ВОЕННО-КОСМИЧЕСКАЯ АКАДЕМИЯ имени А.Ф.МОЖАЙСКОГО

Кафедра № 27 Математического и программного обеспечения

УТВЕРЖДАЮ

Начальник 27 кафедры

полковник

С. Войцеховский

«___» _____ 2022 г.

Практическое занятие № 15
по учебной дисциплине
«Защита информации»
на тему:

«Инструментальные средства обеспечения сертификационных испытаний программного обеспечения на отсутствие не декларированных возможностей»

Рассмотрено и одобрено
на заседании кафедры № 27
« ___ » _____ 202_ г. протокол № ___

Санкт-Петербург
2022

I. ТЕМА И ЦЕЛЬ ПРАКТИЧЕСКОГО ЗАНЯТИЯ

Тема практического занятия: «Защита программных средств от несанкционированного копирования, исследования, модификации».

Учебная цель: овладение навыками составления и отладки модуля защиты ПО от копирования.

Время - 180 мин.

Место – аудитория (класс) по расписанию занятий.

Учебно-материальное и методическое обеспечение

1. Лабораторные установки – персональные ЭВМ с установленным на них программным обеспечением.
2. Методические разработки по программированию модулей защиты ПО от копирования, исследования и модификации.
3. Варианты типовых заданий на практическое занятие.

II. УЧЕБНЫЕ ВОПРОСЫ И РАСЧЕТ ВРЕМЕНИ

№ п\п	Учебные вопросы	Время, мин.
1.	Вступительная часть. Контрольный опрос.	10
2.	Учебные вопросы. ОСНОВНАЯ ЧАСТЬ: 1. Установка и дизассемблирование программных средств с помощью IDA Pro. 2. Оформление отчетов и их защита, выставление оценок.	 120 40
3.	Заключительная часть. Задание и методические указания курсантам на самостоятельную подготовку	5

УЧЕБНЫЕ МАТЕРИАЛЫ

1. Теоретические сведения

Дизассемблирование – преобразование программы на машинном языке к ее ассемблерному представлению. Декомпиляция – получение кода языка высокого уровня из программы на машинном языке или ассемблере.

Зачем нужно дизассемблирование. Цель инструментов дизассемблирования заключается в содействии исследованию функционирования программ, когда их исходные коды не доступны. Наиболее распространенные цели дизассемблирования:

− анализ вредоносного программного обеспечения;
− анализ уязвимостей программного обеспечения с закрытым исходным кодом;
− анализ совместимости программного обеспечения с закрытым исходным кодом;
− валидация компилятора;
− отображение команд программы в процессе отладки.

Базовый алгоритм дизассемблирования

Шаг 1. Первым шагом в процессе дизассемблирования является идентификация кодового сегмента. Так как команды обычно смешаны с данными, то дизассемблеру необходимо их разграничить.

Шаг 2. Получив адрес первой команды, необходимо прочитать значение, содержащееся по этому адресу (или смещению в файле) и выполнить табличное преобразование двоичного кода операции в соответствую ему мнемонику языка ассемблера.

Шаг 3. Как только команда была обнаружена и декодирована, ее ассемблерный эквивалент может быть добавлен к результирующему листингу. После этого необходимо выбрать одну из разновидностей синтаксиса языка ассемблера.

Шаг 4. Далее необходимо перейти к следующей команде и повторить предыдущие шаги до тех пор, пока каждая команда файла не будет дизассемблирована.

2. Практические особенности работы дизассемблера: IDA Pro:

Запуск IDA

При запуске IDA встретит вас загрузочным экраном, отображающим информацию о лицензии. После него отображается диалог, предлагающий три способа попасть в рабочее окружение:

New запускает помощника, который будет направлять вас при выборе файла для анализа. Вначале пользователь должен определить тип файла, который он хочет открыть. После того, как указан тип файла, для выбора файла используется стандартный диалог "Открыть файл". Наконец, отображается один или более дополнительных диалогов, которые позволяют вам указать специфические опции для анализа перед тем, как файл будет загружен, проанализирован и отображён.

Трудность в работе с помощником New File кроется в том, что пользователь должен точно знать тип файла, который необходимо открыть. Если тип файла неизвестен, лучше использовать другой метод загрузки в IDA. Кнопка New соответствует команде File ► New.

