Лабораторная работа №1
«Ресурсы ЭВМ. Ресурсы ОС»

«1» октября 2021 г.

Москва 2021 г.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

Информация – сведения (сообщения, данные) независимо от формы их представления [1].

Информационная система – совокупность содержащейся в базах данных информации и обеспечивающих ее обработку информационных технологий и технических средств [1].

Конфиденциальность – свойство конкретной информации быть доступной только тому кругу лиц, для которого она предназначена.

Целостность – актуальность и непротиворечивость информации, ее защищенность от разрушения и несанкционированного изменения.

Доступность – возможность за приемлемое время получить требуемую информацию легальному пользователю.

Авторство (**Аутентичность**) – гарантия того, что источником информации является именно то лицо, которое заявлено как ее автор.

Неотказуемость от авторства – невозможность автора отказаться от авторства.

Информационная безопасность – это свойство информации сохранять конфиденциальность, целостность, доступность, авторство и неотказуемость от авторства. Угрозой нарушения безопасности считается угроза нарушения одного из свойств безопасности информации.

ОС – операционная система.

ПО – программное обеспечение.

ВМ – виртуальная машина.

ЛР – лабораторная работа.

1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1 Основные положения

Для выполнения лабораторной работы потребуются знания по разделам:

- архитектура компьютера;
- языки программирования – ассемблер и язык высокого уровня (например, язык С);
- принципы работы ОС с объектами.

1.2 Архитектура компьютера

Архитектура компьютера – это система команд и мест размещения операндов (регистров и памяти). Обычно выделяют два типа архитектур:

1. Архитектура фон Неймана.
2. Гарвардская архитектура.

1.2.1 Архитектура фон Неймана

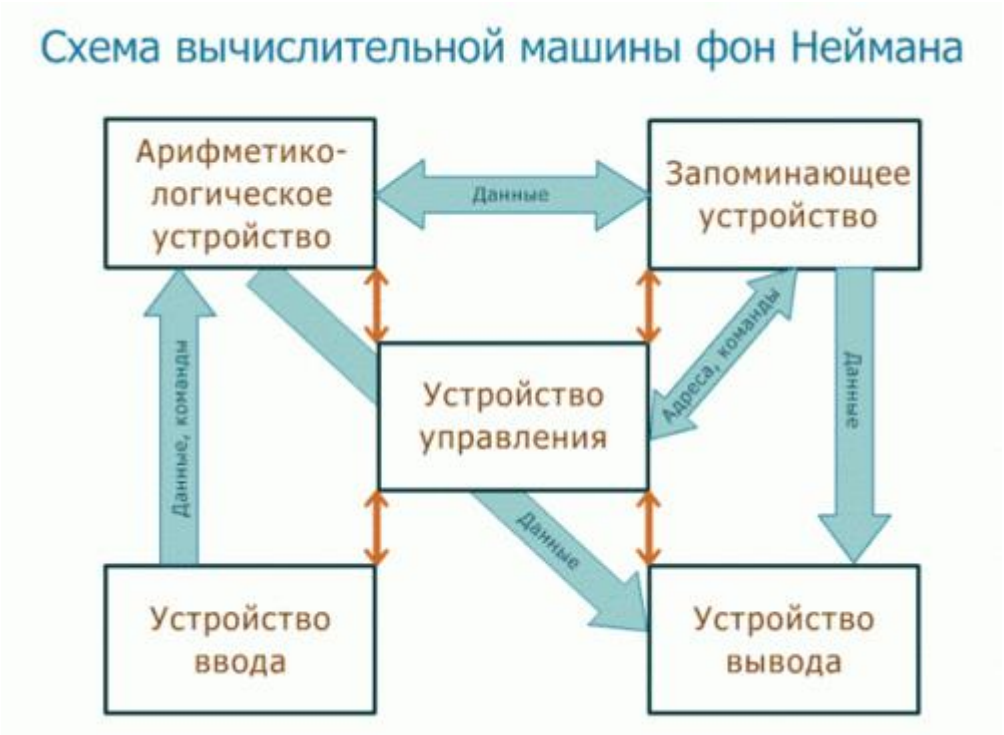

Рисунок 1 – Архитектура ЭВМ фон Неймана

Архитектура компьютера фон Неймана реализует известный принцип совместного хранения программ и данных в памяти компьютера [2]. В общем случае, когда говорят об архитектуре фон Неймана, подразумевают физическое отделение процессорного модуля от устройств хранения программ и данных [2].

В соответствии с принципами фон Неймана компьютер состоит из арифметико-логического устройства (АЛУ), выполняющего арифметические и логические операции; устройства управления (УУ), предназначенного для организации выполнения программ; запоминающих устройств (ЗУ), в т.ч. оперативного запоминающего устройства (ОЗУ); внешних устройств для ввода-вывода данных [2].

Программы и данные вводятся в память из устройства ввода через арифметико-логическое устройство. Все команды программы записываются в соседние ячейки памяти, а данные для обработки могут содержаться в произвольных ячейках [2].

Команда состоит из указания, какую операцию следует выполнить (из возможных операций на данном «железе») и адресов ячеек памяти, где хранятся данные, над которыми следует выполнить указанную операцию, а также адреса ячейки, куда следует записать результат (если его требуется сохранить в ЗУ) [2].

Из арифметико-логического устройства результаты выводятся в память или устройство вывода [2]. Принципиальное различие между ОЗУ и устройством вывода заключается в том, что в ОЗУ данные хранятся в виде, удобном для обработки компьютером, а на устройства вывода (принтер, монитор, жесткий диск и др.) поступают так, как удобно человеку [2].

Управляющее устройство управляет всеми частями компьютера [2]. УУ содержит специальный регистр (ячейку), который называется «счетчик команд». После загрузки программы и данных в память в счетчик команд записывается адрес первой команды программы [2]. УУ считывает из памяти содержимое ячейки памяти, адрес которой находится в счетчике команд, и помещает его в специальное устройство – «Регистр команд» [2]. УУ определяет операцию команды, «отмечает» в памяти данные, адреса которых указаны в команде, и контролирует выполнение команды. Операцию выполняет АЛУ или аппаратные средства компьютера [2].

В результате выполнения любой команды счетчик команд изменяется на единицу и, следовательно, указывает на следующую команду программы [2]. Когда требуется выполнить команду, не следующую по порядку за текущей, а отстоящую от данной на какое-то количество адресов, то специальная команда перехода содержит адрес ячейки, куда требуется передать управление [2].

Нейману удалось обобщить научные разработки и открытия многих других ученых и сформулировать на их основе принципы этого подхода [2]:

1. Использование двоичной системы счисления в вычислительных машинах. Преимущество перед десятичной системой счисления заключается в том, что устройства можно делать достаточно простыми, арифметические и логические операции в двоичной системе счисления также выполняются достаточно просто [2].
2. Программное управление ЭВМ. Работа ЭВМ контролируется программой, состоящей из набора команд. Команды выполняются последовательно друг за другом. Созданием машины с хранимой в памяти программой было положено начало тому, что мы сегодня называем программированием [2].
3. *Возможность условного перехода в процессе выполнения программы*. Несмотря на то, что команды выполняются последовательно, в программах можно реализовать возможность перехода к любому участку кода [2].
4. *Ячейки памяти ЭВМ имеют адреса, которые последовательно пронумерованы*. В любой момент можно обратиться к любой ячейке памяти по ее адресу. Этот принцип открыл возможность использовать переменные в программировании [2].
5. *Память компьютера используется не только для хранения данных, но и программ*. При этом и команды программы и данные кодируются в двоичной системе счисления, т.е. их способ записи одинаков. Поэтому в определенных ситуациях над командами можно выполнять те же действия, что и над данными [2].

