первую очередь нужно искать **«обработчик сообщений Windows»** и уже в нем искать нить, ведущую к **«обработчику кнопки [Check]»**.  
  
После того, как мы найдем обработчик кнопки, нам нужно дойти до функции **0x004043C7** и убедиться, что до ее запуска не было никаких анти-отладочных проверок.  
  
Ну, раз план ясен – действуем! Для того, чтобы найти **«обработчик сообщений Windows»** нам нужно найти кусок кода, который отвечает за создание окна. Как его найти? Ну, можно покопаться в сером веществе или в гугле (кто как предпочитает) и вспомнить все способы задания **WndProc**. После чего установить брейкпойнты на соответствующие функции и надеяться, что вы чего-то не забыли.   
  
Мы же будем делать по-другому, например, лень вспоминать/искать – нужное подчеркнуть. Вместо установок брейкпойнтов, воспользуемся способом, который опирается на то, что сразу после вызова функции создания окна – мы теряем управление, т.е:

Code:

PUSH чего-то там

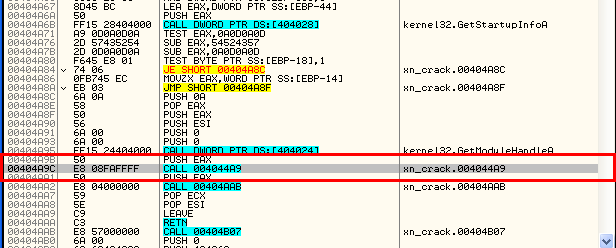
CALL CreateWindow <= После этого мы окажемся

CALL Exit <= не тут

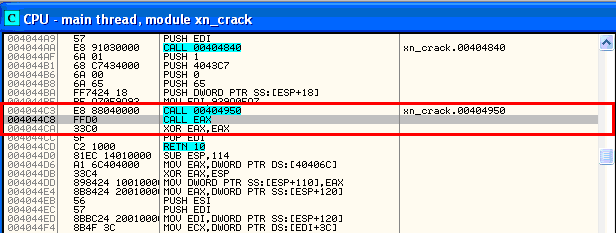
…

Proc WndProc <= A ТУТ!

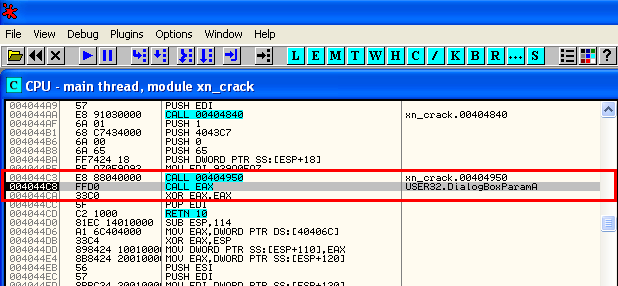
Итак, поехали! Стоя на **OEP** зажимаем кнопку **«F8»** (в некоторых места желательно использовать **«F4»**, буду надеяться, что догадаетесь сами, где и как) и смотрим, где теряется управление (Рис. 15).

  
   
Рис. 15

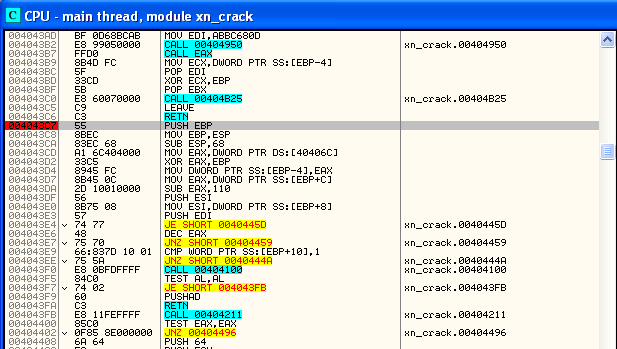
Первый раз мы теряем управление по адресу **0x00404A9C**. Перегружаем крякми, повторяем свои действия и дойдя до этого адреса нажимаем **«F7»**. Далее снова зажимаем **«F8»** (Рис. 16).