Go прерывает процесс загрузки и открывает IDA с пустой рабочей областью. Команда File ► New запускает помощник New File Wizard. Команда File ► Open запускает диалог "открыть файл" без запуска помощника.

По умолчанию IDA применяет фильтр **known extensions**. Убедитесь, что в фильтре установлено нужное вам значение или он вообще отключен (необходимо выбрать "All Files". Когда вы открываете файл этим способом, IDA пытается автоматически определить тип загружаемого файла.

Previous позволяет открыть ранее используемые файлы. Список этих файлов наполняется значениями из записи **History** ключа реестра IDA.

Загрузка файла в IDA. Если вы решите открыть файл, используя команду File ► Open. IDA генерирует список потенциальных типов файлов и отображает этот список вверху диалога. Список показывает, какие загрузчики IDA лучше всего подходят для работы с выбранным файлом. Этот список создается в результате запуска каждого файлового загрузчика в директории loaders. Если не знаете, какой загрузчик выбрать, неплохим решением будет остановиться на выбранном по умолчанию варианте.

Поля **Loading Segment** и **Loading Offset** активны, только если выбраны выходной формат Binary File и процессор семейства x86. Так как двоичный загрузчик не может извлечь из файла какую-либо информацию о размещении памяти, значения сегмента и смещения используются для получения начального адреса загруженного содержимого.

Кнопки **Kernel Options** предоставляют доступ к конфигурации специфических опций дизассемблирования, которые IDA будет использовать для улучшения процесса рекурсивного погружения.

Кнопки **Processor Options** предоставляют доступ к опциям конфигурации выбранного процессорного модуля.

Рабочая область IDA

Область панели инструментов содержит инструменты, соответствующие наиболее часто используемым операциями IDA.

Цветная горизонтальная панель - это ***навигатор обзора.***

Окно дизассемблирования служит основным средством вывода данных. У него есть два режима вывода: граф (по умолчанию) и листинг. Если вы хотите сделать режим листинга режимом по умолчанию, вы должны снять галочку с опции Use Graph View by Default на вкладке Graph в меню Options ► General.

Обзор графа, доступный только когда активен режим графа, предоставляет общий вид базовой структуры графа. Пунктирный прямоугольник показывает текущую область, видимую в окне. окне.

Окно сообщений предоставляет всю информацию, генерируемую IDA.

Обзор графа, доступный только когда активен режим графа, предоставляет общий вид базовой структуры графа. Пунктирный прямоугольник показывает текущую область, видимую в окне.

Окно сообщений предоставляет всю информацию, генерируемую IDA.

Окно дизассемблирования Для окна дизассемблирования доступно два формата: текстовый листинг и графы, добавленные в версии 5.0. Режим графа всегда включается по умолчанию. Вы можете изменить его поведения с помощью вкладки Graph диалога Options ► General. Когда окно дизассемблирования активно, вы можете переключаться между графом и листингом в любое время по нажатию пробела.

Просмотр графа

Просмотр графа немного напоминает блок-схему программы тем, что функция разбита на базовые блоки, что позволяет визуализировать переход функции от одного блока к другому.

IDA использует стрелки разных цветов, чтобы различать течения различных типов между блоками функции. Базовые блоки, которые прерываются переходом по условию, генерируют два возможных течения в зависимости от проверяемого условия: *край «Да»* (есть переход по этой ветке) по умолчанию зеленый, *край «Нет»* - красный. Блоки без ветвления, имеющие всего один вариант продолжения, используют *нормальный край* (по умолчанию синий) для указания на следующий блок.

В режиме графа IDA отображает одну функцию за раз. Зажав CTRL и прокручивая колесико мыши, можно приближать и отдалять граф. То же самое можно выполнить, зажав CTRL и нажимая клавиши + и – на дополнительной клавиатуре. В случаях больших или сложных функции может пригодиться окно Общего вида графа. Оно всегда отображает структуру графа целиком, а пунктирная рамка показывает область, которая отображена в окне дизассемблирования.

Существует несколько способов изменить вид дисплея графов для своих нужд:

Panning. IDA прячет менее необходимую информацию о каждой ассемблерной строчке (например, информацию о виртуальном адресе) чтобы блок занимал меньше места на экране. Вы можете включить отображение дополнительной информации с каждой строчкой кода, выбирая нужные пункты в *disassembly line parts, доступном во кладке Disassembly* через меню Options ►

General. Например, чтобы добавить виртуальные адреса к каждой строчке, мы включаем *line prefixes*.

