Практическая работа №4.1
«Основы отладки на ассемблере. Объекты ОС»

«1» апреля 2024 г.

Москва 2024 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.

ПР – практическая работа.

1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1 Основные положения

Для выполнения практической работы потребуются знания по разделам:

- архитектура компьютера;
- языки программирования – ассемблер и язык высокого уровня (например, язык С);
- принципы работы ОС Windows/Linux 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

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].

Рисунок 1 – Архитектура ЭВМ фон Неймана

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 фундаментальных вопросов информационной безопасности. Одной из основных проблем ИБ является существование эксплойтов для 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 [2].

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 [2]. 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 [2]. По пути разработки систем с данной архитектурой пошли разработчики микроконтроллеров Microchip PIC и Atmel AVR.

1.3 Языки высокого уровня

Для разработки прикладных программ ассемблер давно не используется из-за сложности разработки и отсутствия кроссплатформенности для различных аппаратных архитектур и операционных систем. Для этого созданы 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 (в основном это касается вычислительных задач).

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++** [3] 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 (в примере с **Java** это **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 практических работах будет использоваться язык **C**. 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 C

Язык **C** относится к императивным языкам с типами данных, переменными и операторами управления [3]. 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 **C** предлагается изучить книгу «Язык программирования Си» [3] или один из последних принятых стандартов языка **C**.

1.3.2 Безопасность языка C и программ на языках высокого уровня

Существует широкораспространенная проблема несоответствия введенных пользователем данных и данных, которые предусмотрел программист. Использование так называемых «небезопасных функций» приводит к наиболее критическим проблемам кибербезопасности – уязвимостям в ПО.

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, а объект считается используемым.

1.4 Исполняемый файл

Результатом компоновки и компиляции файла с исходным кодом программы является новый файл, который является исполняемым (или бинарным). Основная метаинформация об исполняемом файле хранится в заголовке, а весь код и данные приведены в секциях. Это справедливо и для ОС Windows, и для ОС Linux.

1.4.1 Заголовок исполняемого файла

В системах под управлением ОС Windows исполняемые файлы соответствуют формату *PE* [4]. Краткое описание заголовков исполняемых файлов под ОС Windows приведено в разделе **1.4.1.1**. Подробнее эту информацию студенту рекомендуется изучить в главе 1 в разделе «Заголовки и разделы PE-файла» [4] и раздел «2.6 Проверка информации о PE-заголовке» [5]. В системах под управлением ОС Linux

используется формат *ELF* для работы исполняемых файлов (коротко – в разделе **1.4.1.2**). Детальная информация описана в разделе «ELF Files» главы «20. Linux Operating System» [6].

1.4.1.1 Заголовок PE/COFF-файла ОС Windows

Формат файлов *PE/COFF* унаследован из ОС MS-DOS, где основными исполняемыми файлами были файлы с расширением *.COM*. С появлением ОС Windows NT данный формат был унаследован и модифицирован для файлов *.EXE* и *.DLL*. Визуализация данных разделов приведена на рисунке (Рисунок 2) [7].