Последние три утверждения являются очень важными для понимания задач информационной безопасности. Одной из основных проблем ИБ является существо-

вание эксплойтов уязвимостей и вредоносных программ, которые могут храниться в зашифрованном виде (то есть представляться в виде данных, а не кода!).

Для более лучшего понимания раздела «Архитектура компьютера» рекомендуется изучить соответствующую литературу [3].

Характерной реализацией архитектуры фон Неймана является архитектура x86/x64. Для ее работы необходимы следующие сущности (Рисунок 5):

1. Само арифметико-логическое устройство.
2. Регистры процессора, где в том числе хранятся счетчики и адреса.
3. Оперативная память со стеком.

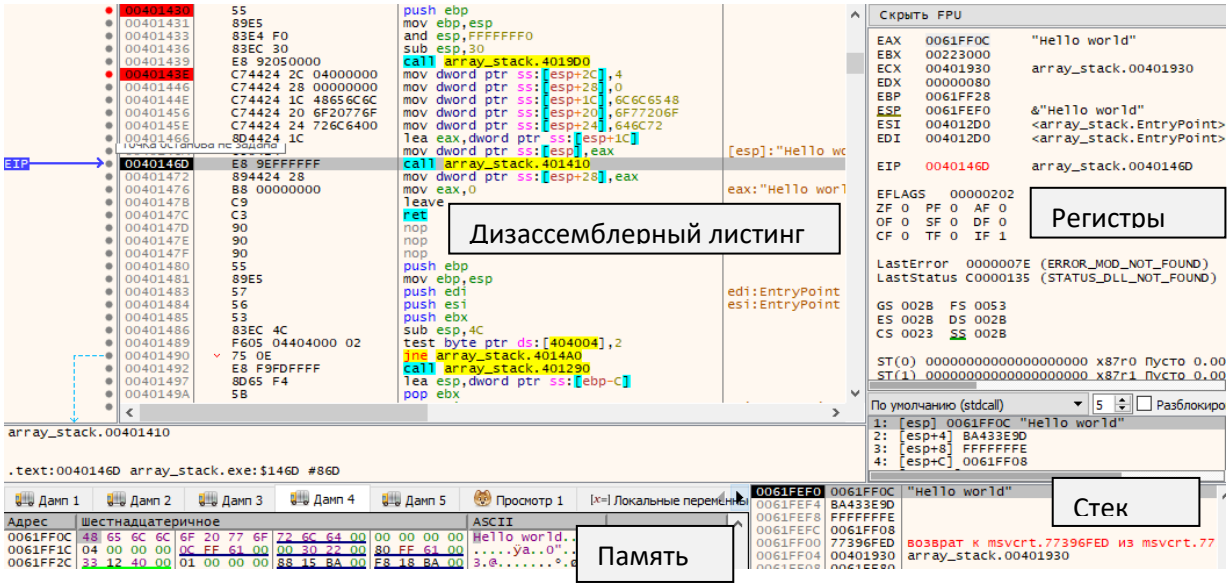

Рисунок 2 – Окно отладчика x64dbg

На рисунке 2 приведен скриншот отладчика x64dbg (x32dbg) [4], используемого для отладки программ на уровне ассемблера x86 архитектуры. В главном окне приводится листинг команд процессора. В левом нижнем углу скриншота (Рисунок 5) приведена память, доступная во время выполнения программы. В правом нижнем углу приведен стек (если выбрать нужный адрес в левом нижнем углу, то стек можно просмотреть и там). В стеке хранятся значения локальных переменных, а также *адреса возврата*, вызовов функций и другие вспомогательные данные.

В правом верхнем углу приведены регистры (регистры общего назначения, регистр флагов, регистр указателя текущей инструкции и т.д.)

Среди важных примеров инструкций можно выделить следующие:

1. MOV EBP, ESP – записывает в регистр EBP (регистра указателя базы) значение регистра ESP (регистра указателя на вершину стека)
2. MOV BYTE PTR [EAX], 10h – записывает в ячейку памяти с адресом, хранящимся в регистре EAX значение 10h (в десятичной системе – 16)
3. PUSH EBP – записывает на вершину стека значение регистра EBP
4. POP EBP – извлекает с вершины стека значение и записывает в регистр EBP
5. JMP EDI – безусловный переход на адрес, содержащийся в регистре EDI
6. CMP EAX,80h –> JZ 4300000 – связка двух команд сравнения значения регистра EAX с числом 80h (в десятичной системе – 128) и переход на адрес 4300000

В рамках более качественного ознакомления с темой архитектуры компьютера и ассемблера для аналитика вредоносного ПО студенту предлагается (это не обязательно) изучить главу 4 [5] и сравнить поведение программ в архиве «test_programs.7z» (представлены простые программы на языке C):

1. Скомпилировать программы любым компилятором, применяемым для работы с C/C++.
2. Скачать отладчик x64dbg, выбрать x32dbg и загрузить туда поочередно скомпилированные программы.
3. Если на первом шаге был выбран компилятор MinGW (gcc), то на адресе 401430 (сочетание клавиш Ctrl+G позволяет перейти на нужный адрес) установить программную точку останова (F2, Рисунок 5).
4. Затем продолжить выполнение программы с помощью клавиши F9.
5. Если все сделано правильно, то отладчик остановится на программной точке останова 401430. Затем можно пройти по программе, нажимая клавишу F8 и обращая внимание на значения переменных в стеке и в окне команд.

1.2.2 Гарвардская архитектура

Архитектура фон Неймана получила довольно широкое распространение. Однако в настоящее время активно ведется работа над конвейерными системами, в которых код и данные физически разделены. В конвейерных системах обращения и к командам, и к данным должны осуществляться одновременно [3]. Архитектура таких систем называется гарвардской (Harvard architecture), поскольку идея использования отдельной памяти для команд и отдельной памяти для данных впервые воплотилась в компьютере Marc III, который был создан Говардом Айкеном (Howard Aiken) в Гарварде [3]. По пути разработки систем с данной архитектурой пошли разработчики микроконтроллеров Microchip PIC и Atmel AVR.

1.3 Языки высокого уровня

Для разработки программ используются языки высокого уровня – C/C++, Java, C#, Python, JavaScript, Go, Scala, Kotlin и т.д. Языки высокого уровня можно поделить на два больших класса – императивные и декларативные. Программы, написанные на императивных языкых программирования, описывают алгоритмы и сам процесс работы с данными. Программист на декларативных языках программирования не описывает алгоритм работы программы, его интересуют входные данные и конечный результат.

Остановимся на императивных языках программирования. Их можно поделить на три вида в зависимости от того, что собой будет представлять конечный объект программы:

1. Компилируемые языки.
2. Языки, компилируемые в промежуточный код.
3. Интерпретируемые языки.