Перемещение блоков. Отдельные блоки внутри графа можно перемещать на новые места, нажимая на панель с названием блока и перетаскивая в нужное место. Вы можете вручную перенаправить края, перетаскивая вершины в новые места. Новую вершину можно сделать, дважды кликнув по желаемому месту на крае, зажав клавишу SHIFT. Если вам понадобится вернуться к расположению графа по умолчанию, вы можете щелкнуть по графу правой кнопкой и выбрать Layout Graph.

Группировка и сворачивание блоков. Блоки можно группировать, по одному или вместе с другими блоками, и сворачивать, чтобы освобождать место на экране. Свернуть блок можно, нажав правой кнопкой по его панели с названием и выбрав Group Nodes.

Создание дополнительных окон дизассемблирования. Если вы захотите видеть графы двух разных функций одновременно, вам надо всего лишь открыть новое окно дизассемблирования, используя Views ► Open Subviews ► Disassembly. Первое окно дизассемблирования называется *IDA View-A*, последующие - *IDA View-B, IDA View-C* и так далее. Каждое окно дизассемблирование независимо от других.

Режим текста

Текстовое окно дизассемблирования – традиционный режим просмотра сгенерированного IDA кода. Текстовый дисплей показывает полный листинг программы (а не единственную функцию, как в режиме графов) и только он предоставляет средства для просмотра области данных двоичного файла. Вся информация, доступная в режиме графа, в том или ином виде доступна и в текстовом режиме.

Ассемблерный код представленный линейно, виртуальные адреса по умолчанию включены и обычно отображаются в формате [название раздела]:[виртуальный адрес], например, **text:0040110C0**.

Окно стрелок используется для отображения нелинейных переходов внутри функции. Обычные стрелки соответствуют безусловным переходам, а пунктирные – переходам по условию. Когда переход передает управление более раннему адресу программы, используется жирная линия. Обратное течение в программе указывает на наличие цикла.

Объявления отражают оценку стекового кадра, сделанную IDA. IDA определяет структуру стекового кадра функции, выполняя детальный анализ поведения указателя стека и всех указателей стековых кадров, использованных в функции.

Комментарии (точка с запятой начинает комментарий) - это *перекрёстные ссылки*.

Для примеров в книге мы чаще всего будем использовать текстовый режим.

Окно имён

Окно имён предоставляет общий список всех глобальных имен в двоичном файле. *Имя* – не более чем символическое описание, данное виртуальному адресу программы. IDA получает список имен с помощью символьной таблицы и анализа подписи во время начальной загрузки файла. Имена можно отсортировать по алфавиту или в порядке виртуальных адресов (по возрастанию и по убыванию). Двойной щелчок по любой записи окна имён отобразит в окне дизассемблирования выбранное имя.

Отображаемые имена кодируются цветом и буквами. Система кодирования описана ниже:

• **F** Обычная функция. Эти функции IDA не распознает как библиотечные.

• **L** Библиотечная функция. IDA распознает библиотечные функции с помощью алгоритма сопоставления подписей.

• **I** Импортированное имя, чаще всего имя функции из общей библиотеки.

• **C** Именованный код. Это именованные места программы, которые IDA не относит ни к каким функциям.

• **D** Данные. Именованные секции даты обычно представляют собой глобальные переменные.

• **A** Строковые данные ASCII. Это секция данных, содержащая четыре или более последовательных символа ASCII, завершаемых нулевым байтом.

В процессе дизассемблирования IDA генерирует имена для всех мест, на которые есть прямые ссылки как на код (цель ветвления или вызова) или как на данные (чтение, запись или взятие адреса). Если место именовано в символьной таблице имен, IDA заимствует имя из таблицы. Если для данного места записи в таблице нет, IDA генерирует имя по умолчанию, которое будет использоваться при дизассемблировании. Когда IDA решает назвать место, виртуальный адрес места совмещается с префиксом, указывающим на тип места. Ниже перечислены наиболее часто встречаемые префиксы, используемые при автоматической генерации:

− sub_***xxxxxx*** **-** Подпрограмма по адресу *xxxxxx*
− loc_***xxxxxx*** **-** Инструкция, расположенная по адресу *xxxxxx*
− byte_***xxxxxx*** **-** 8-битовые данные по адресу *xxxxxx*
− word_***xxxxxx*** **-** 16-битовые данные по адресу *xxxxxx*
− dword_***xxxxxx*** **-** 32-битовые данные по адресу *xxxxxx*
− unk_***xxxxxx***
Данные неизвестного размера по адресу *xxxxxx.*

3. **Пример**

Научиться получать исходные тексты простых программ с помощью дизассемблирования.