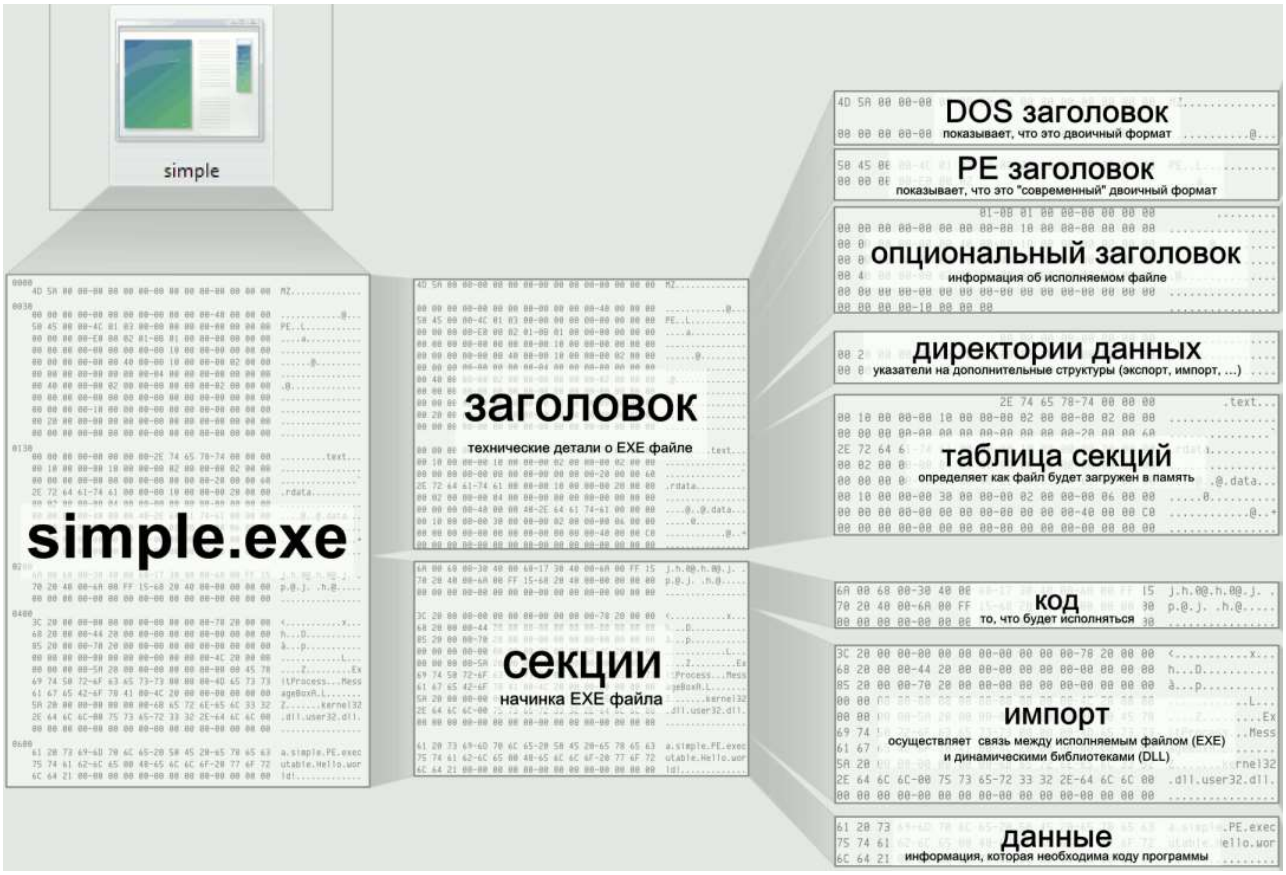

Рисунок 2 – Визуализация разделов PE-файла

DOS-заголовок содержит много атрибутов, большинство из которых в современных ОС Windows уже не используются. В настоящее время используются поля: *e_magic* со значением «4D 5A» и *e_lfanew* со смещением PE-заголовка (обычно

0x40 или 0x80). По смещению *e_lfanew* вычисляется нахождение PE-заголовка с характерными значениями «50 45 00 00». Далее идут характерные поля, необходимые для размещения всех основных разделов и секций программы в памяти процесса.

Метаинформацию по PE-файлу можно посмотреть в программе **PEStudio** [8]. В ней предоставляется основная информация по magic-байтам, считаются хеши и энтропия основных секций, а также производится обогащение на основе открытых источников, например, VirusTotal (Рисунок 3).