После компиляции файлов на языке C/C++ создается исполняемый файл, который будет выполняться в виртуальном адресном пространстве своего процесса (если это не DLL-библиотека) и работать с системой команд ЭВМ, а не промежуточной виртуальной машины. Представители второго вида языков (например, Java) транслируются в промежуточный код, который исполняется виртуальной машиной

Java – Java Virtual Machine. Третьим видом языков высокого уровня являются интерпретируемые, то есть те, которые работают в контексте интерпретатора (скрипты Python работают в контексте процесса python.exe, скрипты VBS – wscript.exe/cscript.exe, а PowerShell – powershell.exe).

Однако для понимания работы операционных систем (в частности, менеджера памяти) в лабораторных работах будет использоваться язык С. Он относится к процедурным языкам программирования. Его используют при разработке системного программного обеспечения по причине высокой скорости, гибкости и возможности гарантированно выполнить любую операцию за определенное время.

1.3.1 Основные конструкции языка С

Язык С относится к императивным языкам с типами данных, переменными и операторами управления. Поддерживаются разные типы переменных:

- целочисленные (байт, машинное слово/двойное машинное слово);
- вещественные;
- массивы и указатели на области памяти (то есть переменная хранит адрес) с данными и указатели на функции;
- структуры;
- перечисления;
- объединения.

Для понимания раздела по программированию на С предлагается изучить книгу «Язык программирования Си» [6] или один из последних принятых стандартов языка С.

1.3.2 Безопасность языка С и программ на языках высокого уровня

В исходном тексте лабораторной работы допущены некоторые логические ошибки. В нем отсутствуют грубые ошибки, характеризующиеся использованием так называемых «небезопасных функций». Существует широкораспространенная проблема несоответствия введенных пользователем данных и данных, которые предусмотрел программист.

Потенциально небезопасные функции C/C++ и пути замены:

- gets -> fgets/gets_s (MSDN)
- strcpy -> strncpy -> strlcpy/strcpy_s
- strcat -> strncat -> strlcat/strcat_s
- strtok
- sprintf -> snprintf
- vsprintf -> vsnprintf
- makepath -> _makepath_s (MSDN)
- _splitpath -> _splitpath_s (MSDN)
- scanf/sscanf -> sscanf_s (MSDN)
- snscanf -> _snscanf_s (MSDN)
- strlen -> strnlen_s (MSDN)

Однако использование небезопасных функций – это лишь видимая часть айсберга проблем кибербезопасности, возникающих в программных средах. Бывают ситуации, которые можно предусмотреть заранее и заранее предупредить небезопасное поведение программ (переполнение буфера, замену форматных строк и т.д.) В большинстве случаев ошибки, приводящие к проблемам безопасности (уязвимости программного кода) случаются только при стечении обстоятельств (несколько раз вызывается одна и та же функция с разными аргументами и разным поведением).

Другим случаем ошибок являются утечки памяти и ресурсов. Утечка памяти характерна для всех языков программирования, где разрешена прямая работа с указателями памяти и ее выделением (языки C/C++). К данной ситуации приводит выделение памяти и отсутствие процедуры ее освобождения. Подобным образом обстоит дело и с утечкой ресурсов. К примеру, дескриптор к открытому объекту (файлу, процессу, секции памяти) не освобождается программой.

1.4 Исполняемый файл

Результатом компоновки и компиляции файла с исходным кодом программы является новый файл, который является исполняемым. В системах под управлением ОС Windows исполняемые файлы соответствуют формату PE [5]. Подробнее эту информацию студенту рекомендуется изучить в главе 1 в разделе «Заголовки и разделы PE-файла» [5] и раздел «2.6 Проверка информации о PE-заголовке» [7].

2 ПРАКТИЧЕСКАЯ ЧАСТЬ

2.1 Подготовка к работе

В ходе выполнения работы студенту пригодится компьютер с установленными программами:

1. Отладчик уровня ассемблера (подойдет любой отладчик из x64dbg/OllyDbg/WinDbg).
2. Компилятор с языка C (Visual Studio, MinGW).
3. Process Monitor.
4. Process Hacker или Process Explorer.

Хорошей практикой для выполнения ЛР является использование виртуальной машины для возможности сохранения состояний.

2.2 Исходный код программы

Для выполнения практической части требуется скопировать файлы «file.c» (файл с исходным кодом программы) и «file.h» (заголовочный файл). Затем в соответствии с индивидуальным заданием модифицировать код программы.

Т.к. исходный код программы нам известен, то заранее разобьем работу программы на следующие этапы:

1. Инициализация работы программы и выделение памяти в main.
2. Считывание команды из консоли и ее обработка.
3. Выполнение одной из 5 команд в соответствующей функции (команда «end» своей функции не имеет) с использованием API-вызовов к Windows.
4. Цикл выполнения одной из 5 команд о тех пор, пока не введен «end».

2.3 Ход работы

Примечание. Не обязательно приводить все скриншоты, как в данном разделе. Обязательно привести скриншоты в соответствии с требованиями пункта 2.4.

Для получения исполняемого образа нужно скомпилировать файл «file.c», например, с использованием установленного компилятора MinGW командой (Рисунок 3):

```
gcc file.c -o file.exe
```

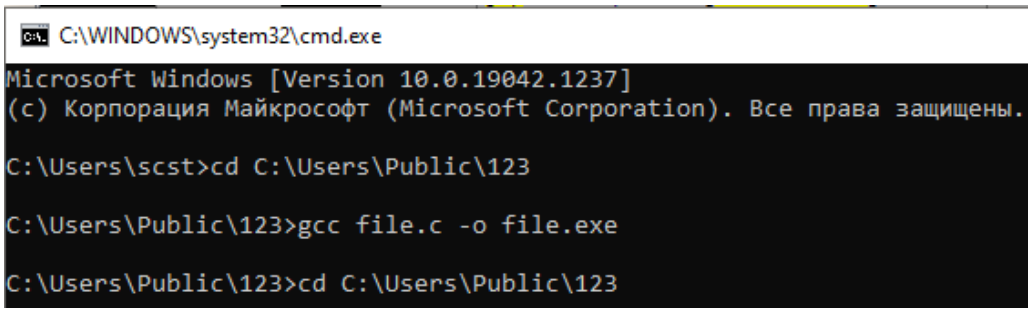

Рисунок 3 – Компиляция и сборка исполняемого файла file.exe

Полученный скомпилированный PE-файл «file.exe» загрузим в отладчик x32dbg (т.к. компиляция была по умолчанию, то создался именно 32-разрядный исполняемый файл). Далее убедимся, что в параметрах отладчика (окно «Параметры», Рисунок 4) установлены следующие настройки и **снята** галочка «Загрузке DLL». Обратить внимание, чтобы была галочка напротив пункта «Точке входа». Данная настройка позволит пропустить загрузку системных библиотек и остановиться на точке входа в исполняемый PE-файл.