  
   
Рис. 16

Второй раз мы теряем управление по адресу **0x004044C8**. Самые внимательные могли заметить, что этот вызов соответствует функции **DialogBoxParamA** (Рис. 17):

  
   
Рис. 17

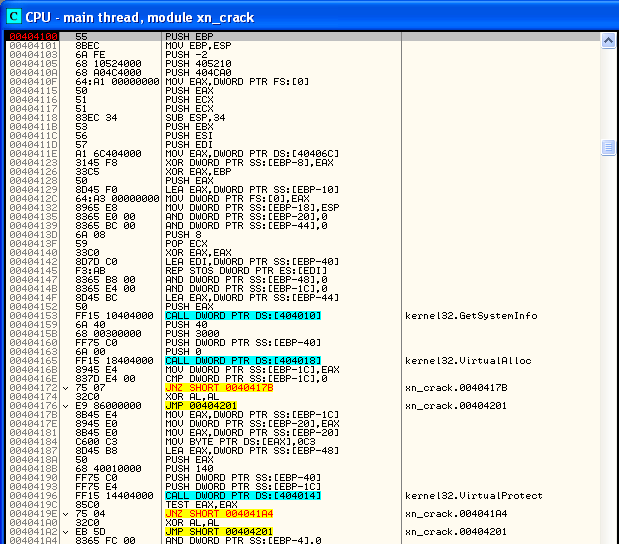
Как раз то, что мы искали! Если вы посмотрите в стэк и сравните переданные функции параметры, то увидите, что **WndProc** соответствует адресу **0x004043С7**. Вам это ничего не напоминает? Верно! Это имена та функция, дойдя до которой, мы собирались убедиться в том, что до срабатывания **«фальшивого» брейкпойнта** не было никакой **другой** анти-отладки. Что же, наша работа немного сократилась =)  
  
Что касается функции **0x00404950**, то она возвращает адрес функции **DialogBoxParamA**.  
  
Теперь, когда мы определили адрес **WndProc**, давайте установим на нее брейкпойнт и нажмем **«F9»**. После чего мы окажемся тут (Рис. 18):

  
   
Рис. 18

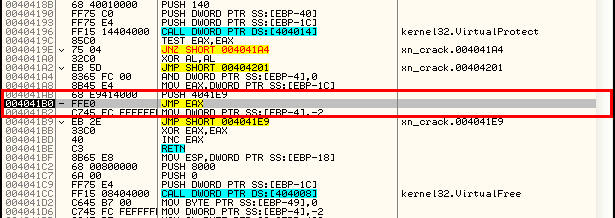
Это место должно быть уже вам знакомо. Чтобы освежить память посмотрите на **Рис. 14**. Проанализировав функцию, я определил, что обработчик кнопки располагается по адресу **0x004043F0** и срабатывает он во время наступления события **WM\_COMMAND** с **wParam = 1**(единица в нашем случае обозначает идентификатор кнопки).  
  
Снимите брейкпойнт с **0x004043C7**. Поставьте брейкпойнт по адресу **0x00404100** и нажимаем **«F9»**. Крякми продолжит свою работу. Затем введите какой-нибудь текст в поле ввода (Рис. 19) и нажмите **«[Check]»**.

   
Рис. 19

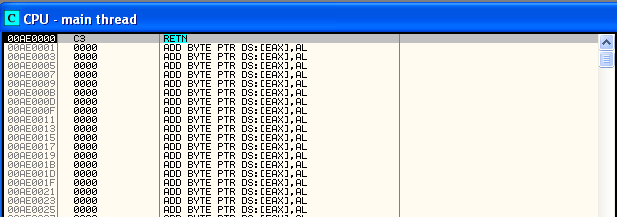
После чего мы окажемся тут (Рис. 20).