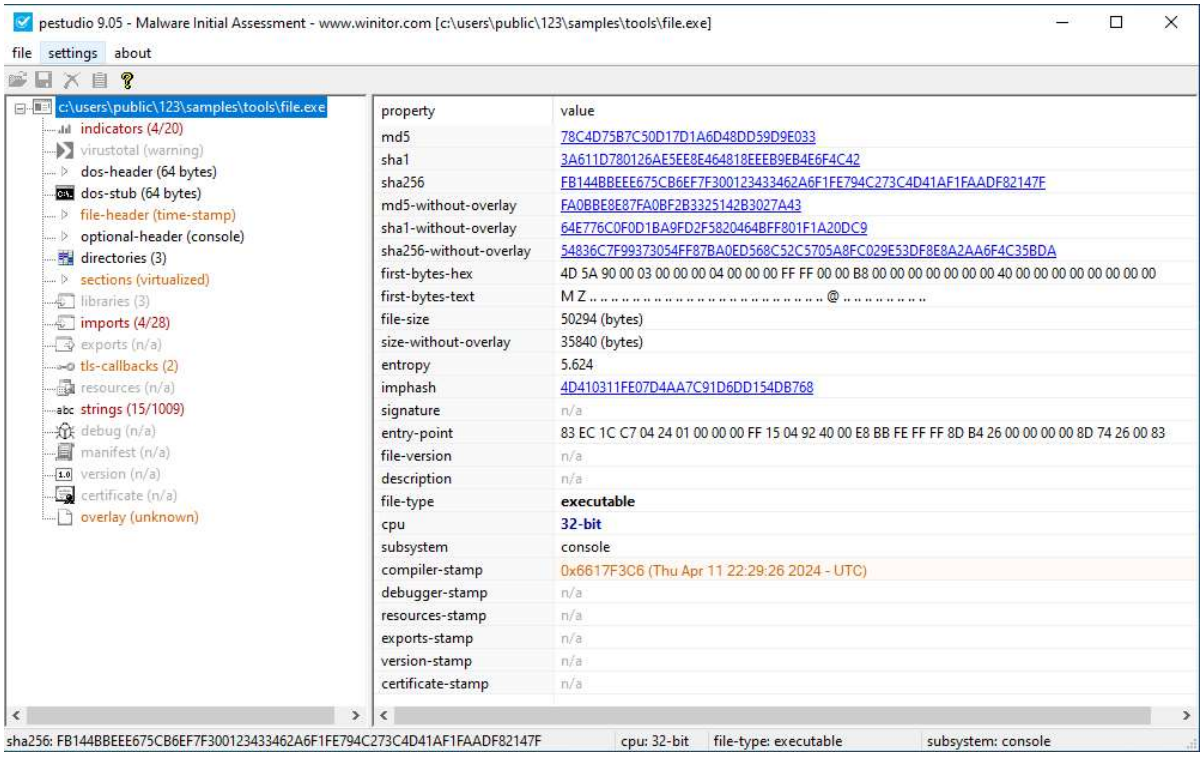

Рисунок 3 – Вывод программы PEStudio

Более мощным инструментом для работы с PE-файлами является **CFF Explorer** [9]. Он так же содержит в себе парсеры заголовков и секций, а также встроенный hex-редактор с возможностью изменения основных свойств PE-файла, например, *DllCharacteristics* (Рисунок 4).

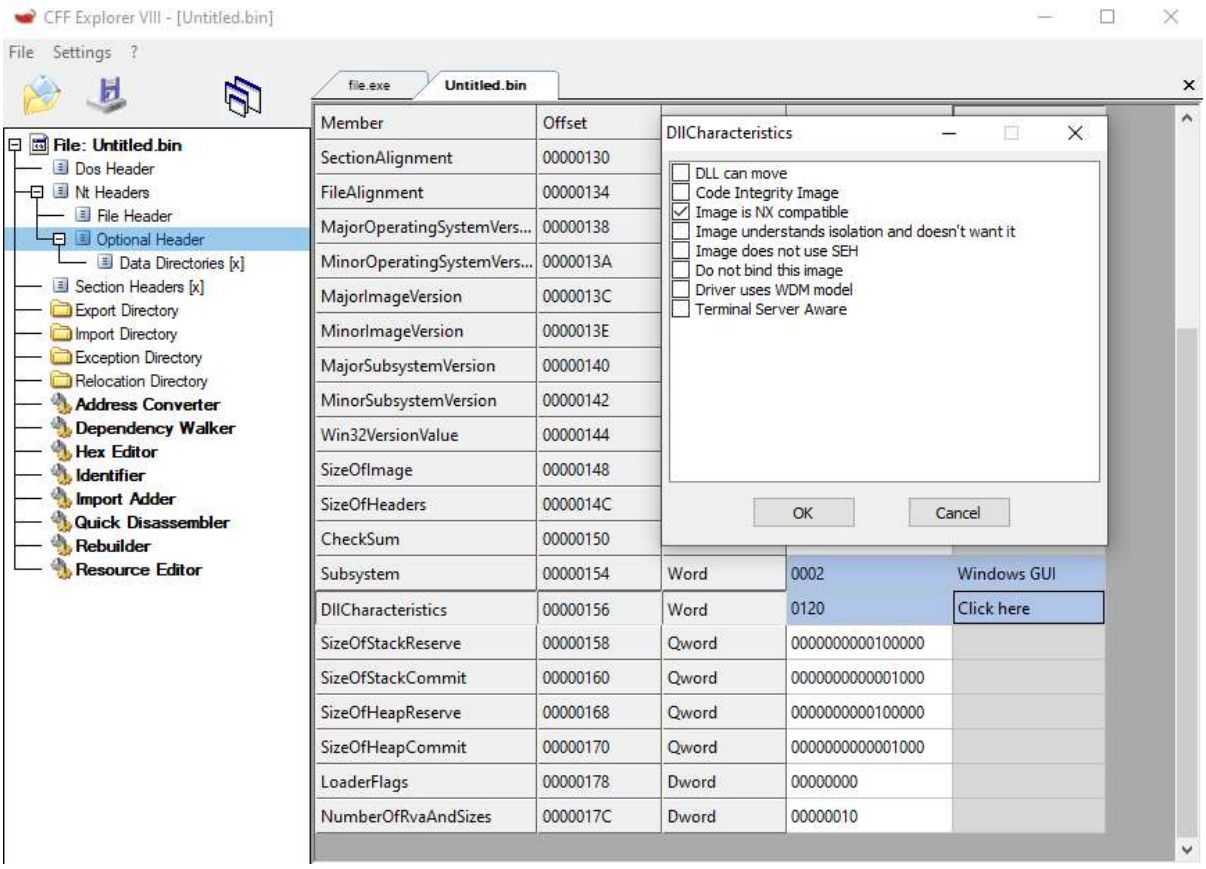

Рисунок 4 – Вывод программы CFF Explorer

1.4.1.2 Заголовок ELF-файла ОС Linux

Формат файлов *ELF* был создан UNIX System Laboratories в рамках создания Application Binary Interface (ABI) и применялся в Unix-подобных системах. Данный формат также был унаследован и в ОС Linux. Так же, как и у PE-файлов, у ELF-файлов есть magic-байты, в данном случае «7F 45 4C 46». Особенности ELF-файла можно просмотреть утилитой **readelf** (Рисунок 5).