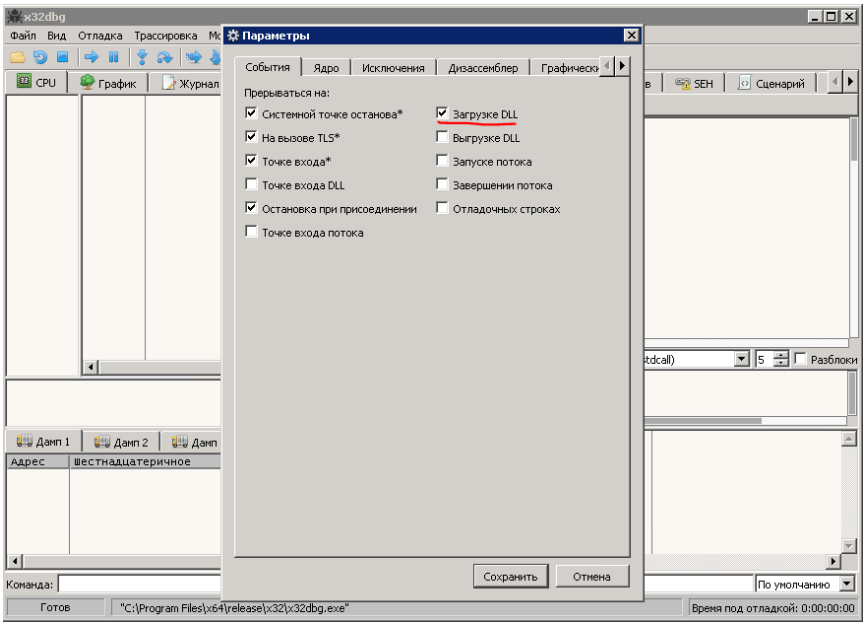

Рисунок 4 – Параметры событий останова отладчика

Нажмем F9 до тех пор, пока отладчик не остановится в EntryPoint (Рисунок 5).

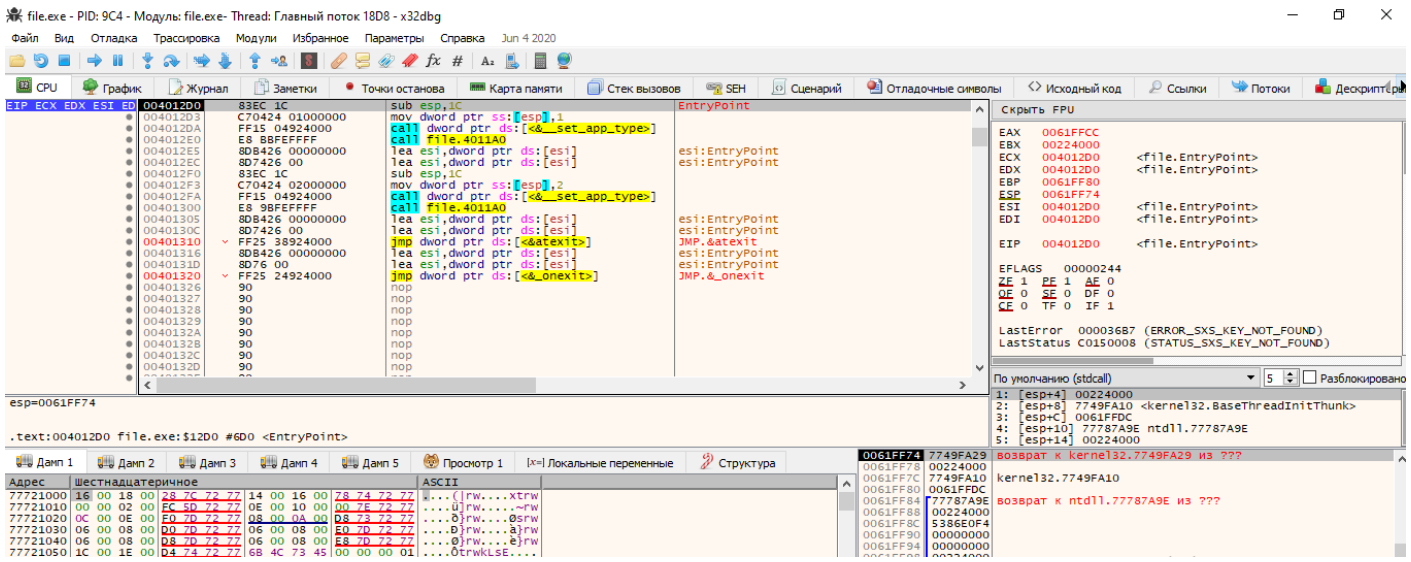

Рисунок 5 – EntryPoint программы

Далее нужно найти функцию, которая соответствует функции main в исходной программе. Поступим следующим образом, зайдем в первый *call 4011A0* (F7) и найдем тот вызов *call*, который будет последним до выхода из программы *_cexit* (Рисунок 6). В нашем случае этот *call* имеет адрес 40122E и переходит на адрес 4019DF. Здесь можно установить программную точку останова (F2).

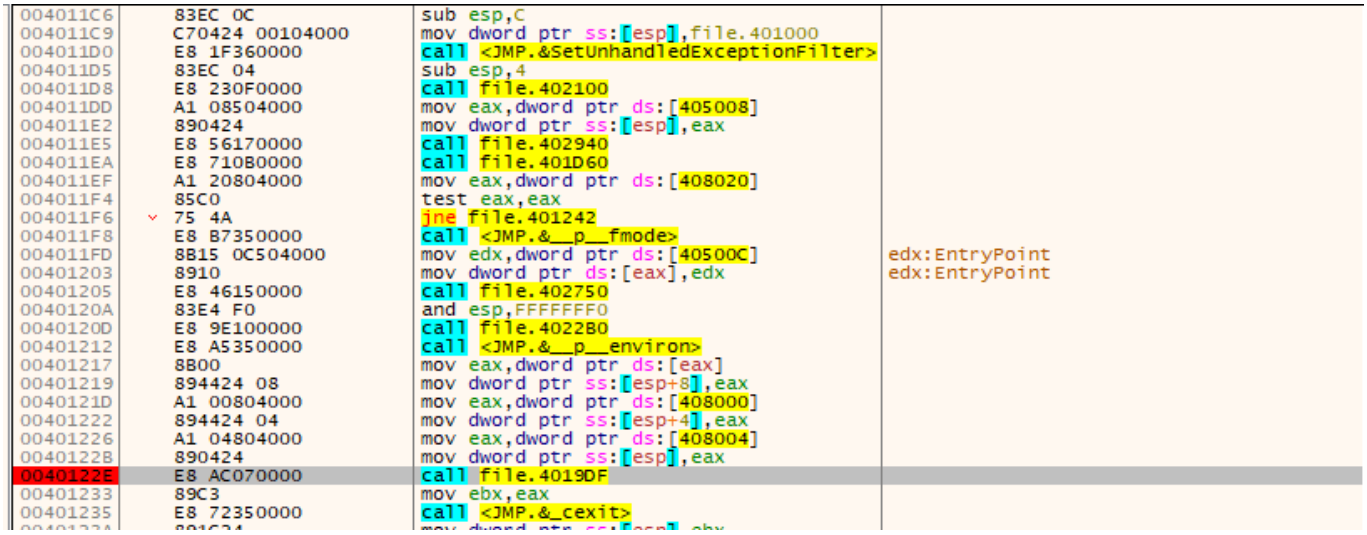

Рисунок 6 – Поиск функции main в исполняемом файле

Для того, чтобы не «проскочить» мимо API-функций *CreateFileA*, *ReadFile*, *WriteFile* можно перейти в «Отладочные символы» найти загруженный модуль *kernel32.dll* и в поиске найти все интересующие библиотечные функции (Рисунок 7) и установить точки останова.