  
   
Рис. 20

Эта функция вам так же должна быть знакома. Единственное, что нам тут было неизвестно это то, для чего тут нужны были функции **VirtualAlloc**, **VirtualProtect**, **VirtualFree** и где собственно происходил переход на **0x00AE0000**. Теперь мы знаем, что с помощью этих функций осуществлялся один из известных анти-отладочных трюков, который был **[описанн](http://www.openrce.org/reference_library/anti_reversing_view/20/OllyDbg%20Memory%20Breakpoint%20Detection/" \t "_blank)** еще в 2006 году. Сам же переход происходит по адресу **0x4041B0** (Рис. 21):

  
   
Рис. 21

Снимаем брейкпойнт с **0x004043C7** и жмем **«F9»** (Рис. 22).

  
   
Рис. 22

И мы оказываемся там же, куда мы уже попадали еще на **Рис. 10**. Зачем же мы все это проделали? Если вы забыли, то напомню, «что бы не зайти в тупик». Если бы до первого анти-отладочного приема имелся еще какой-нибудь, то он бы мог направить нас по неправильному маршруту и в итоге завести в какое-нибудь болото.  
  
Сейчас мы имеем следующую цепочку вызовов:

Code:

WndProc (0x004043C7) =>

Обработчик кнопки (0x00404100) =>

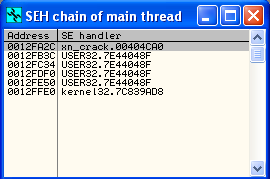
Фальшивый брейкпойнт (4041B0) =>

Анти-отладка (00AE0000):

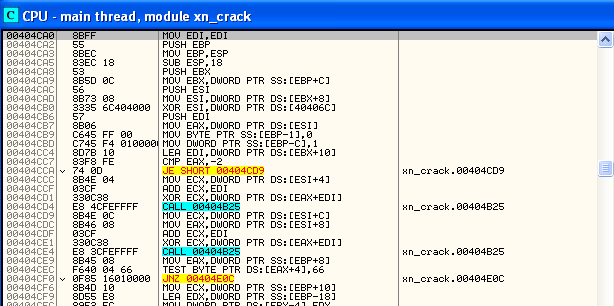
RETN => 0x004041E9 => Exit

SEH => 0x00404CA0 => Продолжение работы

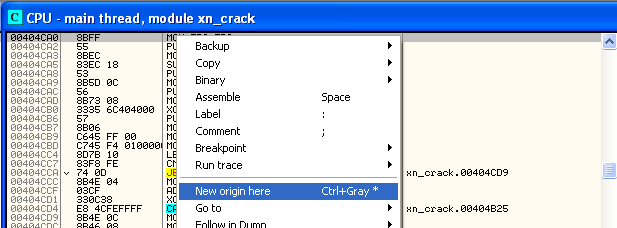
Из которой следует, что дальнейшее выполнение работы программы должно идти через **SEH**. Для этого идем на вкладку **«View => SEH chain»** (Рис. 23):

   
Рис. 23

Далее делаем **двойной щелчок** по самому верхнему обработчику. После чего мы окажемся тут (Рис. 24):

  
   
Рис. 24

Далее устанавливаем курсор на адрес **0x00404CA0**, жмем правой кнопкой мыши и выбираем **«New origin here»**. Это действие переместит указатель команд (регистр EIP) на адрес **0x00404CA0**, что в свою очередь означает, что следующая команда начнет свое выполнения, начиная с указанного адреса (Рис. 25)

  
   
Рис. 25

***2.2. Основной механизм крякми***  
  
Теперь, когда мы выяснили причину возникновения **«фальшивого» брейкпойнта**, можно продолжать дальнейшее исследование крякми.   
  
Сейчас вы должны находиться в SEH. И нам нужно ответить на следующие вопросы.

1. Как крякми получает доступ к строке?
2. Где происходит сравнение введенной строки?
3. Где выводится сообщение, сообщающее о правильном решении?

На эти вопросы мы постараемся ответить в последующих трех разделах.

**ПРИМЕЧАНИЕ:** Начиная с этого места, я не буду подробно описывать свои действия, иначе решение этого крякми затянется на долго. Если вам не будет понятно, почему я пропустил, тот или иной кусок кода, вы всегда можете исследовать и понять его сами, там ничего сложного нет.