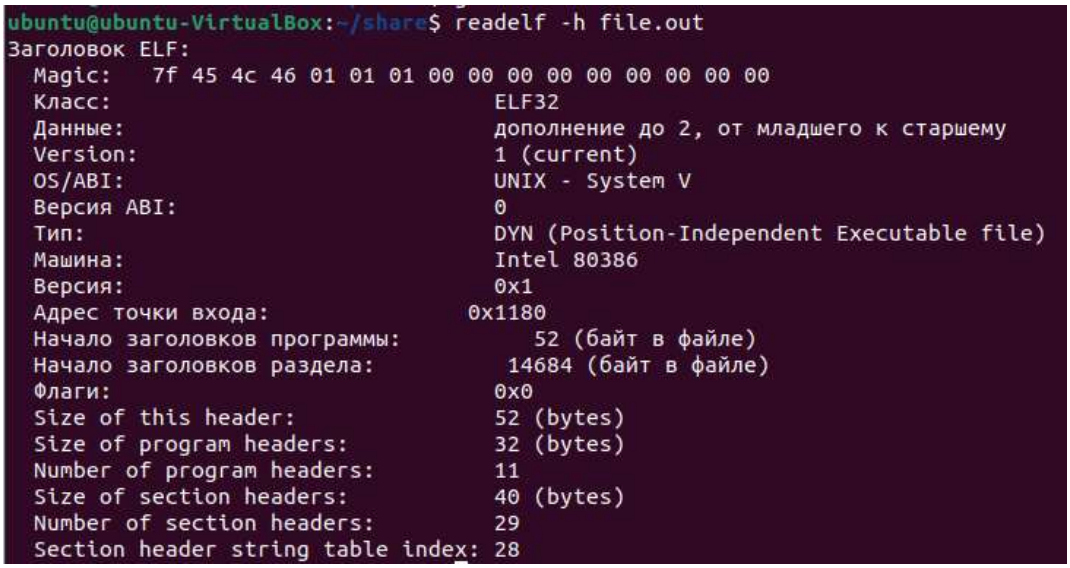

Рисунок 5 – Вывод утилиты **readelf**

1.4.2 Секции исполняемого файла и исполняемый код

Рассмотрим более подробно секции исполняемого файла. Данные секции содержат в себе исполняемый код, основные инициализированные переменные и константы, а также глобальные и другие данные. Названия всех секций зависят от компилятора. Секции кода обычно имеют название *.text* (характерно для большинства компиляторов типа **Visual Studio** или **gcc**) или *.CODE* (характерно для среды **Borland**).

Для изучения бинарного скомпилированного кода используются дизассемблеры (статический анализ) или отладчики с опцией дизассемблирования (динамический анализ). На рисунке (Рисунок 6) приведен скриншот отладчика **x64dbg** (**x32dbg**) [10], используемого для отладки программ на уровне ассемблера x64 (x86) архитектуры.