Скомпилировать следующую простую программу:

```
#include «stdafx.h»
#include <iostream.h>
int main (int argc, char* argv[ ])
{
cout << «Hello, Ivan!»;
return 0;
}
```

С помощью любого дизассемблера получить из файла типа *.exe исходный текст программы. При использовании дизассемблера: IDA Pro для этого необходимо:

Если все настройки оставить по умолчанию, то после завершения анализа экран должен выглядеть следующим образом:

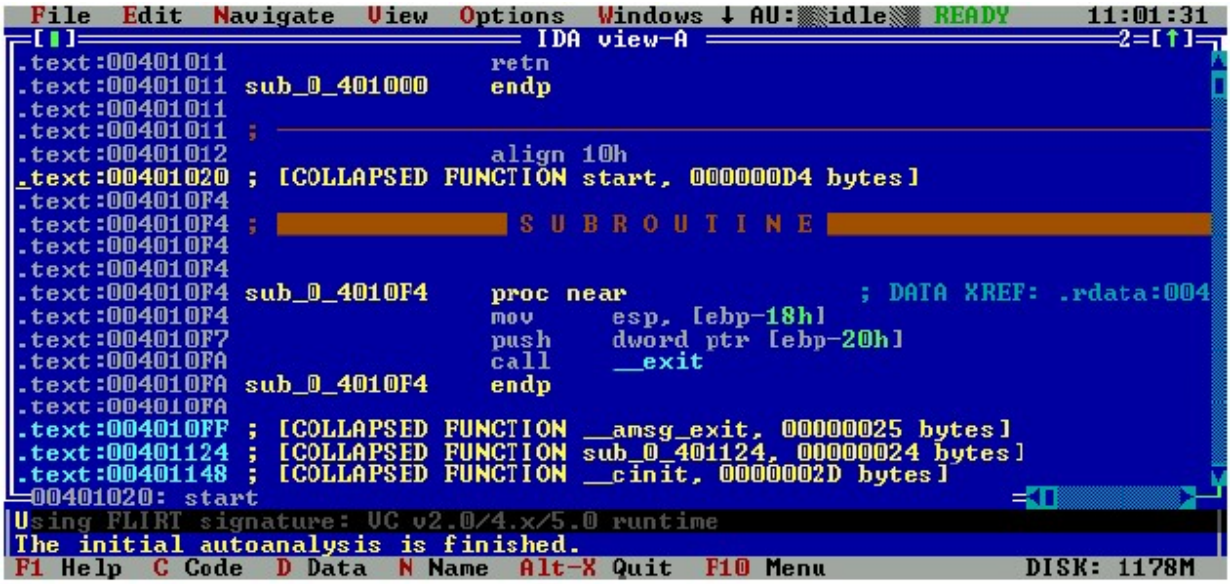

Рис.1. Интерфейс программы IDA Pro

«Сворачивание» функций очень упрощает навигацию по файлу, позволяя втиснуть больше информации в тесное пространство экрана. «Развернуть» функцию можно, подведя к ней курсор и нажав «+» на дополнительной цифровой клавиатуре. Соответственно, что бы свернуть, необходимо нажать «-». По умолчанию все библиотечные функции представляются свернутыми.

В нашем случае мы имеем дело с библиотечной функцией «START», а, точнее говоря, со сгенерированным компилятором стартовым кодом. Он выполняет инициализацию всех библиотек, подготавливает систему ввода вывода и делает массу других дел, совершенно не интересующих нас нас на данный момент. Но в каком-то месте он передает управление функции main(), содержимое которой мы и пытаемся проанализировать.

Воспользуйся бы мы любым другим дизассемблером — нам пришлось бы первым делом тщательно изучить стартовый код компилятора в поисках места передачи управления на интересующую нас функцию (или заглянуть в исходные коды библиотек компилятора). Но первое трудоемко, а второе предполагает наличие у нас той же версии компилятора, что далеко не всегда выполнимо.

