Министерство образования Российской Федерации
Санкт-Петербургский политехнический университет

В.Ю. Сальников

Защита информационных процессов в компьютерных системах.

Методические указания по проведению лабораторных работ

Санкт-Петербург
2011

ББК ??.???

УДК ???.???.?? (???.?)

Сальников В.Ю.
Защита информационных процессов в компьютерных системах: Методические указания по проведению лабораторных работ. -2008г. - 23 с.

Данное руководство написано для студентов 5-го курса, изучающих дисциплину «Защита информационных процессов в компьютерных системах» на кафедре ИИТ ФТК СПбГПУ.

Основная цель курса заключается в формировании навыков написания программ на языке C/C++ (возможно, со вставками ассемблера), обладающих элементами защиты от статического изучения и изменения (reverse engineering) и отладки во время выполнения.

Методы, рассматриваемые в курсе, являются базовыми и во многом универсальными.

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ

Защита программного обеспечения – область, в которой все меняется довольно быстро. То, что еще вчера можно было считать надежным методом защиты, сегодня устаревает. Более того, использование устаревших методов создает ложную иллюзию защищенности программы, в то время как методы обхода или снятия защиты могут быть не только широко известны, но и автоматизированы.

Поэтому автором в данном курсе предпринята попытка выделить наиболее универсальные методы защиты авторского программного обеспечения, которые не являются передовыми и уникальными, но простыми и одновременно надежными. Так же в расчет берется сложность создания автоматизированных (без участия человека) методов преодоления защиты.

Изучение курса требует базовых знаний ассемблера x86 (i386+) и хорошего знания языка программирования C++.

Неоднозначное толкование некоторых основополагающих понятий и определений в литературе в области защиты информации и в частности программного обеспечения существенно затрудняет строгое представление некоторых методов. Естественно, что содержание пособия в этом плане отражает субъективный подход автора.

Автор

Лабораторная работа № 1.
Поиск константы и изменение кода.

Целью данной работы является формирование знаний студентов о принципах представления целочисленных констант исходного кода программы в исполнительном модуле, понимание какую информацию стоит скрывать и как это делать наиболее простым методом.

1. Запустите программу 1.EXE с параметром "1998":

1.EXE 1998

Для этого воспользуйтесь командной строкой в Total Commander или запустите консоль Windows “cmd” и в ее окне вводите строку с параметрами (кстати, вводить 1.EXE не обязательно, достаточно ввести 1 , а расширение будет определяться автоматически).

Программа выдаст сообщение "All is Fine !", т.е. будем считать, что проверка прошла успешно.

2. Теперь запустите ее с параметром "1999":

1.EXE 1999

Программа выдаст сообщение "Incorrect usage !", т.е. будем считать, что проверка НЕ прошла.

Допустим, что наша задача заставить программу воспринимать число 1999 как корректное, т.е. следствием нашей работы будет сообщение " All is Fine!" при запуске с параметром 1999. При этом бесполезно делать так, что программа будет всегда выводить указанное сообщение, а необходимо найти саму проверку (т.е. изменить не следствие, а причину).

3. Логично предположить, что где-то в программе имеется проверка введенного числа с числом 1998 (или 1999). Для начала этой информации нам будет вполне достаточно.

4. Прежде чем что-либо делать скопируйте исходный файл в другой, чтобы была возможность восстановить оригинал. Например, 1.EXE в 1A.EXE.

5. Используйте программу hiew.exe и загрузите нашу программу:

HIEW.EXE 1.EXE

На экране представлен исполнительный код в интерпретации ASCII символами.

6. Нам больше подходит режим 16-тиричного представления. Перейдем в него последовательным нажатием F4 и F2 (подсказка внизу экрана).

Теперь каждый байт представлен двумя символами от 00 до FF (256 градаций).

Слева также в 16-тиричном режиме представлено смещение самого левого байта от начала файла. В каждой строке - 16 байт, поэтому адреса все имеют последний символ 0. Черточки стоят после 4-го, 8-го и 0Ch (12-й) байтов.

Для EXE файла образ кода начинается с некоторого смещения от начала файла и выровнен на границу 256 байтов (100h). Кроме того hiew автоматически учитывает настройки, указанные в PE заголовке программы и показывает нам сразу адрес исполняемого кода как будто он загружен в память для исполнения. В данном случае код программы начинается с адреса 256*16, т.е. с 1000h. Это легко заметить, пролистав файл до этого адреса. Несмотря на то, что после адреса 0FFFh должен идти адрес 1000h, мы видим адрес .401000h. Обратите внимание, что если отображаемый адрес равен смещению от начала файла, то число написано БЕЗ точки в начале. Если не совпадает, то это виртуальный адрес в момент выполнения и пишется он с точкой. С этого адреса и начинается код, в котором мы будем искать константу.

7. Теперь необходимо перевести число 1998 в 16-тиричную систему. Можете попробовать сами - 07CEh. Так как программа под Windows, то можно ожидать, что число сравнивается/храниться как 32-х разрядное (4 байта). А значит настоящее число скорее всего не 07CEh, а 000007CEh.