***2.2.1. Определение доступа к строке***  
  
Определить код, отвечающий за получение введенной строки, можно поставив брейкпойнты на определенные функции (например, **GetWindowText/GetDlgItemText**). Беря в учет то, что исследуемый крякми (как в принципе и большинство) имеет сравнительно небольшой размер, я пошел другим путем и просто напросто прошел его вдоль и в поперек по **«F7»**.  
  
Результат оказался следующим:

Code:

SEH =>

Функция 404F35 =>

4041СС: VirtualFree – 0x00AE000

404204: Удаляем установленный SEH

Функция 404211 =>

Очистка памяти 404С65 => (с 12F76C до 12FA38)

Очистка памяти 404С65 => (с 12FA4C до 12FAB0)

Функция 404950 =>

Вернуть указатель на функцию 404680 => (eax = GetWindowTextA)

GetWindowTextA => (Edit, буфер=12FA4C)

Функция 40429С =>

Бинго!

12FA18: Great Work!\_

12FA30: Yes!\_

12FA24: gjebaZinw\_

Функция 404950 =>

Вернуть указатель 404680 => (eax = GetForegroundWindow)

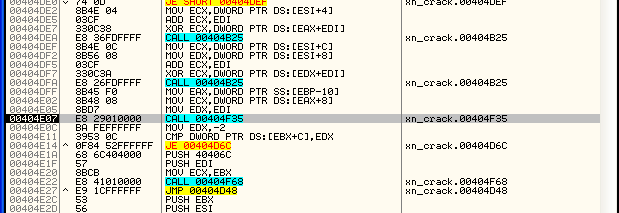
GetForegroundWindow => (eax = текущее окно (Olly))

Функция 404950 =>

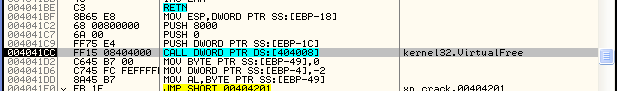
Вернуть указатель 404680 => (eax = GetDlgItemTextA)

GetDlgItemTextA(Olly, Edit, буфер=12F9B4)

Разберемся с полученным результатом более подробно.  
  
Механизм защиты крякми (если так можно сказать) сосредоточен в функции **0x00404F35**, которая вызывается из **SEH** (Рис. 26).

  
   
Рис. 26

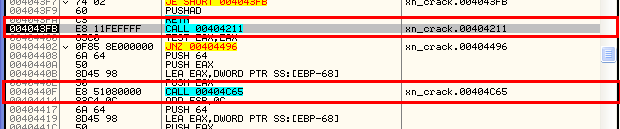
Вначале она осуществляет очистку памяти, ранее выделенную под «фальшивый» брейкпойнт (Рис. 27)

  
   
Рис. 27

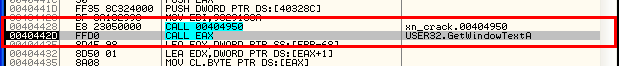
Затем удаляет установленный SEH (Рис. 28)

  
http://www.reverse4you.org/web_files/h-school/Cracking/Crackmes/Manuals/Crackme_04/26.png   
Рис. 28

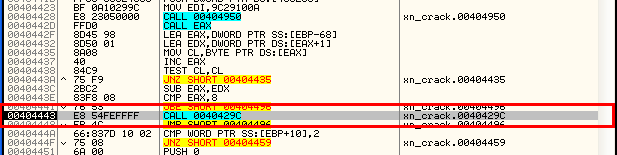
После этого очищает память (с **12F76C** до **12FA38**) и (с **12FA4C** до **12FAB0**) для дальнейшего использования (Рис. 29)

  
   