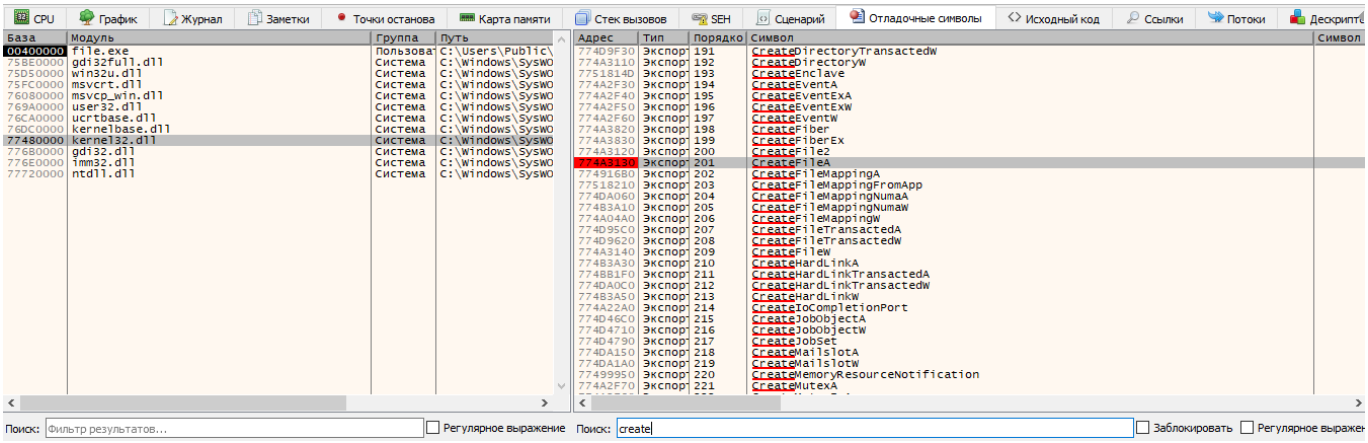

Рисунок 7 – Поиск библиотечных API-символов и установка точек останова

Если перейти в функцию по адресу 4019DF, то можно увидеть такую картину (Рисунок 8).

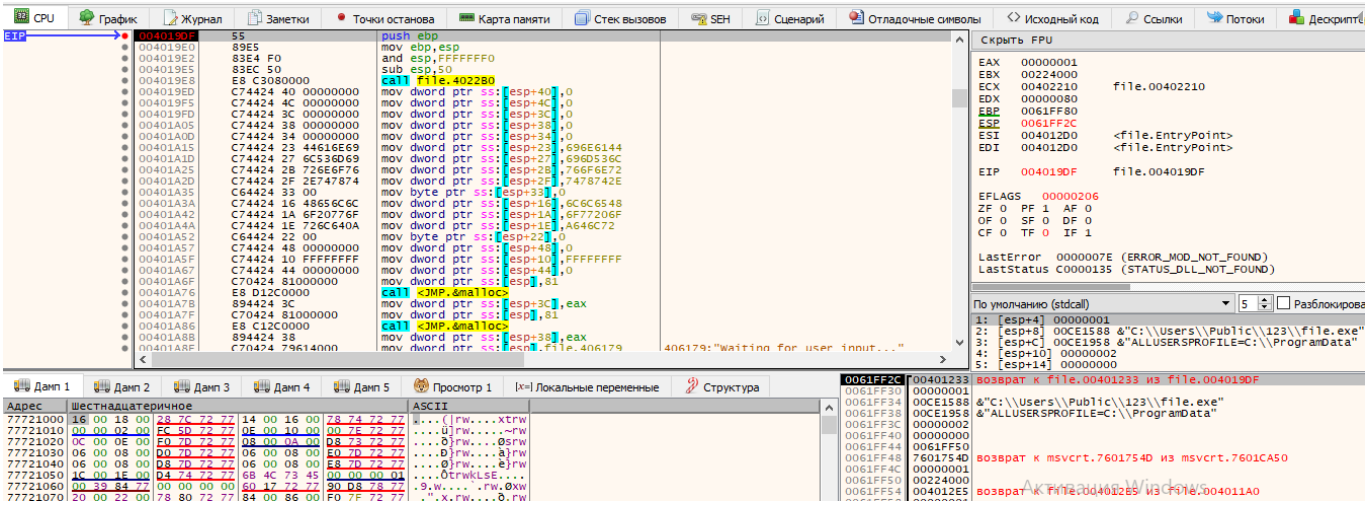

Рисунок 8 – Функция main

Дойдем до функций выделения памяти с помощью *malloc* (Рисунок 9) в main. Сразу за соответствующими *call malloc* поставим программные точки останова с целью изучения адресов, которые выдаст ОС Windows после вызова *malloc* (адрес воз-

вращается функцией через *EAX*, если нажать на него ПКМ и «Перейти к дампу» –> Дамп 2)

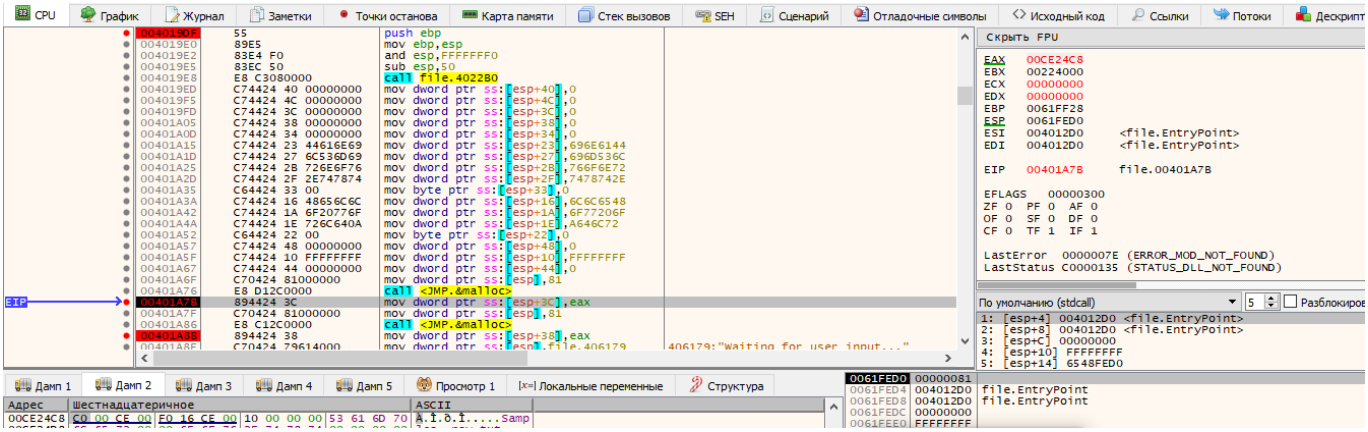

Рисунок 9 – Адреса после вызовов *malloc*

В функции *main* вызывается *malloc* дважды (Рисунок 10): для выделения памяти под переменную *buf* (куда попадают вводимые данные с консоли) и переменную *command* (первое слово из *buf*).