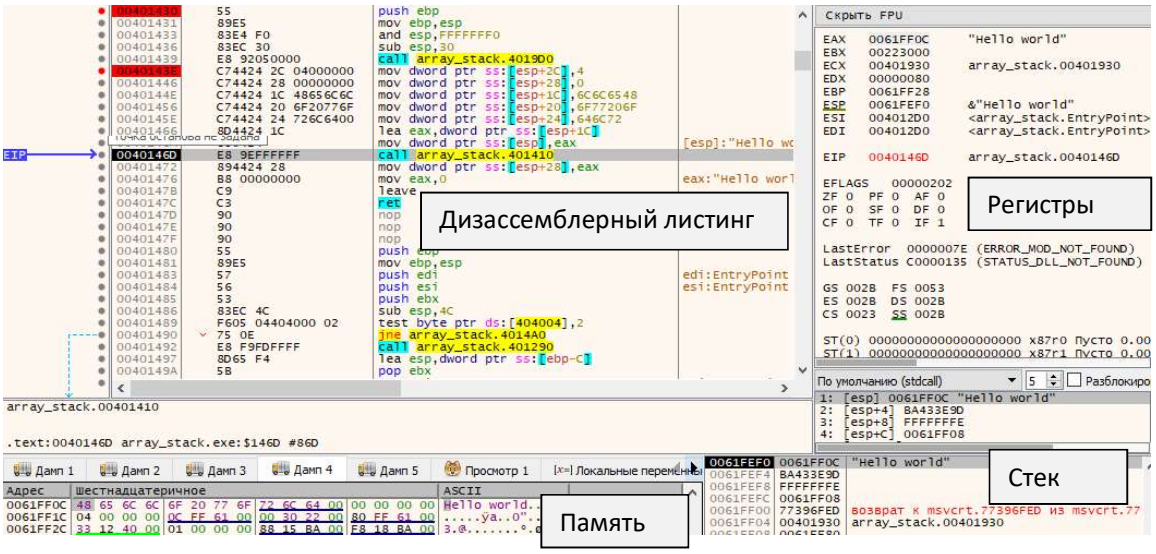

Рисунок 6 – Окно отладчика **x64dbg**

В главном окне приводится листинг команд процессора. В левом нижнем углу скриншота приведена память всего процесса. В ней хранится сам исполняемый код и все необходимые данные для работы программы. В правом верхнем углу приведены регистры: регистры общего назначения, регистр флагов, регистр указателя текущей инструкции и т.д. В правом нижнем углу приведен стек – часть памяти процесса, отведенная под определенный поток. Так как стек является частью памяти процесса, то его точно так же можно просмотреть и в левом нижнем углу. Стек выполняет важную служебную функцию: в стеке хранятся значения локальных переменных, а также *адреса возврата в функции*, *адреса переходов на функции* и другие вспомогательные данные. В ОС Windows у каждого потока свой стек.

Аналогичную функцию отладки выполняет и отладчик **edb-debugger** в ОС Linux (Рисунок 7). Его графический интерфейс и горячие клавиши аналогичны соответствующим элементам управления **x64dbg**.

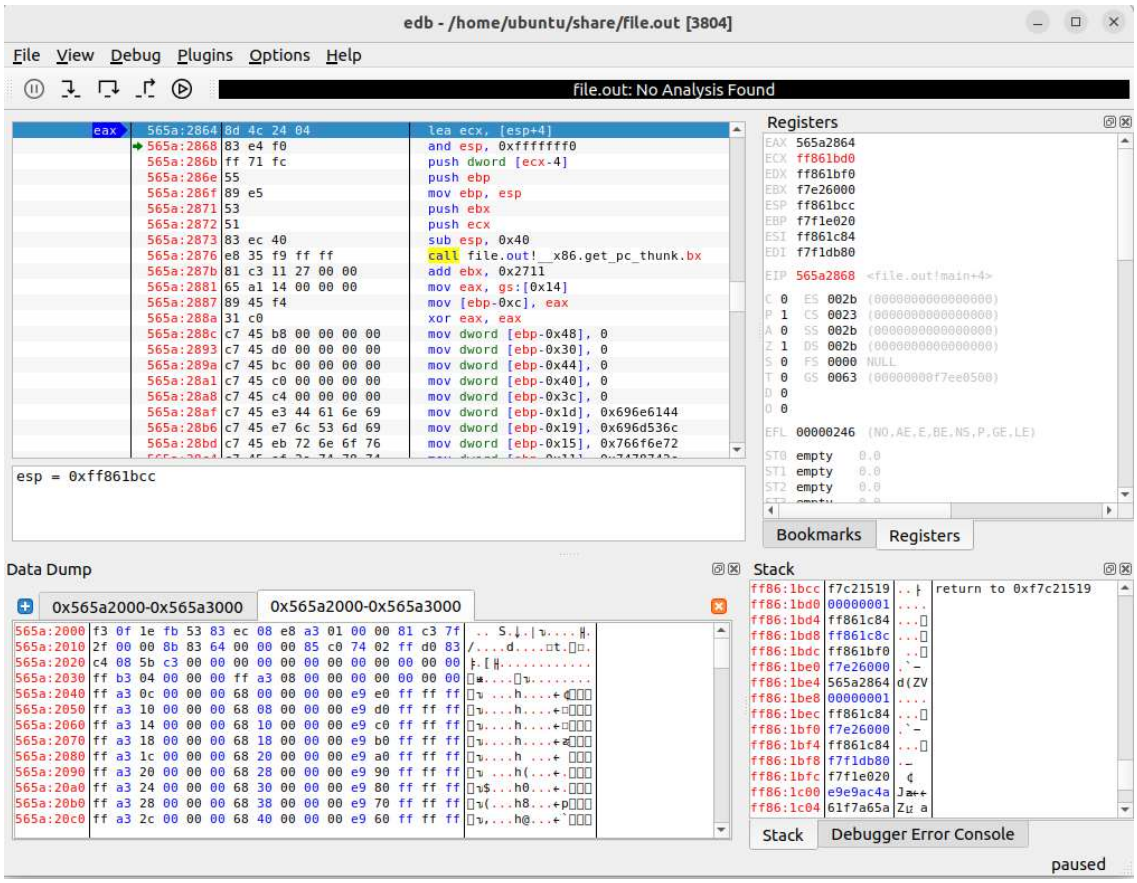

Рисунок 7 – Окно отладчика **edb-debugger**

1.4.2.1 Вызов функций языка С с использованием стека

Остановимся более подробно на примерах инструкций исполняемого кода. Среди важных примеров инструкций можно выделить следующие:

1. PUSH EBP – записывает на вершину стека значение регистра EBP, или регистра *указателя базы кадра стека*. Обычно это начало *пролога функции* [11] – участка кода, с которого начинается любая функция языка С.
2. MOV EBP,ESP – записывает в регистр EBP значение регистра ESP, или регистра *указателя на вершину стека*. Это следующая инструкция *пролога функции*. Таким образом, пролог функции сохраняет *указатель базы* (EBP) в стек, затем в регистр EBP сохраняет текущий *указатель на вершину стека* ESP.
3. MOV ESP,EBP – записывает в регистр ESP значение регистра EBP. Обычно это первая инструкция *эпилога функции*, то есть области кода, обратной *прологу функции*.

4. POP EBP – извлекает с вершины стека значение и записывает в регистр EBP. Тоже является частью *эпилога функции*.
5. LEAVE – является аналогом пары команд MOV ESP,EBP и POP EBP
6. MOV BYTE PTR [EAX], 10h – записывает в ячейку памяти с адресом, хранящимся в регистре EAX значение 10h (в десятичной системе – 16).
7. JMP EDI – безусловный переход на адрес, содержащийся в регистре EDI.
8. CMP EAX,80h –> JZ 4300000 – связка двух команд сравнения значения регистра EAX с числом 80h (в десятичной системе – 128) и переход на адрес 4300000.

Рассмотрим более подробно связь команд и работы стека на примере ниже.

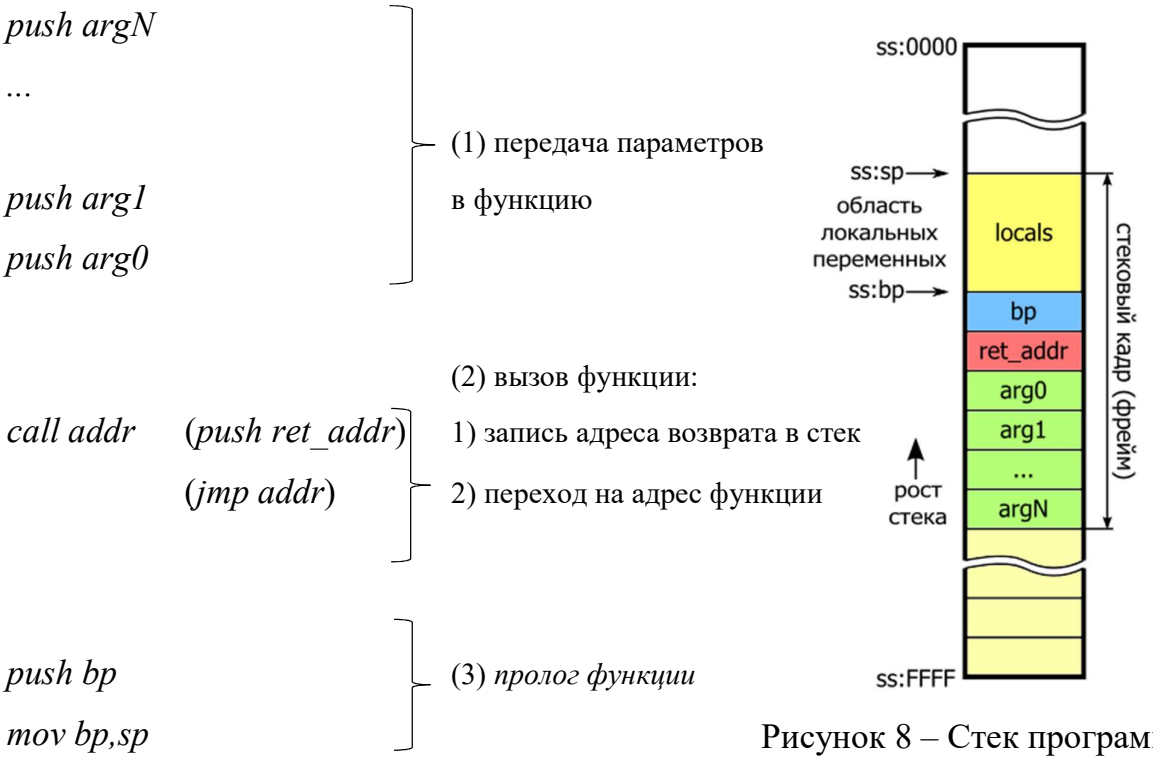

Рисунок 8 – Стек программы

На примере приведен дизассемблерный листинг, состоящий из упаковки (1) аргументов в стек для последующей их передачи в функцию; собственно, сам вызов функции (2); и *пролог функции* после перехода в нее (3). Данному листингу соответствует рисунок (Рисунок 8), на котором приведен стек, в который передаются

параметры, адрес возврата, регистр EBP и выделение локальных переменных. Как видно на рисунке, то стек возрастает от старших адресов к младшим.