8. Будем искать теперь это число. По F7 в поле Hex введите последовательно "CE 07 00 00" (байты хранятся в обратном порядке). Число скорее всего int, но если short, то не будет 2-х нулевых байтов. Такое сочетание байтов может быть в случайном месте, поэтому стоит найти все совпадения в файле. Так как в файле такое вхождение только одно, то скорее всего это оно и есть.

9. Теперь, не отходя от места вхождения байтов, перейдите в режим дизассемблирования - F4 и F3. Возможно необходимо несколько "отъехать" от места вхождения (байтов на 10 назад, т.е. вверх).

На месте вхождения байтов будет следующая строка:

```
.0040101B: 3DCE070000  cmp  eax,0000007CE
```

Как видно, это очень похоже на то, что мы искали. Сравнивается некоторое двойное слово в регистре EAX с нашим числом - 1998.

После операции сравнения устанавливаются флаги результата, которые затем определяют поведение следующей команды условного перехода.

Действительно, за нашей командой следует условный переход:

```
.00401020: 7E3E        jle  .000401060
```

Это переход, если меньше или равно, т.е. в нашем случае x<=1998. Очевидно, что "хорошее" поведение программы заключается в выполнении данного условия и передачи управления по адресу .000401060.

В противном случае ("плохое" поведение) выполняется следующий оператор после операции условного перехода.

10. Теперь все готово к изменению кода в "нужном" нам направлении. Основная мысль в том, чтобы программа всегда осуществляла переход при любом условии.

Этого можно добиться несколькими способами.

А) Самый простой способ поменять число с 1998 на 2998. В данном случае это удобно.

Б) Убрать проверку вообще.

В) Заменить условный переход на безусловный.

11. В способе А все просто:

- Перейдите в режим редактирования – клавиша F3.
- Подведите курсор к нашим байтам "CE070000" в теле команды.
- Введите новое значение, но не больше 7FFFFFFFh, иначе число станет отрицательным (не забудьте как число должно храниться).
- После этого нажмите F9 (запись изменений в файл).

12. Во втором способе Б самое удобное ввести сразу переход в нужное место, или ввести многократно код "нет операции" (90h). Для этого:

- Выберем начальный и конечный адреса для ввода - .00401022h и .00401060 (сами догадайтесь почему);
- Перейдите в режим редактирования F3;
- Подведите курсор к байту по адресу .00401022h (обратите внимание, что адрес изменился на 1022h - в режиме редактирования отображаются не виртуальные адреса, а реальные смещения от начала файла);
- Последовательно вводите байты 90h, пока не достигнете адреса 105Fh (включительно). Во время ввода можно не обращать внимание на то, что код после места ввода изменяется (дизассемблирование на лету пытается «разобраться» в изменяемом вами коде);
- После этого нажмите F9 (запись изменений в файл).

13. В третьем способе В:

- Перейдите в режим редактирования F3;
- Переместите курсор к команде условного перехода;
- Нажмите табуляцию. Появится окно ввода ассемблерной команды;
- Вместо команды "jle" введите команду "jmp", а адрес перехода не меняйте. Т.к. в нашем случае длина команды безусловного перехода оказалась длиннее команды условного, то последующий код исказился. Нас это не должно волновать, т.к. этот код не получит управления никогда. При большом желании можно его забить НОПами до адреса 105Fh (включительно), или ввести свои инициалы. В случае перехода в пределах

+127 ... -128 байтов переход может быть и коротким. Мнемоника команды будет не "jmp", а "jmps". В нашем случае подходят оба варианта;

- После этого нажмите F9 (запись изменений в файл).

14. Попробуйте все 3 варианта, восстанавливая каждый раз оригинал файла, копируя 1A.EXE в 1.EXE . При этом вам понадобится адрес для редактирования, чтобы не искать каждый раз нужное место. Запомните его и при очередном изменении используйте переход непосредственно на нужное смещение в файле с помощью F5.

15. Определите, какой метод вам нравится больше всего.
16. Исходный текст программы:

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char* argv[]){
  if(argc<2) return -1;
  int v=atoi(argv[1]);
  if(v>1998){
    puts("Incorrect usage !");
    getchar();
    return -2;
  }
  puts("All is Fine !");
  getchar();
  return 0;
}
```

17. Задание. Попробуйте осуществить те же действия над программой 2.EXE, которая делает все то же самое. В исходный код на Си внесены минимальные изменения с целью простой защиты от данного вида поиска. При этом не рекомендуется использовать при поиске информацию о структуре файла 1.EXE (считайте, что его нет). Используйте метод, описанный выше, сведения из лекционного материала и включите свое воображение.

18. По результатам напишите отчет, где отразите все ваши манипуляции по поиску необходимой информации в файле 2.EXE (без ссылок на файл 1.EXE).

В выводах напишите насколько хорош или плох метод, примененный в файле 2.EXE с целью защиты. Напишите как по вашему был незначительно изменен исходный текст программы 1 в программе 2.

Также напишите ваши мысли по поводу пункта 15.

Лабораторная работа № 2.
Поиск кода по сообщению программы.

По результатам данной работы студенты получат знания о том как связаны в исполнительном модуле код и сообщения, какая информация о сообщениях представлена в коде, где сами сообщения хранятся и как скрыть использование сообщений.

1. Для примера используем программу : 2.EXE

2. В случае неудачи как и в первом задании программа выдает сообщение "Incorrect usage !". Попробуем отталкиваться от этого.

3. Прежде чем что-либо делать скопируйте исходный файл в другой, который и будете изменять. На пример 2.EXE в 2A.EXE

4. Используем программу hiew.exe и загрузим нашу программу:
HIEW.EXE 2.EXE

5. Нам больше подходит режим 16-тиричного представления. Обратите внимание, что с правой стороны все байты представлены в виде ASCII символов, что поможет нам в поисках строки.

6. По F7 ищите желаемую строку сообщения в режиме ASCII (можно по нескольким первым символам).

7. Так как в Windows применяется FLAT модель, то и сегмент кода и сегмент данных и стека все перекрывают общее линейное адресное пространство. А это значит, что, например, адрес данных 12345h будет точно таким же при доступе к данным через любой сегментный регистр (ну почти любой).

8. Теперь посмотрим, откуда начинается строка “Incorrect usage !”. Hiew показывает нам адрес .407030 (с точкой, значит виртуальный адрес на момент выполнения). Нажав Alt+F1 можно перейти в режим отображения смещения в файле. В нашем случае это 7030, хотя нам это не понадобится.
Значит, если строка используется в коде, то в этой команде в качестве аргумента должен присутствовать адрес 00407030.

9. Далее процедура вам знакома. Найдите вхождение байтов "30", "70", "40" и "00" в коде. Вхождение одно по адресу команды .401018, однако проблема может быть в том, что вывод сообщения может быть значительно удален от места проверки условия, которое нам нужно в действительности. Поэтому поиск в обратной последовательности может быть затруднен, если

проверка и вывод сообщения находятся в различных функциях. В этом случае нам больше подойдут другие методы, которые мы рассмотрим позже.

10. Пока попробуем обойтись средствами Hiew. Как вы помните в команде вызова функции CALL указывается не абсолютный адрес функции, а относительный (расстояние от адреса назначения до текущего адреса). Вручную искать место вызова функции довольно сложно, но процесс можно автоматизировать, т.к. алгоритм очевидный.

Это и сделано в Hiew. Встаньте на предполагаемое начало функции и нажмите клавишу F6. Hiew будет искать первое место вызова функции в программе. Если вы хотите найти следующие вхождения, то далее следует нажимать Ctrl+F6.

Так и поступим. Очевидно, что выбор сообщения для вывода определяется значением параметра функции в [esp][04]. Он сравнивается с нулем. Это явно не 1998, поэтому найдем начало функции - .401000, поставим туда курсор и нажмем F6. Hiew отправит нас в .401035 и действительно там можно видеть вызов нашей функции:

.00401035: E8C6FFFFFF call .000401000 -----↑ (1)

Обратите внимание на цифру 1 справа в скобках. Если вы нажмете 1, то перейдете прямо к вызываемой функции.

Вызов функции найден, но и тут нас ждет разочарование - сравнения нет. Видно что и данная функция вызывается и проверка, видимо, раньше. Проверяем. Встаем на .401030, нажимаем F6, попадаем в .401065 и тут как раз видим сравнение чуть выше в .40105C (использована простая защита от поиска констант как и в первом задании).

Задача выполнена - проверка найдена. Что делать дальше вы уже знаете.

11. Вот код программы на C:

```
void message(int index){
  switch(index){
    case 1: puts("Incorrect usage !"); break;
    case 2: puts("All is Fine !"); break;
  }
}
void dosomething(int v){
  int t;
  t=v; message(t);
}
void main(int argc, char* argv[]){
  int v;
  if(argc<2) return;
  v=atoi(argv[1])-1000;
  if(v>998){
    dosomething(1);
    getchar();
    return;
  }
  dosomething(2);
  getchar();
}
```

12. Задание. Попробуйте осуществить те же действия над программой 3.EXE, которая делает все то же самое. В исходный код на Си внесены минимальные изменения с целью защиты от данного вида поиска (плюс немного мусорного кода, чтобы программа не была очень похожа на 2.EXE). В случае успеха предположите как выглядит исходный код программы на C в части вывода сообщения на экран.

Придумайте пример программы, где возможность поиска точек вызова функция по F6 мало помогает (чем проще решение - тем лучше).

Отразите все результаты в отчете.

Лабораторная работа № 3.
Поиск кода путем обратной трассировки стека во время выполнения.

1. Для работы используйте программу: 3.EXE (поведение программы как и в предыдущих заданиях).

2. Для работы нам необходим отладчик. Например, Turbo Debuger TD32.EXE как наиболее простой, компактный, не требующий установки и, в то же время, вполне достаточный в нашем случае. Для загрузки под ним программы необходимо набрать следующую строку:

```
TD32.EXE 3.EXE 1999
```

Здесь первый параметр - имя исполняемой программы, а второй – параметр для нашей исполняемой программы (аргумент в командной строке).

3. Для поиска можно использовать те же действия, что и ранее (см. работу 2), но поиск осуществлять непосредственно в оболочке TD32.EXE.

Предположим, вы самостоятельно рассмотрели программу в HIEW и обнаружили что сообщение “Incorrect Usage!” начинается с адреса .407030 (можете проверить). Следовательно, непосредственное использование данной строки означает упоминание ее адреса. Значит, будем искать в загруженном образе программы число 00407030. Если число для поиска большое (как в нашем случае) TD32 автоматически разобьет его на байты, поэтому нет необходимости указывать их в обратном порядке, однако, если вам это необходимо, то перечисляйте байты через запятую (в нашем случае: 30h, 70h, 40h, 0).

Есть еще одно действие, которое стоит сделать до поиска. Т.к. TD32 сразу Показывает точку входа в программу, а ссылка на строку теоретически может быть как позже, так и раньше точки входа, то надо переместить область просмотра в начало кода программы (т.к. поиск идет с текущего адреса).

Перейдем на начало программы - нажимаем Cntrl+G и вводим адрес 00400000. Команда поиска в TD32 - Cntrl+S. Введите значение 407030. Код находится по адресу 401019. Вспомним, что адрес это параметр команды, А начало ее раньше, “отодвинемся” на несколько байтов выше. Видим:

```
00401018   6830704000  push    00407030
0040101D   E80C010000  call    0040112E
00401022   83C404      add     esp,00000004
00401025   EB0D        jmp     00401034 (00401034)
```

Очень похоже, что это вывод сообщения.

4. Поставим точку останова на адресе 401018. Для этого поместите курсор на строку с этим адресом и нажмите F2. При перемещении курсора строка будет оставаться выделенной (красным цветом).

5. Все готово для запуска. Если вы забыли указать аргумент в командной строке при загрузке отладчика, то это можно сделать сейчас через меню:
Run->Arguments... (отладчик предложит перезагрузить программу).

Так же можно поменять аргумент без выхода из отладчика.

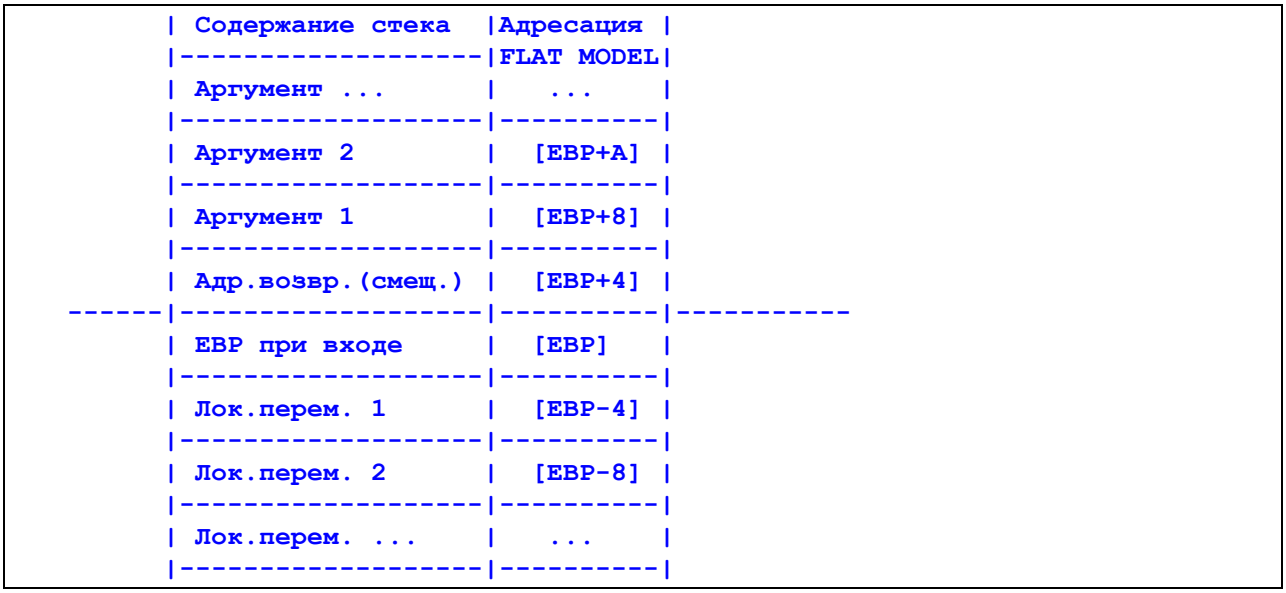
Запустите программу на выполнение - F9.

6. Если все было сделано правильно, то сработает установленная точка останова (появится треугольник в этой строке после адреса).

7. Теперь мы имеем нужный нам кадр стека, по которому и будем осуществлять обратную трассировку. Содержание стека представлено в правом нижнем углу окна отладчика. Так как стек растет сверху вниз, то нам нужны значения в стеке, начиная с треугольника (текущее значение ESP) и выше.

Перемещение между полями окна отладчика - Tab.

8. Для большинства языков высокого уровня свойственно использование регистра EBP для доступа к аргументам функции и ее локальным переменным. Это естественно и удобно (так оно и было задумано), т.к. адресация [EBP] берется по умолчанию относительно SS, т.е. в стеке (так же как и для ESP). Кадр стека внутри функции выглядит следующим образом:



Для сохранения значения EBP неизменным внутри функции (для реализации адресации в стеке) используются следующие команды при входе в функцию:

```
push ebp
mov  ebp,esp
```

При выходе из функции значение EBP восстанавливается из стека:

```
pop  ebp
ret
```

Таким образом, значение EBP восстанавливается на оригинальное при выходе из функции и остается неизменным внутри ее. Из этого следует, что для всех функций, которые на данный момент активны (вошли, но не вышли) в стеке имеется оригинальное значение EBP. Исходя из этого можно определить местоположение адреса возврата из функции, а следовательно и точку вызова. Более того, если посмотреть внимательно, то текущее значение EBP указывает на адрес в стеке, который содержит значение EBP для объемлющей функции (из которой вызывалась данная), которое, в свою очередь, указывает на EBP для функции, из которой вызывалась эта и т.д. Если допустить, что все функции используют такой способ адресации, то по текущему значению EBP легко прослеживается вся цепочка вызовов. Проверим это.

9. Если переместиться по коду вверх и вниз, то можно найти те самые "магические" команды работы с EBP. Кстати, по ним можно определить и точку входа в функцию, которую иначе можно и не найти вовсе.

10. Значение EBP = 12FF58 (может быть разным на разных машинах, но это не важно). Если оно равно ESP, то очевидно, что функция не содержит локальных переменных. Для каждой функции количество слов, помещенное в стек перед вызовом функции фактические аргументы вызываемой функции) равно количеству извлекаемому из стека после возврата. Для Си/Си++ слова извлекаются из стека в вызывающей функции либо "бесполезными" командами (если количество слов мало)

```
pop ecx,
```

или путем коррекции ESP на необходимую величину

```
add esp,10
```

Для большинства других языков (PASCAL,FORTRAN,BASIC...) слова извлекаются из стека в момент возврата из вызываемой функции или процедуры, путем использования команды возврата с коррекцией стека:

```
ret 10
```

Можно заметить, что для Си о количестве помещенных в стек параметров "обязана" знать лишь вызывающая функция, а для других как вызывающая так и вызываемая. Именно поэтому в Си возможны функции с переменным числом параметров, а в других языках - нет. По этой же причине в Си параметры помещаются в стек в обратном порядке (сами подумайте почему).

11. В стеке у нас следующие значения (опять - таки смещения стека и значения ESP и EBP могут быть ДРУГИМИ, т.к. это зависит от машины и все в стеке имеет относительную, а не абсолютную природу, однако, принцип абсолютно [каламбур] такой же):

```
          Стек              Функция Комментарий
       0012FF8C 003211C0  |
       0012FF88 00000002  |
       0012FF84 00401471  |    3     <-- Адрес возврата в пред. функцию
     ->0012FF80 0012FFC0--    3     <-- Обратная трассировка EBP
    |  0012FF7C 000009F8      3     <-- Локальная переменная в функции
    |  0012FF78 0012EF50      3     <-- Локальная переменная в функции
    |  0012FF74 00000004      3     <-- Локальная переменная в функции
    |
    |  0012FF70 00000001      2     <-- Фактический аргумент для функции
    |  0012FF6C 0040108A      2     <-- Адрес возврата в пред. функцию
     --0012FF68 0012FF80<-    2     <-- Обратная трассировка EBP
       0012FF64 00000001  |    2     <-- Локальная переменная в функции
                          |
       0012FF60 00000001  |    1     <-- Фактический аргумент для функции
       0012FF5C 0040104B  |    1     <-- Адрес возврата в пред. функцию
 EBP-> 0012FF58 0012FF68--    1     <-- Обратная трассировка EBP
 ESP-> 0012FF54 00000001
```

Здесь все соответствует. Возникает лишь одна тонкость - как отличить в стеке локальную переменную вызывающей функции от фактического параметра для вызываемой функции? Предлагаю вам подумать самостоятельно.

Итак, получаем как минимум два адреса возврата в стеке: .40104B и .40108A. Посмотрим что там.

12. Перейдите по адресу .40104B. Для этого перейдите в поле CPU и через Cntrl+G задайте нужный адрес. Код выглядит так:

```
     00401038 55             push   ebp
     00401039 8BEC           mov    ebp,esp
     0040103B 51             push   ecx
     0040103C 8B4508         mov    eax,[ebp+08]
     0040103F 8945FC         mov    [ebp-04],eax
     00401042 8B4DFC         mov    ecx,[ebp-04]
     00401045 51             push   ecx
     00401046 E8B5FFFFFF     call   3.00401000
---> 0040104B 83C404 ------> add    esp,00000004
     0040104E 8BE5           mov    esp,ebp
     00401050 5D             pop    ebp
     00401051 C3             ret
```

Как видно адрес действительно указывал на команду, следующую за командой вызова процедуры. Тут можно заметить, сколько слов извлекается из стека после вызова - одно. Значит, параметр был один. Если посмотреть на код до вызова функции, то видно, как копируется формальный параметр (смещ. от EBP положительное) [BP+8] в локальную переменную (смещ. от EBP отрицательное) [EBP-4]. Внимательное изучение кода функции говорит о том, что никакой проверки в данной функции не производится, поэтому необходимо двигаться дальше - к предыдущей вызывающей функции и проверить как параметр формируется там.

13. Рассмотрим адрес 40108A. Код следующий:

```
    00401069 51             push   ecx
    0040106A E843030000     call   3.004013B2
    0040106F 83C404         add    esp,00000004
    00401072 0529020000     add    eax,00000229
    00401077 8945FC         mov    [ebp-04],eax
    0040107A 817DFCF7090000 cmp    dword ptr [ebp-04],000009F7
    00401081 7E56           jle    3.004010D9 (004010D9)
    00401083 6A01           push   00000001
    00401085 E8AEFFFFFF     call   3.00401038
--->0040108A 83C404 ------->add    esp,00000004
```

Теперь видно, что условие вызова функций и сама проверка находятся именно в этой функции чуть ранее.

14. Осталось внести необходимые изменения. Так как в TD32 нельзя записать образ программы обратно на диск (почему - догадайтесь сами), то необходимо "запомнить" несколько последовательных байтов до или после "нужного" места и искать их затем в любом 16-тиричном редакторе, например в HIEW. Следует лишь запомнить, что среди этих байтов не должно быть настраиваемых адресов (сегментных). Это актуально только для FAR модели кода. Для FLAT модели, используемой в Windows это не актуально. Возможна так же настройка и ближних адресов в случае, если образ программы грузится не по тому адресу, что подразумевался при компиляции. Но ведь в нашем случае все нормально – адрес загрузки образа 00400000, а значит, проблем быть не должно.

Проделайте модификацию кода любым известным вам способом самостоятельно.

15. Заключительные комментарии и задание.

Иногда удобнее в каждой функции "доходить" до ее конца (инструкции ret) или до ее начала, чтобы рассчитать действительное значение адреса возврата. Это актуально, если возникают проблемы с трассировкой по EBP.

В качестве примера попытайтесь проделать все вышесказанное для файла 3A.EXE, найти место сравнения и изменить код таким образом, чтобы верхняя граница поднялась с 1998 до 2010 года.

Отразите процесс поиска и его результат (для файла 3A.EXE) в отчете.

В программе применена несложная защита от трассировки стека. Ваша задача обнаружить ее и отразить это в отчете.

Дополнительно ответьте на вопросы:

А) Почему в пункте 3 мы переходим на адрес 00400000, а не какой-то другой?

Б) Почему в языке Си могут существовать функции с переменным числом параметров, а, например, в BASICe нет?

В) Почему в Си параметры, передаваемые в функцию, помещаются в стек в обратном порядке?

Г) Как при трассировке стека отличить фактические параметры функции от локальных переменных вызывающей ее функции?

Д) Предположите, как может выглядеть код простой защиты от трассировки стека в программе 3A.EXE?

Лабораторная работа № 4.
Динамическое дешифрование.

1. Для примера используйте программу : 4.EXE (поведение как в пред. заданиях).

2. Для работы вам необходим отладчик. Опять используйте Turbo Debugger TD32.EXE. Наберите следующую строку:

TD32.EXE 4.EXE 1999

Здесь первый параметр - имя исполняемой программы, а второй – параметр для вашей исполняемой программы (аргумент в командной строке).

3. Посмотрите, что находится внутри программы. Начальная позиция в программе 4013F0. Здесь следуют различные настройки программной среды, которые выполняются до пользовательского кода, т.е. до функции main. Нам сейчас это не очень интересно, поэтому перейдите непосредственно к функции main. Можно выполнять код последовательно, а можно сразу перейти на 401394 (Ctrl+G). Это вызов функции main. Непосредственно до этого адреса находятся три инструкции PUSH (догадайтесь зачем - напишите в отчете) и затем CALL (это и есть вызов main).

Выполните программу до этого места (поместив курсор на эту строку, нажмите F4).

Войдите в функцию main (F7).

4. Начало функции стандартно. Далее идет вызов некоторой функции

00401074 E887FFFFFF call 00401000

Если посмотреть дальше, то там достаточно абсурдный набор инструкций. Можно предположить, что этот код зашифрован, а вызываемая функция - дешифровщик.

5. Для того чтобы убедиться, что дешифровщик работает, запустите программу на выполнение (F9). Не двигаясь с начала блока, можно заметить, что код явно изменился. Именно сейчас после завершения программы можно понять, как она выглядит "в оригинале", т.е. в момент выполнения. Т.к. вы уже знаете что и как проверяется, то сразу можете заметить следующую инструкцию:

004010A9 817DFCCE070000 cmp dword ptr [ebp-04],000007CE

Знакомое число 1998.

Будем вносить изменения. Заменим число 0x000007CE на значительно большее, причем так, чтобы менялся лишь 1 байт. Например, превратим число в 0x00FF07CE (меняем 2-й байт в числе). После таких изменений все должно работать как требуется. Посмотрите еще раз на инструкцию. Синим выделен тот самый байт. Его адрес 004010AE.

Однако как внести изменения так, чтобы после дешифрации получить нужное число?

6. Для этого надо рассмотреть сам дешифратор. Вы уже знаете, что он вызывается непосредственно перед шифрованным блоком. Перейдите на начало функции дешифровки, т.е. на 00401000. Давайте его разберем:

```
00401000 55              push    ebp
00401001 8BEC            mov     ebp,esp
00401003 83EC10          sub     esp,00000010
  Стандартный пролог.

00401006 C745FC00000000  mov     dword ptr [ebp-04],00000000
  Инициализация локальной переменной в 0.

0040100D 8B4504          mov     eax,[ebp+04]
  Вспомните, что по [ebp+04] хранится адрес возврата из функции, т.е. адрес
следующего байта за вызовом CALL.

00401010 8945FC          mov     [ebp-04],eax
00401013 8B45FC          mov     eax,[ebp-04]
  Сохраняем и опять читаем (по секрету скажу, что оптимизация отключена).

00401016 8A08            mov     cl,[eax]
00401018 884DFB          mov     [ebp-05],cl
  Интересно, по адресу после CALL читается 1 байт и сохраняется.

0040101B 8B55FC          mov     edx,[ebp-04]
0040101E C60290          mov     byte ptr [edx],90
  А теперь по тому же адресу записывается байт 0x90. Если вспомните, то это
код инструкции NOP.

00401021 C745F401000000  mov     dword ptr [ebp-0C],00000001
00401028 EB09            jmp     4.00401033 (00401033)
  Похоже на цикл и [ebp-0C] - счетчик итераций.

0040102A 8B45F4          mov     eax,[ebp-0C]
0040102D 83C001          add     eax,00000001
00401030 8945F4          mov     [ebp-0C],eax
  Вот и инкремент счетчика

00401033 0FB64DFB        movzx   ecx,byte ptr [ebp-05]
00401037 394DF4          cmp     [ebp-0C],ecx
0040103A 7F25            jg      4.00401061 (00401061)
  Обратите внимание. Значение счетчика цикла сравнивается с тем самым
байтом, который мы извлекли и заменили на NOP. Значит первый байт в
зашифрованной последовательности – количество зашифрованных элементов.

0040103C 8B55FC          mov     edx,[ebp-04]
0040103F 0355F4          add     edx,[ebp-0C]
00401042 8A02            mov     al,[edx]
  К адресу начала блока прибавляется счетчик (его начальное значение 1) и
читается зашифрованный байт.

00401044 8845F3          mov     [ebp-0D],al
  Временно сохраняем.

00401047 0FB64DF3        movzx   ecx,byte ptr [ebp-0D]
  Опять читаем.

0040104B 81F1FF000000    xor     ecx,000000FF
```

```
    Вот оно! Прочитанный байт ксорится с байтом 0xFF, что меняет его значение
на противоположное.

  00401051 884DF3          mov     [ebp-0D],cl
    Вновь временно сохраняем.

  00401054 8B55FC          mov     edx,[ebp-04]
  00401057 0355F4          add     edx,[ebp-0C]
    Опять к адресу начала блока прибавляется счетчик, т.е. адрес этого байта.

  0040105A 8A45F3          mov     al,[ebp-0D]
  0040105D 8802            mov     [edx],al
    А теперь сохраняем байт обратно.

  0040105F EBC9            jmp     4.0040102A (0040102A)
    Окончание цикла.