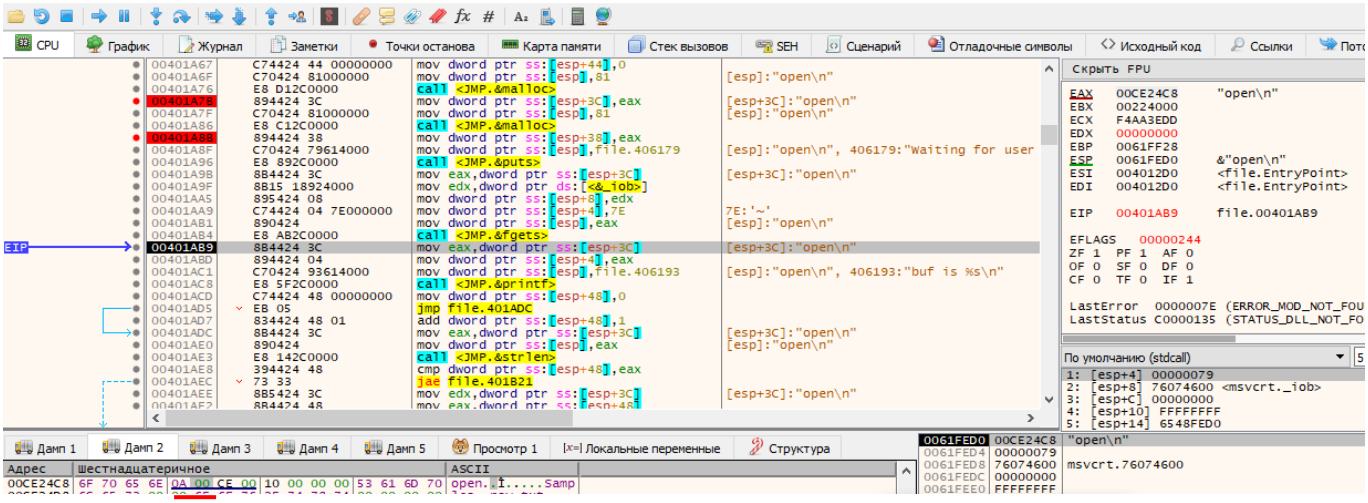

Рисунок 10 – Содержимое памяти по указателю *buf* после набора данных из консоли

Стоит отметить, что после вызова *fgets* функция добавит нуль-терминатор в строку автоматически после спецсимвола «\n» (Рисунок 10). Однако нужно учесть, что так бывает не всегда: если вызывается функция *strncpy* (более безопасный аналог *strcpy*) с количеством символом, равным длине строки без нуль-терминатора, то

он записан не будет (Рисунок 11). Тогда его надо записать отдельным действием, либо учесть при копировании.

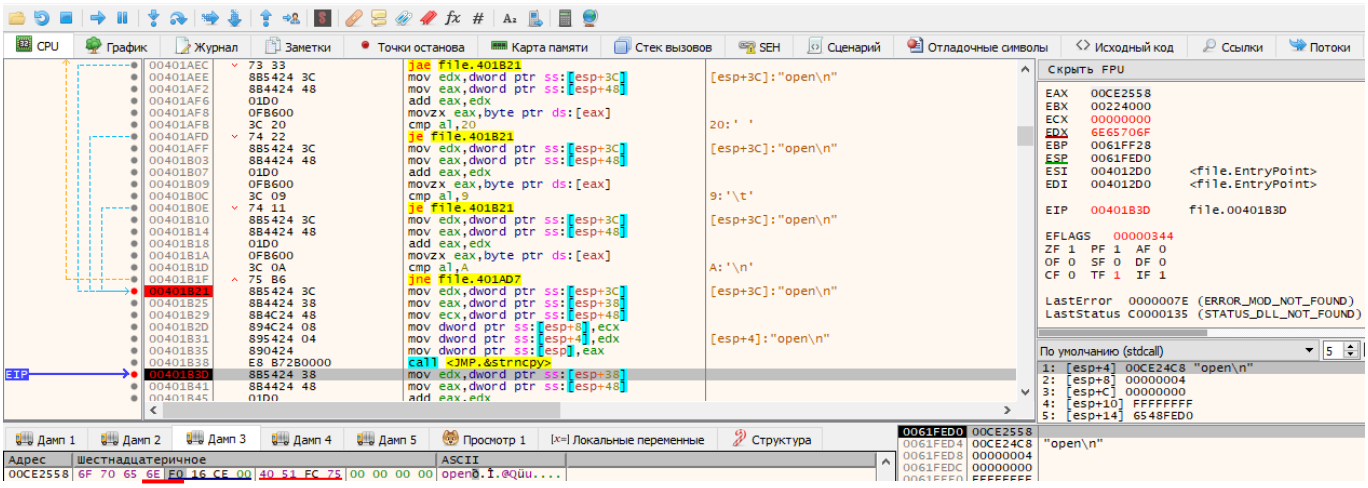

Рисунок 11 – Содержимое памяти по указателю *command* после копирования из *buf*

Если в консоль записать значение «open», то после проверки условий будет вызвана функция по адресу 4015DD. В ней будет в самом начале произведена проверка валидности строки с именем файла (функция находится по адресу 401410). Затем производится вызов API-функции *CreateFileA* (Рисунок 12).

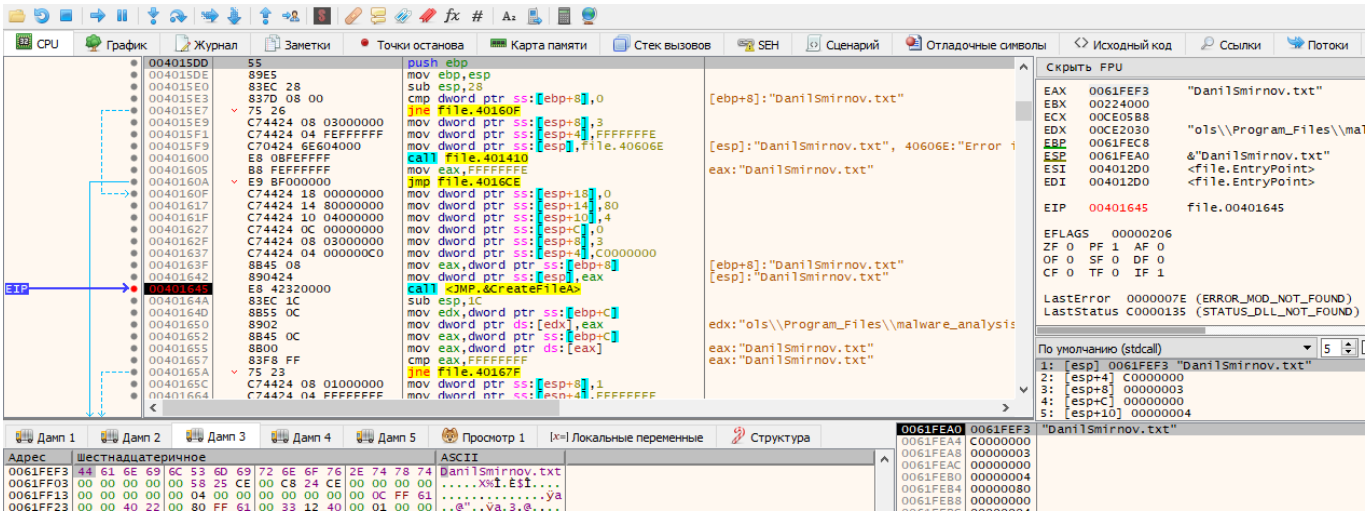

Рисунок 12 – Состояние перед вызовом *CreateFileA*

После перехода на адрес 401645 и нажатия на шаг с заходом (F7) можно посмотреть, как будет вызываться функция внутри *kernel32.dll* (для выхода по завершению функции можно нажать на Ctrl + F9).