Рис. 29

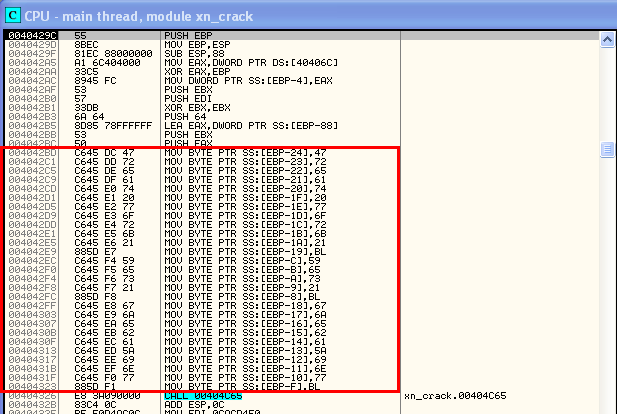
Далее получает указатель на **GetWindowTextA** и вызывает ее с **«дескриптором поля вода»** и указателем на буфер **0x0012FA4C** (Рис. 30):

  
   
Рис. 30

Как выяснится позже – это был **«фейковый» доступ** к введенной строке!  
  
Затем вызывается **«Функция 40429С»** в которой сосредоточен **«основной механизм защиты»** крякми (Рис. 31).

  
   
Рис. 31

Сразу после входа в нее по **«F7»**, ваше внимание должна привлечь следующая последовательность инструкций (Рис. 32):

  
   
Рис. 32

Им соответствует запись очень интересных строк, по следующим адресам:

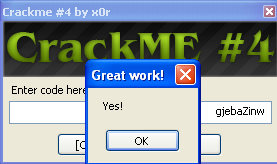
Code:

12FA18: Great Work!\_

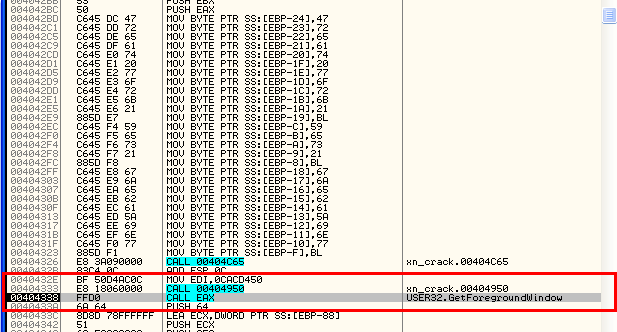
12FA30: Yes!\_

12FA24: gjebaZinw\_

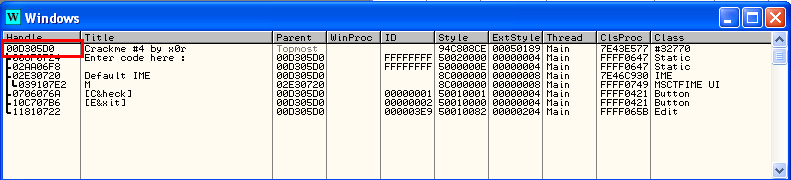
Очевидно, что первые две строки будут использоваться при выводе об успешном вводе пароля. А вот третья строка, возможно и является тем самым паролем!  
  
Запустите крякми (не под отладчиком) и введите **«gjebaZinw»**. Вы увидите следующее (Рис. 33)

   
Рис. 33

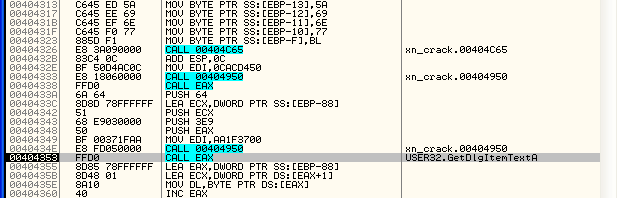
Не спешите закрывать отладчик! Лучше сходите и приготовьте еще чашечку чая.   
  
На самом деле автор этого крякми сильно облажался. На что он надеялся я собственно не понимаю. Как и не понимаю того, за чем был нужен фейковый вызов GetWindowTextA, если сразу за ним идет функция в нутрии которой находятся подобные строки (Рис. 32). Я бы как минимум перенес запись этих строк за вызов **GetForegroundWindow**.  
  