  00401061 8BE5            mov     esp,ebp
  00401063 5D              pop     ebp
  00401064 C3              ret
    Эпилог и возврат.
```

7. Итак, теперь ясно, что сразу за вызовом функции дешифровщика располагается байт размера зашифрованной области в байтах, а затем и сами зашифрованные XORом с 0xFF байты.

Таким образом можно сделать вывод, что алгоритм шифрования довольно прост и на самом деле является кодированием (каждому байту в соответствие ставится уникальный байт). Это сильно упрощает задачу внесения изменений, т.к. для изменения одного байта необходимо изменить именно этот сам байт и больше ничего. На что менять? Наш байт 0xFF и если использовать XOR с 0xFF, то получим 0.

8. Все готово для внесения изменений. Скопируйте файл 4.exe в 4a.exe и откройте его в heiw. Перейдем прямо к нужному байту. Адрес .4010AE - там находится байт 0xFF (напишите в отчете почему именно 0xFF). Замените его на 0 и сохраните файл.

Теперь запустите программу 4a.exe под отладчиком (TD32.EXE 4A.EXE 1999) и убедитесь что изменения присутствуют и программа работает как ожидается.

9. Вот код программы.

```
  #include <stdlib.h>
  #include <stdio.h>

  typedef unsigned char BYTE;

  void Decode(void){
    BYTE *Buf=0;
    __asm{
      mov eax,[ebp+4]
      mov Buf,eax
    }
    BYTE size=Buf[0];
    Buf[0]=0x90;
```

```
    for(int i=1;i<=size;i++){ BYTE v=Buf[i]; v^=0xFF; Buf[i]=v; }
}

int main(int argc, char* argv[]){
  Decode(); __asm nop;
  if(argc<2){ puts("No Parameter !"); getchar(); return -1;}
  int v=atoi((char *)argv[1]);
  if(v>1998){ puts("Incorrect usage !"); getchar(); return -2;}
  puts("All is Fine !");
  getchar();
  return 0;
}
```

А вот текст программы шифровальщика.

```
#include <windows.h>

struct _block{
  int Start; // size,data
  int Stop;  // first byte after data
  int Code;  // type of coding 1-XOR 0xFF
} CAr[]={{0x479,0x4DF,1}};
int WriteEnable=0x1FF; // установка прав на запись в кодовую секцию

int main(void){
   HANDLE h=CreateFile("test.exe",GENERIC_WRITE | GENERIC_READ, 0, NULL,
   OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
   if(h==0) exit(-1);
   int sz=GetFileSize(h,NULL);
   if(sz==INVALID_FILE_SIZE) exit(-2);
   BYTE *Buf=0;
   Buf=new BYTE[sz];
   if(Buf==0) exit(-3);
   DWORD br=0;
   int ret=ReadFile(h,Buf,sz,&br,NULL);
   if(ret==0) exit(-4);
   if(br!=sz) exit(-5);
   Buf[WriteEnable]=0xC0;
   int bnum=sizeof(CAr)/sizeof(_block);
   for(int n=0;n<bnum;n++){
     _block *bl=&CAr[n];
     int start=bl->Start+1;
     int size=bl->Stop - start;
     Buf[bl->Start]=size;
     for(int i=0;i<size;i++){
       BYTE v=Buf[start+i];
       v^=0xFF;
       Buf[start+i]=v;
     }
   }
   ret=SetFilePointer(h,0,0,FILE_BEGIN);
   if(ret==INVALID_SET_FILE_POINTER) exit(-6);
   ret=WriteFile(h,Buf,sz,&br,NULL);
   if(ret==0) exit(-7);
   if(br!=sz) exit(-8);
   CloseHandle(h);
   return 0;
}
```

10. Попробуйте внести аналогичные изменения в проверку в программе 4p.exe. В этой программе применен несколько более сложный (хотя все же

простой) алгоритм шифрования. Вызов функции main находится по адресу 401403.

А) Результаты ваших действий отразите в отчете.

Б) Объясните в чем разница в алгоритмах дешифрования в программах 4.exe и 4p.exe и как это проявляется при попытках внесения изменений в код.

В) Ответьте на вопросы в тексте задания (отмечены синим).

Г) предложите способы усиления защиты методом динамического дешифрования.

Лабораторная работа № 5.
Поиск вызова процедур DLL с использованием таблицы импорта.

1. Ваша основная задача найти все вызовы функции DefWindowProcA, экспортируемой в библиотеке USER32.DLL. При этом вы должны решить задачу, используя сведения о структуре PE заголовка и таблицы импорта.

2. Откройте программу test.exe в редакторе hiew.exe.

3. Найдите начало PE заголовка.

4. В нем найдите RVA таблицы импорта.

5. В таблице определите, из каких DLL импортируются элементы в данной программе, перечислите их все по порядку. Какая по счету в таблице искомая DLL?

6. Найдите искомую функцию в таблице “Hint-Name Table”. Какая она по счету?

7. Используя полученные данные и таблицу “Address Table”, определите адрес настроенной точки входа в процедуру.

8. Теперь, зная адрес настроенной точки входа, найдите все ссылки на нее. Приведите полный список адресов, где используется данная импортируемая функция.

9. Напишите отчет, в котором подробно отразите все ваши действия и по каждому пункту (с 3-го по 8-й) представьте все полученные и промежуточные результаты с точным указанием как они связаны с данными из предыдущих пунктов.