1.4.2.2 Куча и стек в языке C

В рамках более качественного ознакомления с темой архитектуры компьютера и ассемблера студенту предлагается (это не обязательно) изучить главу 4 [4] 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. На рисунке (Рисунок 6) приведен пример точки останова на адресе, который выделен красным цветом. Для активации программной точки останова можно использовать горячую клавишу F2.
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.

1.5 API-функции ОС и системные вызовы ОС

Прикладные программы, которые разрабатываются для решения задач конечного пользователя, взаимодействуют с операционной системой для доступа к ресурсам. Например, в ОС Windows для доступа к ресурсам программа обращается к исполнительной подсистеме, реализованной в виде различных DLL-библиотек. В каждой библиотеке реализован системный код, который позволяет подготовить нужные аргументы функций, предварительно их обработать, чтобы затем передать их

системным сервисам в режиме ядра. В ОС Linux реализован подобный механизм с помощью SO-файлов.

1.5.1 API-функции ОС Windows

Обращение программ к ОС Windows реализован в виде вызова определенных API-функций ОС Windows. Характерным примером является открытие файла с помощью API-функции *CreateFileA* [12]. Хотя открыть файл в ОС Windows можно различными способами (например, с использованием функции языка **C** *fopen*), большинство из них будут сводиться к вызову более низкоуровневой недокументированной процедуры *NtCreateFile* из *ntdll.dll*.

Такой последовательный вызов вложенных друг в друга API-функций ОС можно отследить в отладчике. Для этого можно перейти в «Отладочные символы» (*символами* называют функции, у которых можно восстановить название: например, любая импортируемая функция из сторонних библиотек – это *символ*) найти загруженный модуль (например, *kernel32.dll*) и в поиске найти все интересующие библиотечные функции (Рисунок 9) и установить точки останова. Это позволяет увидеть вызов API-функции *CreateFileA* из *kernel32.dll*.

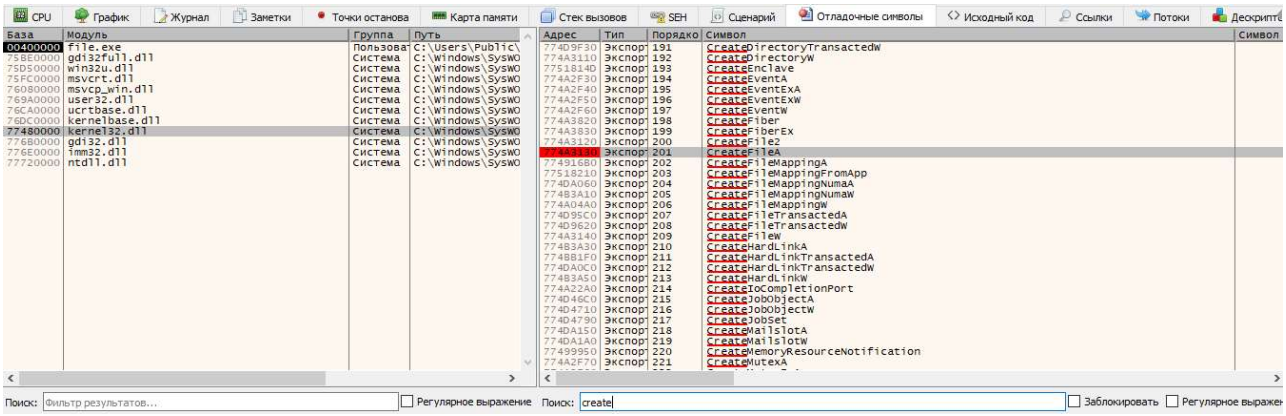

Рисунок 9 – Поиск библиотечных API-символов и установка точек останова

Если провалиться внутрь библиотечной функции путем нажатия кнопки «шаг с заходом» (F7), то можно посмотреть, как будет вызываться функция внутри *kernel32.dll* (для скорого выхода и «проскакивания шагов» из функции по ее

завершению можно нажать на Ctrl + F9). При вызове API-функции *CreateFileA* из *kernel32.dll* будет производиться вызов функции из *kernelbase.dll*, затем недокументированной API-функции *NtCreateFile* из *ntdll.dll* (вся цепочка вызово будет раскрыта более подробно ниже). Чтобы не заблудиться в ассемблерном листинге внутри DLL, рекомендуется периодически обращаться к карте памяти и отслеживать, в какой области в настоящее время находится управление (Рисунок 10).