Давайте попробуем оправдать надежды автора! Перегрузите крякми, но уже **под отладчиком**. Введите найденный пароль **«gjebaZinw»** и нажмите **[Check]**.  
  
Ну как, что-то получилось? Пароль сработал? Нет? У меня тоже. Оказывается, это еще не конец. В программе присутствует еще как минимум один метод анти-отладки. И что самое интересное, автор не учел того момента, что единицы будут проверять найденный пароль **«gjebaZinw»**, во время активной сессии отладки, непосредственно в самом отладчике! Если такие вообще будут… А значит, на его трюк практически никто не попадется (тех кто получил пароль на халяву, в расчет не берем).  
  
Проблема, почему крякми отказывается работать под отладчиком, раскрывается **в следующих четырех вызовах**. **Первый** получает указатель на функцию **GetForegroundWindow**, **второй**вызывает ее (Рис. 34)

  
   
Рис. 34

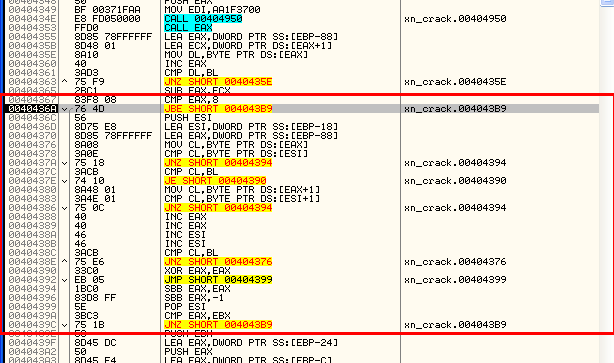
Функция **GetForegroundWindow** возвращает дескриптор приоритетного окна (окна, с которым пользователь в настоящее время работает). Под отладчиком этим окном является окно самого отладчика. Результат своей работы эта функция возвращает в регистре **EAX**.  
  
Собственно таким макром нас и определяют. Чтобы обойти этот фокус, сразу после выполнения **GetForegroundWindow**, измените регистр **EAX**, вписав правильный дескриптор окна. Узнать дескриптор окна крякми можно с помощью **«View => Windows»** (Рис. 35)

  
   
Рис. 35

Далее после вызова **GetForegroundWindow**, идет вызов процедуры, которая возвращает адрес **GetDlgItemTextA**. Затем происходит вызов **GetDlgItemTextA** со следующими параметрами [дескриптор окна, поле ввода=3E9, буфер=12F9B4] (Рис. 36). Здесь **«дескриптор окна» == дескриптору возвращаемому GetForegroundWindow**.

  
   
Рис. 36

В итоге получается, что введенный нами пароль считывается два раза, и что самое интересное двумя разными функциями (дальше разберемся).  
  
С получением строк разобрались – переходим к сравнению.  
  
***2.2.2. Поиск сравнения введенного пароля***  
  
Код сравнения введенной строки располагается сразу за вызовом функции **GetDlgItemTextA**, начиная с адреса **0x00404367** по **0x0040439C** (Рис. 37). Тут же становится ясно, что используется буфер, который заполнила функция **GetDlgItemTextA**, а не **GetWindowTextA**. Отсюда следует, **GetWindowTextA** – это **фейковый вызов**.

  
   
Рис. 37

Цепочка вызовов теперь имеет следующий вид:

Code:

SEH =>

Функция 404F35 =>

4041СС: VirtualFree – 0x00AE000

404204: Удаляем установленный SEH

Функция 404211 =>

Очистка памяти 404С65 => (с 12F76C до 12FA38)

Очистка памяти 404С65 => (с 12FA4C до 12FAB0)

Функция 404950 =>

Вернуть указатель на функцию 404680 => (eax = GetWindowTextA)

GetWindowTextA => (Edit, буфер=12FA4C)

Функция 40429С =>

Бинго!