После вызова API-функции *CreateFileA* можно увидеть, что в *EAX* записано значение *000000F4* (согласно спецификации с MSDN [8], это и есть хендл открытого файла, Рисунок 13).

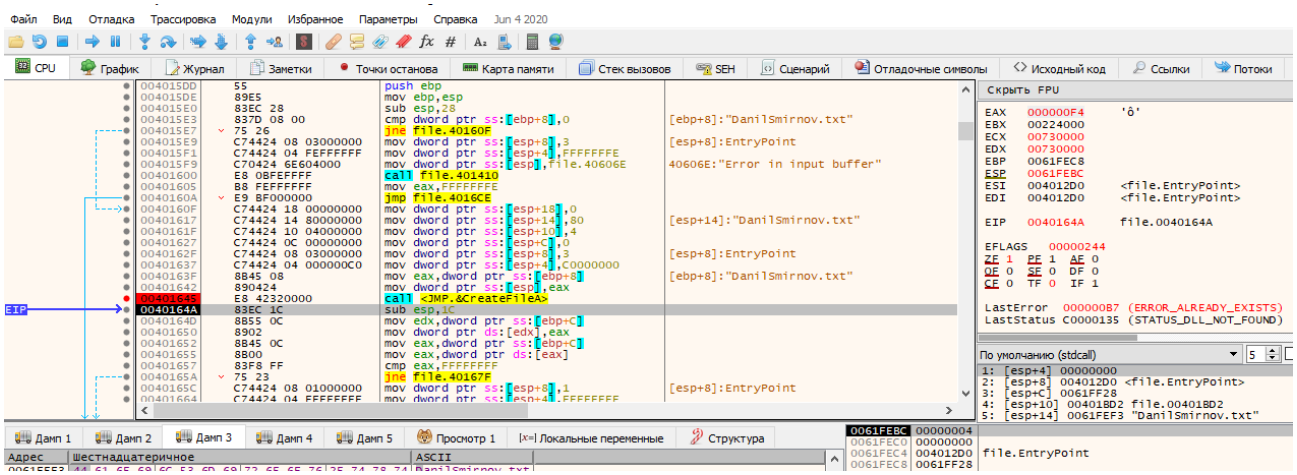

Рисунок 13 – Открытие файла с помощью WinAPI *CreateFileA*

То же значение хендла можно увидеть и в программе Process Hacker [9] (или Process Explorer из набора Sysinternals), если выбрать процесс-потомок от отладчика x32dbg.exe и выбрать его свойства и вкладку «Handles» (Рисунок 14).

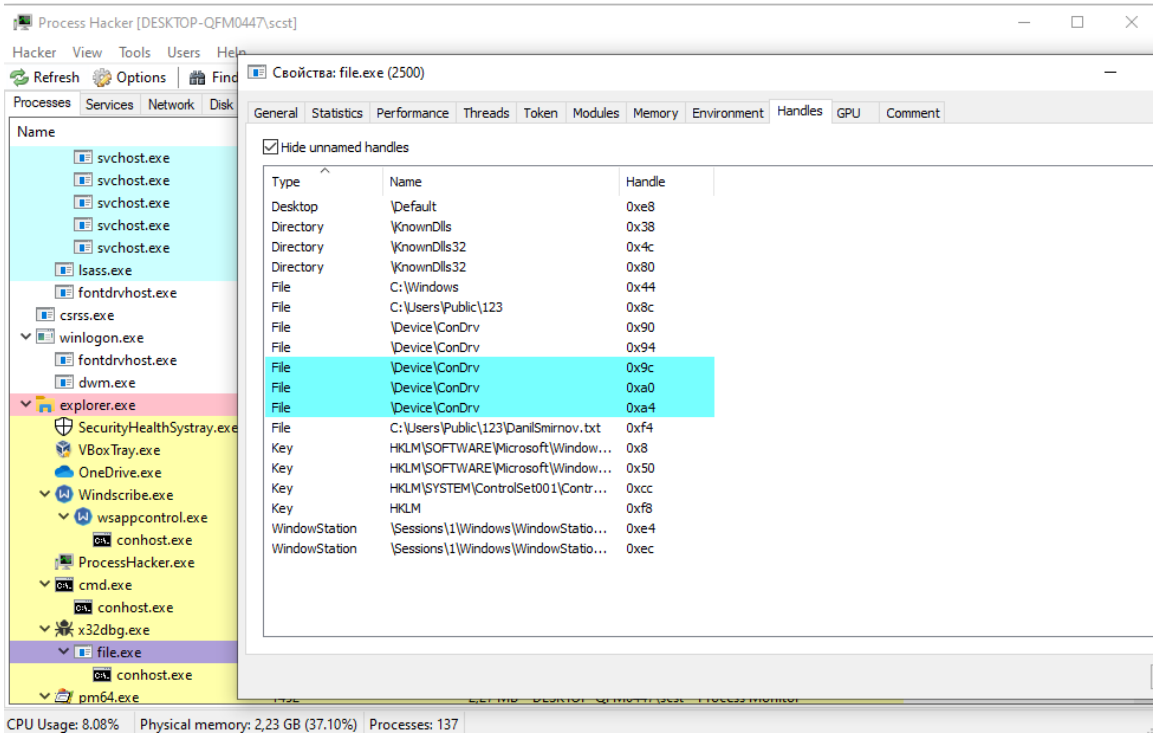

Рисунок 14 – Хендлы исследуемого процесса

Другим интересным способа мониторинга действий программы является Process Monitor из набора утилит Sysinternals [10]. Данная утилита позволяет отслеживать события, происходящие в ОС Windows путем внедрения драйвера-минифильтра файловой системы с дополнительно обрабатываемыми функциями нотификации о запуске процессов PsSetCreateProcessNotifyRoutine [11], загрузке исполняемого образа в память процесса PsSetLoadImageNotifyRoutine [12] и т.д.

Таким образом, использование данной утилиты позволяет отслеживать и файловые операции программ в ОС Windows. В частности, открытие новых файлов так же записывается в отображаемые действия программы «file.exe», если выставить фильтр «Process Name is file.exe» (Рисунок 15). В ней так же видно последовательность вызовов различных слоев обертки WinAPI по стеку, начиная с загрузки исполняемого образа в память процесса в пользовательском режиме и заканчивая системными службами в режиме ядра (NtCreateFile, самая верхняя операция в ntoskrnl.exe, Рисунок 15).