Все будет гораздо проще, если мы воспользуемся возможностью IDA находить перекрестные ссылки. Поскольку стартовый код вызывает только одну функцию (не считая библиотечных), то последняя и окажется искомой!

.text:00401000 sub_0_401000 proc near ; CODE XREF: start+AF↓p

Прокрутим экран чуть дальше и рассмотрим следующую строку. Комментарий, указывающий на перекрестную ссылку, говорит, что эту процедуру вызывает стартовый код, и, если мы хотим взглянуть на него поближе, то нужно подвести курсор в границы выражения «start+AF↓p» и нажать «Enter». При этом IDA автоматически перейдет к требуемому адресу. Это, действительно, очень удобное средство навигации.

IDA распознает не только константы и символьные имена, но и сложные выражения и конструкции, причем независимо от того, как последние были созданы.

Попробуем нажать «Insert» и ввести следующую строку, которая будет отображена как комментарий «А сейчас мы перейдем по адресу 0x40103D». Если теперь подвести курсор к «0x40103D» и нажать «Enter», то IDA действительно перейдет по требуемому адресу! И возвращается назад клавишей «Esc». Это дает возможность организовывать в комментариях свои гиперссылки, позволяющие легко ориентироваться в исследуемом файле и быстро переключаться между разными фрагментами.

Вернемся назад и попробуем заглянуть в функцию start(). Увы, на этот раз IDA себя поведет не так, как ожидалось, и просто переместит курсор на свернутую функцию. Попробуем развернуть ее (клавишей «+» на дополнительной клавиатуре) и повторить операцию. На этот раз все проходит успешно. Интуитивно понятно, что должна быть функция авторазвертки при подобных переходах, но, по крайней мере, в версии 3.84 таковая отсутствует.

```
.text:004010A9 call __setargv
.text:004010AE call __setenvp
.text:004010B3 call __cinit
.text:004010B8 mov eax, dword_0_408784
.text:004010BD mov dword_0_408788, eax
.text:004010C2 push eax
.text:004010C3 push dword_0_40877C
.text:004010C9 push dword_0_408778
.text:004010CF call sub_0_401000
.text:004010D4 add esp, 0Ch
.text:004010D7 mov [ebp+var_1C], eax
.text:004010DA push eax
.text:004010DB call _exit
```

Как видно, sub_0_401000 — единственная функция (за исключением библиотечных), вызываемая стартовым кодом. Следовательно, именно она и есть main(). Подведем курсор к sub_0_401000 и нажмем Enter. Было бы неплохо дать ей осмысленное символьное имя, и IDA это позволяет. Для этого нужно подвести курсор к началу функции и нажать N (или в меню View/Name выбрать ее из списка всех функций, изменить которые можно нажатием «Ctrl+E», введя в открывшемся окне диалога любое осмысленное имя). В результате получится следующее:

```
.text:00401000 main proc near ; CODE XREF: start+AF↓p
.text:00401000 push offset aHelloSailor; «Hello, Ivan!»
.text:00401005 mov ecx, offset dword_0_408900
.text:0040100A call ??6ostream@@QAEAAV0@PBD@Z
; ostream::operator<<(char)
.text:0040100F xor eax, eax
.text:00401011 retn
.text:00401011 main endp
```

Обратим внимание на строку 401000h, а точнее на метку «aHello Ivan» — IDA распознала в ней строку символов и сгенерировала на основе их осмысленное имя, а в комментариях продублировала для наглядности оригинал. При этом, как уже отмечалось, IDA понимает символьные метки, и, если подвести к последней курсор и нажать на «Enter», то можно увидеть следующие:

```
.data:00408040 aHeloSailor db 'Hello, Ivan!',0 ; DATA XREF: main↑o
```

«o» — это сокращение от «offset», т.е. IDA позволяет уточнить тип ссылки. Ранее мы уже сталкивались с использованием в этом качестве символа «p», т.е. указателем (pointer). Иногда еще используется символ «u» (от слова undefine) — неопределенный, нераспознанный. О нем мы поговорим позднее. Стрелка (вверх или вниз, соответственно) указывает, где расположена указанная ссылка.

```
.text:0040100A call ??6ostream@@QAEAAV0@PBD@Z
                                        ; ostream::operator<<(char)
```

IDA сумела распознать в этой функции библиотечный оператор «ostream::operator<<», освободив нас от большой части работы. Но как она это проделала? Точно так, как антивирус распознает вирусы — по сигнатурам. Понятно, что бы этот механизм работал необходимо сначала создать базу сигнатур для библиотек распространенных компиляторов и оперативно ее обновлять и расширять. IDA, конечно, невсемогуща, но список поддерживаемых компиляторов, очень впечатляющий

(он расположен в каталоге SIG/LIST), при этом реально поддерживаются многие версии, даже не указанные в перечне, поскольку они часто имеют схожие сигнатуры.