12FA18: Great Work!\_

12FA30: Yes!\_

12FA24: gjebaZinw\_

Функция 404950 =>

Вернуть указатель 404680 => (eax = GetForegroundWindow)

GetForegroundWindow => (eax = текущее окно (Olly))

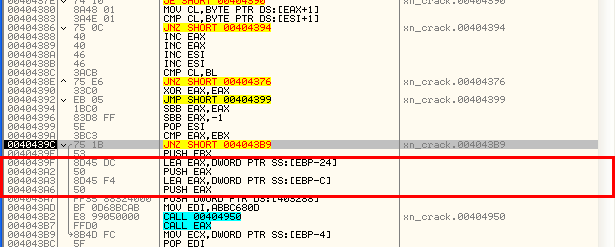
Функция 404950 =>

Вернуть указатель 404680 => (eax = GetDlgItemTextA)

GetDlgItemTextA(Olly, буфер=12F9B4)

+ Проверка (0x00404367 – 40439C)

Переходим к поиску окна выводящего сообщение об успехе.  
  
***2.2.3. Поиск окна выводящего сообщение об успехе***  
  
Сразу после проверки происходит подготовка данных для окна (Рис. 38)

  
   
Рис. 38

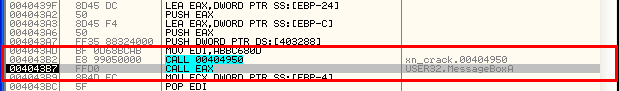
Тут происходит сохранение в стэке следующих строк:

Code:

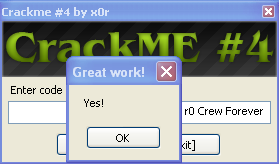
Great Work!

Yes!

Сразу после этого происходит вызов функции, которая подготавливает указатель на **MessageBoxA**. И собственно непосредственный вызов самой **MessageBoxA** (Рис. 39).

  
   
Рис. 39

В результате получаем следующее окно (Рис. 40).

   
Рис. 40

У вас на **Рис. 40** в место **«r0 Crew Forever»** должен быть пароль **«gjebaZinw»**, я просто не перегружал отладчик, как просил сделать вас после **Рис. 33**, а продолжил идти с теми данными, которые уже были.  
  
Теперь можете допить свой чай (если он конечно у вас еще остался) – исследование крякми завершено! Все последующие функции, которые последуют за **MessageBoxA**, просто раскрутят и очистят стек от предыдущих вызовов, после чего будет передано управление в **«обработчик сообщений Windows»** (WndProc).  
  
Окончательная цепочка вызовов будет иметь, примерно, следующий вид:

Code:

SEH =>

Функция 404F35 =>

4041СС: VirtualFree – 0x00AE000

404204: Удаляем установленный SEH

Функция 404211 =>

Очистка памяти 404С65 => (с 12F76C до 12FA38)

Очистка памяти 404С65 => (с 12FA4C до 12FAB0)

Функция 404950 =>

Вернуть указатель на функцию 404680 => (eax = GetWindowTextA)

GetWindowTextA => (Edit, буфер=12FA4C)

Функция 40429С =>

Бинго!

12FA18: Great Work!\_

12FA30: Yes!\_

12FA24: gjebaZinw\_

Функция 404950 =>

Вернуть указатель 404680 => (eax = GetForegroundWindow)

GetForegroundWindow => (eax = текущее окно (Olly))

Функция 404950 =>

Вернуть указатель 404680 => (eax = GetDlgItemTextA)

GetDlgItemTextA(Olly, буфер=12F9B4)

Проверка (0x00404367 – 40439C)

Подготовка строк для MessageBox:

Great Work!

Yes!

Функция 404950 =>

Вернуть указатель 404680 => (eax = ?)

Функция 404880 =>

Формирует указатель на MessageBoxA 4044D0 =>

(eax = MessageBoxA)

RETN

Функция 4046A0 =>

Выход из всех ранее вызванных функций.

MessageBoxA

RETN

RETN

RETN => Windows => WndProc

Успехов =)

*Last edited by root; 08-10-2012 at 17:01.*