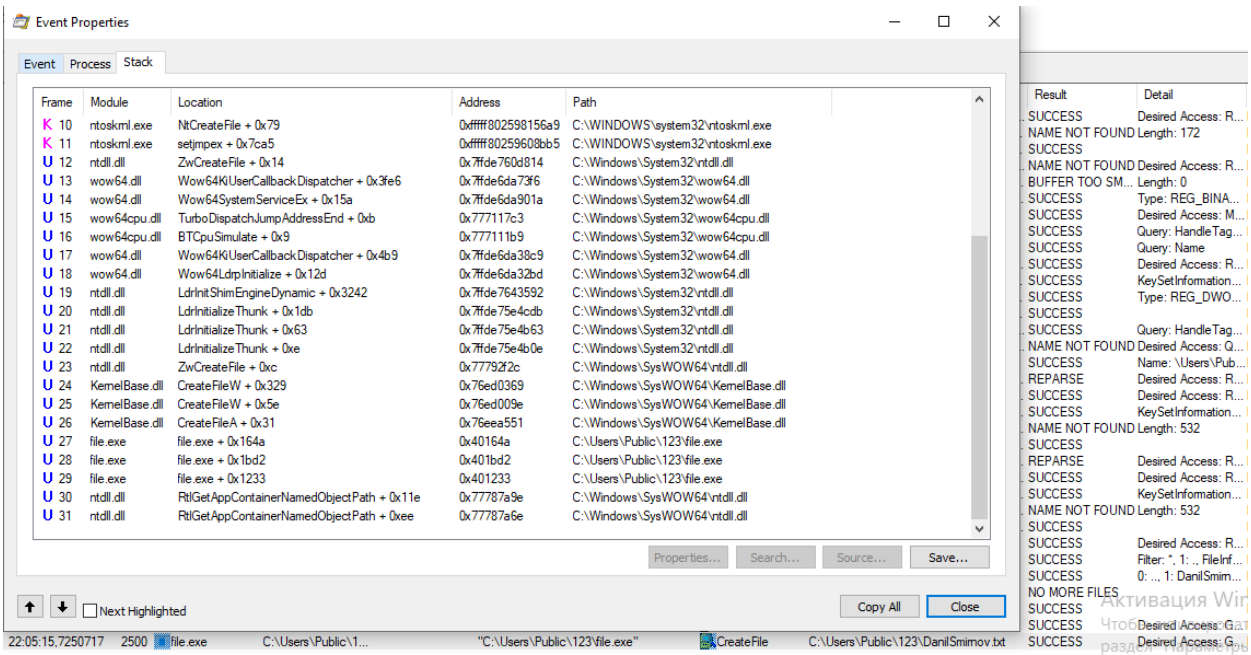

Рисунок 15 – Просмотр событий программы «file.exe» в Process Monitor

2.4 Индивидуальное задание

Индивидуальное задание:

1. Получить файлы «file.c» и «file.exe».
2. Скачать и настроить необходимые инструменты в соответствии с инструкцией из раздела 2.1, затем запустить их.
3. **Изменить** значение в переменной *chFileName* на свои «**ИмяФамилия.txt**» без пробелов на английском языке.
4. Перекомпилировать файл.
5. Запустить файл в отладчике x32dbg.
6. Написать отчет, в котором отразить основные этапы со скриншотами:
    a. Выделение памяти с помощью *malloc* в *main*. В дампе показать участок новой памяти, привести это на том же скриншоте.
    b. Обработку команды, полученной от пользователя в функции *fgets*. В дампе показать область памяти, куда записались данные от пользователя. Привести это на том же скриншоте.
    c. Вызов API-функции *CreateFileA* после написания команды «*open*» в консоль, где указаны все аргументы функции в стеке – в частности, выде-

лить на скриншоте имя файла, которое должно быть индивидуальным. После вызова показать возвращаемое значение в регистре *EAX* и интерпретировать, что оно означает (хендл, количество байт, область памяти и т.д.). Сравнить со значением в Process Explorer (как на Рисунок 14) и Process Monitor (Рисунок 15).

d. Вызов API-функции *WriteFile* после написания команды «*write*» в консоль, где указаны все аргументы функции в стеке. После вызова показать *в дампе то место, какие данные записались в файл* и привести на скриншоте, либо *показать возвращаемое значение в регистре EAX* и интерпретировать, что оно означает (хендл, количество байт, область памяти и т.д.) и привести на скриншоте. Сравнить со значением Process Monitor (Рисунок 15).

e. Вызов API-функции *ReadFile* после написания команды «*read*» в консоль, где указаны все аргументы функции в стеке. После вызова показать *в дампе то место, какие данные были считаны из файла* и *показать возвращаемое значение в регистре EAX* и интерпретировать, что оно означает (хендл, количество байт, область памяти и т.д.) и привести на скриншоте. Сравнить со значением Process Monitor (Рисунок 15).

f. *На более высокую оценку*. Найти утечку ресурсов в программе, вывести проблемное место на скриншоте и предложить несколько путей решения проблемы. Один из них реализовать и перекомпилировать программу в соответствии с ним.

g. *На более высокую оценку*. Исправить программу таким образом, чтобы можно было использовать те же команды, но уже с аргументами (для открытия файлов можно было в программе выбрать имя файла, для чтения, записи – используемый хендл и записываемое значение, а для закрытия файла – хендл).

Стоит отметить, что выполнение пунктов от *а* до *е* позволяет получить максимально только 7 баллов из 10. Для оценки 8 и более за лабораторную работу нужно выполнить все пункты от *а* до *е* и сделать хотя бы одно из дополнительных заданий.

Критерии оценки работ:

1. Правильно приведены все требуемые этапы работы по заданию в соответствии с инструкцией к выполнению ЛР №1 в логически структурированном отчёте (6 из 10 баллов за этот пункт).
2. Отчет соответствует ГОСТ 7.32. В нем есть обязательные разделы:
    - титульный лист;
    - введение;
    - основная часть;
    - заключение.

Все рисунки в виде скриншотов пронумерованы и подписаны. Каждый скриншот является информативным (4 из 10 баллов за этот пункт).

3 ЗАКЛЮЧЕНИЕ

В рамках данной работы получены знания:

- о принципах работы ЭВМ;
- об особенностях архитектуры компьютера фон Неймана и x86/x64;
- о работе с объектами в ОС Windows.

В рамках данной работы получены навыки:

- анализ программ с использованием отладчика уровня ассемблера;
- отладка и поиск проблем с использованием средств Sysinternals.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Об информации, информационных технологиях и о защите информации: Федеральный закон от 27.07.2006 № 149 // Российская бизнес-газета. – 2006 – № 4131.
2. Архитектура компьютера [Электронный ресурс]. URL: https://ru.bmstu.wiki/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%B0 (дата обращения 08.10.2021)
3. Таненбаум, Э. Архитектура компьютера. 6-е издание / Э. Таненбаум, Т. Остин – 2018.
4. x64dbg. An open-source x64/x32 debugger for windows. [Электронный ресурс]. URL: https://x64dbg.com/#start (дата обращения 08.10.2021)
5. Сикорски, М. Вскрытие покажет! Практический анализ вредоносного ПО. / М. Сикорски, Х. Эндрю. – 2018.
6. Керниган, Б. Язык программирования Си / Б. Керниган, Д. Ритчи. – 2017.
7. Монаппа, К. Анализ вредоносных программ. / К. Монаппа. – 2019.
8. CreateFileA function (fileapi.h) [Электронный ресурс]. URL: https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea (дата обращения 08.10.2021)
9. Process Hacker 2.39 r124 [Электронный ресурс]. URL: https://processhacker.sourceforge.io/downloads.php (дата обращения 08.10.2021)
10. Process Monitor v3.85 [Электронный ресурс]. URL: https://docs.microsoft.com/en-us/sysinternals/downloads/procmon (дата обращения 08.10.2021)
11. PsSetCreateProcessNotifyRoutine function (ntddk.h) [Электронный ресурс]. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-pssetcreateprocessnotifyroutine (дата обращения 08.10.2021)

12. PsSetLoadImageNotifyRoutine function (ntddk.h) [Электронный ресурс]. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-pssetloadimagenotifyroutine (дата обращения 08.10.2021)