Все функции имеют два имени: одно -которое дает им библиотека, и второе - общепринятое из заголовочных файлов. Если заглянуть в библиотеку используемого компилятора (в нашем случае это MS VC 6.0), то можно увидеть, что оператор «cout <<» есть ни что иное, как одна из форм вызова функции «??6ostream@@QAEAAV0@PBD@Z», трудночитаемое имя которой на самом деле удобно для компоновщика и несет определенную смысловую нагрузку.

Две следующие строки завершают выполнение main() с нулевым кодом возврата (эквивалентно return 0).

.text:0040100F xor eax, eax

.text:00401011 retn

По завершении реализации программы оформляется письменный отчет, в который помещается текст задания, текст программы и модуля, структурная схема программы и выводы по работе. Пример отчета приведен в приложении к Методической разработке.

4. Общие методические указания курсантам (слушателям) по подготовке к практическим занятиям

Практические занятия по дисциплине «Защита информации» проводятся в классе ПЭВМ. Индивидуальные задания выполняются каждым курсантом лично.

Перед выполнением задания обучающийся изучает материал, приведенный в разделе «Учебные материалы», в ходе которого необходимо разобрать приведенные примеры и выполнить задания раздела. На следующем этапе работы обучающийся выполняет индивидуальное задание.

Результаты работы оформляются в виде отчета. Содержание отчета приведено в руководстве по соответствующему практическому занятию.

По готовности к защите работы курсант (слушатель) докладывает преподавателю.

5. Индивидуальные задания к практическому занятию №1

Вариант 1.

Скомпилировать следующую простую программу:

```
#include <iostream>
#include <conio.h>
using namespace std;
void main()
{
	setlocale(LC_ALL,"Russian");
	int *a = new int[10];
	int *b = new int[10];
	cout<<"Введите 10 элементов массива:\n";
	for(int i=0;i<10;i++)
	{
		cout<<i<<"-й элемент:  ";
		cin>>a[i];
	}
	int sum =0;
	int j = 0;
```

```
	for(int i=0;i<10;i++)
	{
		if(a[i] > 0)
		{
			sum = sum + a[i];
		}
		if(i == 9) continue;
		else
		{
			if(a[i+1] > a[i])
			{
				b[j] = i+1;
				j++;
			}
		}
	}
	cout<<"Индексы элементов массива больших предыдущих равны: ";
	for(int i=0;i<j;i++)
	{
		cout<<b[i]<<"  ";
	}
	cout<<"\nУдвоенная сумма положительных элементов равна: "<<sum*2;
	getch();
	delete a;
	return;
}
```

С помощью любого дизассемблера получить из полученного файла типа *.exe исходный текст программы.

Вариант 2.

Скомпилировать следующую простую программу:

```
#include <iostream>
#include <conio.h>
using namespace std;

int sum(int a,int b)
{
      return a+b;
}

void main()
{
      int a,b;
      cout<<"Enter A: ";
      cin>>a;
      cout<<"\nEnter B: ";
      cin>>b;
      cout<<"\n"<<a<<" + "<<b<<" = "<<sum(a,b);
      getch();
     return;
}
```

С помощью любого дизассемблера получить из полученного файла типа *.exe исходный текст программы.

Вариант 3.

Скомпилировать следующую простую программу:

```
#include <iostream>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
using namespace std;

int main()
{
      setlocale(LC_ALL,"Russian");
      FILE *fi, *fo;
      fi = fopen("c:\\file.txt", "r");
      if(fi == 0)
      {
            cout<<"Ошибка открытия входного файла";
            getch();
            return 1;
      }
      fo = fopen("c:\\binfile.out", "w+b");
      if(fo == 0)
      {
            cout<<"Ошибка открытия выходного файла";
            getch();
            return 1;
      }
```

```
      const int dl = 80;
      char s[dl];
      struct
      {
            char type[20];
            int opt, rozn;
            char comm[40];
      }mon;
      int kol = 0 ; // Количество записей в файле
      while (fgets(s, dl, fi))
      {
      // Преобразование строки в структуру
            strncpy(mon.type, s,19);
            mon.type[19]='\0';
            mon.opt = atoi(&s[20]);
            mon.rozn = atoi(&s[25]);
            strncpy(mon.comm, &s[30], 40);
            fwrite(&mon, sizeof mon, 1, fo);
            kol++;
      }
      fclose(fi);
      int i;
      cout<<"Введите номер записи: ";
      cin>>i; // Номер записи
      if (i >= kol)
      {
            cout<<"Запись не существует";
            getch();
            return 1;
      }
      // Установка указателя текущей позиции файла на i-ю запись
      fseek(fo, (sizeof mon)*i, SEEK_SET);
      fread(&mon, sizeof mon, 1, fo);
      cout<<"Тип монитора: "<<mon.type<<" Опт. цена: "<<mon.opt
            <<" Розн. цена: "<<mon.rozn<<endl;
      fclose(fo);
      getch();
      return 0;
}
```

С помощью любого дизассемблера получить из полученного файла типа *.exe исходный текст программы.

Вариант 4.

Скомпилировать следующую простую программу:

```
#include <iostream>
#include <conio.h>

using namespace std;

void main()
{
      setlocale(LC_ALL,"Russian");
      float K,L,C,X,Y;
      cout<<"Введите вещественное число K: ";
      cin>>K;
```

```
      cout<<"Введите вещественное число L: ";
      cin>>L;
      cout<<"Введите вещественное число C: ";
      cin>>C;
      cout<<"Введите вещественное число X: ";
      cin>>X;

      Y = ((K+L+C)*(K+L+C))/(2*X) + 25;

      cout<<"Y = "<<Y;

      getch();
      return;
}
```

С помощью любого дизассемблера получить из полученного файла типа *.exe исходный текст программы.

Вариант 5.

Скомпилировать следующую простую программу:

```
#include <iostream>
#include <conio.h>
using namespace std;

#define SQR(x) x*x  //макрос возведения числа в квадрат

int main()
{
      setlocale(LC_ALL,"Russian");
      float i, s;
      cout<<"Введите число, возводимое в квадрат: ";
      cin>>i;
      s = SQR(i);
      cout<<"\nКвадрат числа "<<i<<" равен "<<s<<endl;
      getch();
      return 0;
}
```

С помощью любого дизассемблера получить из полученного файла типа *.exe исходный текст программы.

Вариант 6.

Скомпилировать следующую простую программу:

```
using namespace std;

#ifndef GROUP_SIZE
      #define GROUP_SIZE 10
#endif

void main()
{
 setlocale(LC_ALL,"Russian");
 cout<<"В группе "<<GROUP_SIZE<<" курсантов.\n";
 cout<<"Введите оценки по аттестации каждому курсанту:\n";

 int *marks = new int[];
```

```
 float av_marks = 0;
 for(int i=0;i<GROUP_SIZE;i++)
 {
  cin>>marks[i];
  av_marks = av_marks + marks[i];
 }
 cout<<"\nСредний балл равен: "<<av_marks/GROUP_SIZE;
 getch();
 return;
}
```

С помощью любого дизассемблера получить из полученного файла типа *.exe исходный текст программы.

Вариант 7.

Скомпилировать следующую простую программу:

```
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;
int main()
{
      setlocale(LC_ALL,"Russian");
      char line[71],text[1001];
     string str;
      FILE *f;
      if((f = fopen("C:/projects/strings/text.dat","rt")) == 0)
      {
            cout<<"Ошибка открытия входного файла";
            getch();
            return -1;
      }
      //построчное копирование строки из файла длиной 70 символов
      //и добавление к строке str
      while(!feof(f))
      {
            fgets(line,71,f);
            str.append(line);
      }
      int size = str.size();
      int count = 0;
      char *p = strcpy(text,str.c_str()); //копирование строки в массив text
      //перебор массива text и сравнение символов с '.'
      while(p = strchr(p,'.'))
      {
            count++;
            p++;
      }
     cout<<str<<"\n";
      cout<<"Число предложений в тексте: "<<count;
      getch();
      return 0;
}
```

С помощью любого дизассемблера получить из полученного файла типа *.exe исходный текст программы.

Задание повышенной сложности:

Скомпилировать следующую простую программу:

```
#include <iostream>
#include <conio.h>
using namespace std;

//===== Функция расчета среднего балла ========
float average_ball(int marks[],int n = 5)
{
float s = 0;
for(int i=0;i<n;i++)
          {
               float m = marks[i];
               s = s + m;
          }
          return s/n;
     }
     //=============================================
     void main()
     {
          setlocale(LC_ALL,"Russian");
          struct KURSANT
          {
               char fio[30];
               int group_number;
               int marks[5];
          };
          int count;
          cout<<"Введите число курсантов в группе: ";
          cin>>count;

          KURSANT *group = new KURSANT[count];
     //====== Ввод =========
          for(int i=0;i<count;i++)
          {
               cout<<"\nВведите данные по "<<i+1<<" курсанту:";
               cout<<"\nФамилия: ";
               cin>>group[i].fio;
               cout<<"Номер группы: ";
               cin>>group[i].group_number;
               cout<<"Успеваемость (5 оценок): ";
               for(int j=0;j<5;j++)
               {
                    cin>>group[i].marks[j];
               }
          }
          cout<<"\nКурсанты со средним баллом выше 4.0:\n";
     //====== Вывод =========
          int k = 1;
          for(int i=0;i<count;i++)
          {
               int *m = group[i].marks;
               float av = average_ball(m,5);
               if(av > 4.0)
               {
                    cout<<"\n"<<k<<". "<<group[i].fio;
```

```
55.                         cout<<"; Уч. группа "<<group[i].group_number;
56.                         cout<<"; Средний балл: "<<av;
57.                         k++;
58.                   }
59.             }
60.             if(k == 1)
61.             {
62.                   cout<<"\nВ группе нет курсантов со средним баллом выше
   4.0";
63.             }
64.             getch();
65.             return;
66.       }
```

С помощью любого дизассемблера получить из полученного файла типа *.exe исходный текст программы.

6. Отчетность по работе

По выполнению работы каждый курсант должен представить отчет. Отчет должен содержать:
– название практического занятия;
– текст индивидуального задания;
– блок-схему алгоритма решения задачи;
– исходный текст программы;
– результаты тестирования решения.
В процессе выполнения индивидуального задания или после завершения его выполнения преподаватель проводит собеседование с каждым курсантом по теме выполненной работы, проверяя также практические навыки, приобретенные в ходе занятия. Отчетный материал предоставляется преподавателю, а результаты защищаются.

7. Заключительная часть

В заключительной части подводятся итоги проделанной работы, дается краткая оценка действиям участников, прослеживается связь с теоретическими положениями и перспективой на будущую деятельность

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПРЕПОДАВАТЕЛЮ ПРИ ПРОВЕДЕНИИ ПРАКТИЧЕСКОГО ЗАНЯТИЯ

Во вступительной части занятия производится контроль присутствия и готовности обучающихся к занятию. Объявляется тема, цель, учебные вопросы занятия и особенности его проведения.
Готовность группы к занятию проверяется контрольным опросом.
Вопрос 1: Что подразумевают под понятием дизассемблирование?
Вопрос 2: Какие средства дизассемблирования вы знаете?
Вопрос 3: Что такое динамический анализ исходных текстов программ?
При отработке первого вопроса занятия основное внимание обратить на усвоение обучающимися принципов дизассемблирования программных средств с помощью IDA Pro.
При отработке третьего вопроса необходимо акцентировать внимание на структуре отчета о проделанной работе и защите его основных положений.
В заключительной части занятия подвести итоги, оценить действия обучающихся, ответить на вопросы.
Дать задание на самоподготовку. Объявить тему следующего занятия.

8. Задание и методические указания курсантам на самостоятельную подготовку:

1. Повторить по конспекту лекций и рекомендованной литературе основные возможности дизассемблеров.
2. Быть готовыми к самостоятельному дизассемблирования программ.

V. ИСПОЛЬЗОВАННАЯ ЛИТЕРАТУРА

1. **Оголюк А. А.** Защита приложений от модификации: учебное пособие. – СПб: СПбГУ ИТМО, 2013. – 56 с.

2. Вихорев С.В. Классификация угроз информационной безопасности. - http://www2.cnews.ru/comments/security/elvis_class.shtml.

Доцент 27 кафедры
к.т.н.
подполковник

С. Краснов

«___»_____20__г.