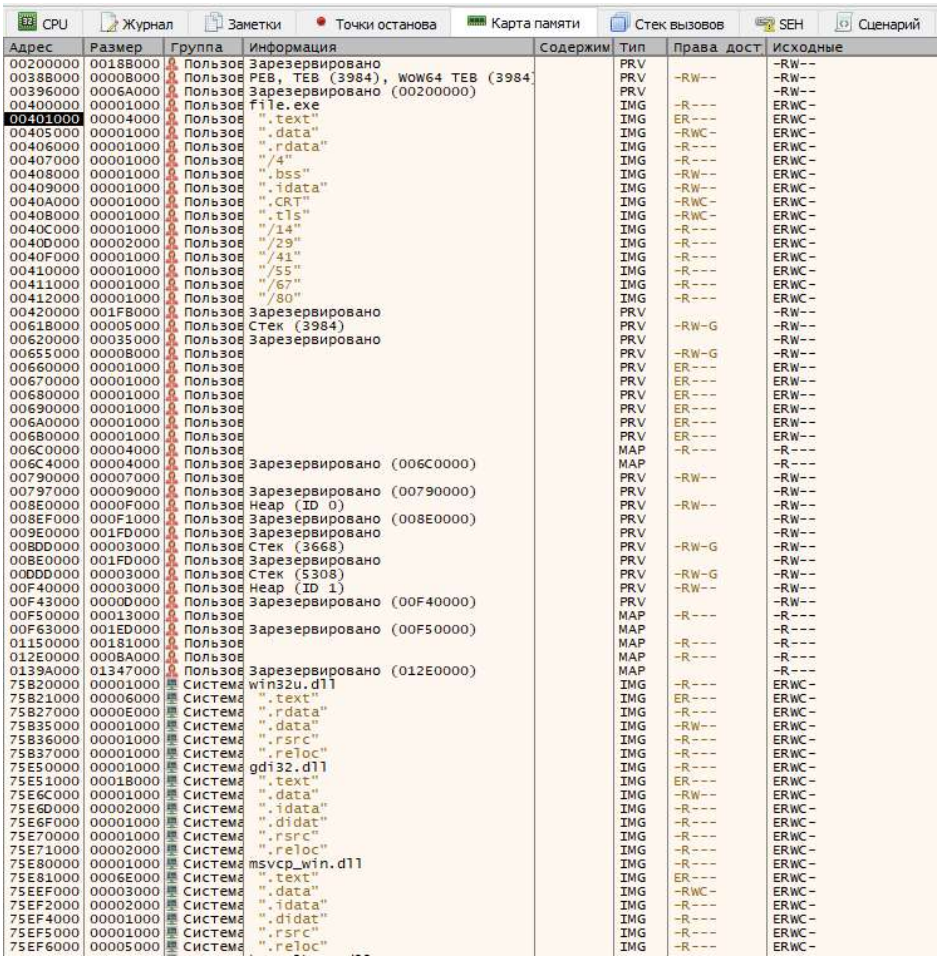

Рисунок 10 – Карта памяти процесса

Если речь идет об отслеживании определенных API-функций ОС Windows, то одним из способов мониторинга действий программы является **Process Monitor** из набора утилит **Sysinternals** [12]. Данная утилита позволяет отслеживать события, происходящие в ОС Windows путем внедрения драйвера-минифильтра файловой системы с дополнительно обрабатываемыми функциями нотификации о запуске

процессов *PsSetCreateProcessNotifyRoutine* [14], загрузке исполняемого образа в память процесса *PsSetLoadImageNotifyRoutine* [15] и т.д.

Таким образом, использование данной утилиты позволяет отслеживать и файловые операции программ в ОС Windows: в частности, вызов программой «file.exe» определенных API-функций для работы с файловой системой, если выставить фильтр «Process Name is file.exe» (Рисунок 11).

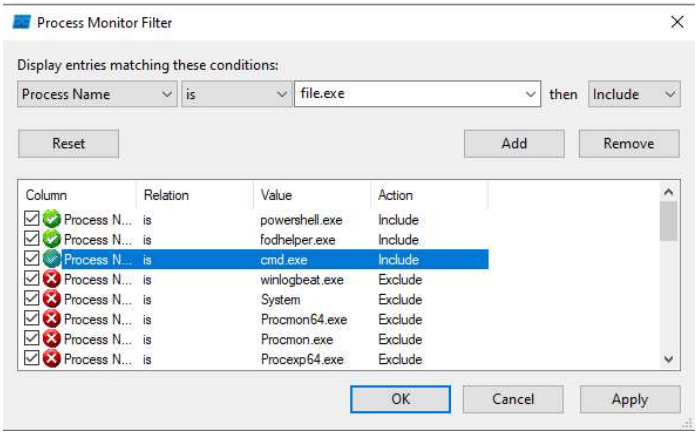

Рисунок 11 – Применение фильтра в Process Monitor

В **Process Monitor** можно отследить 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. В данном случае произведено отслеживание вызова API-функции *CreateFileA*, которая в конечном итоге выполнила системный вызов *NtCreateFile* (самая верхняя операция в *ntoskrnl.exe*, Рисунок 12).

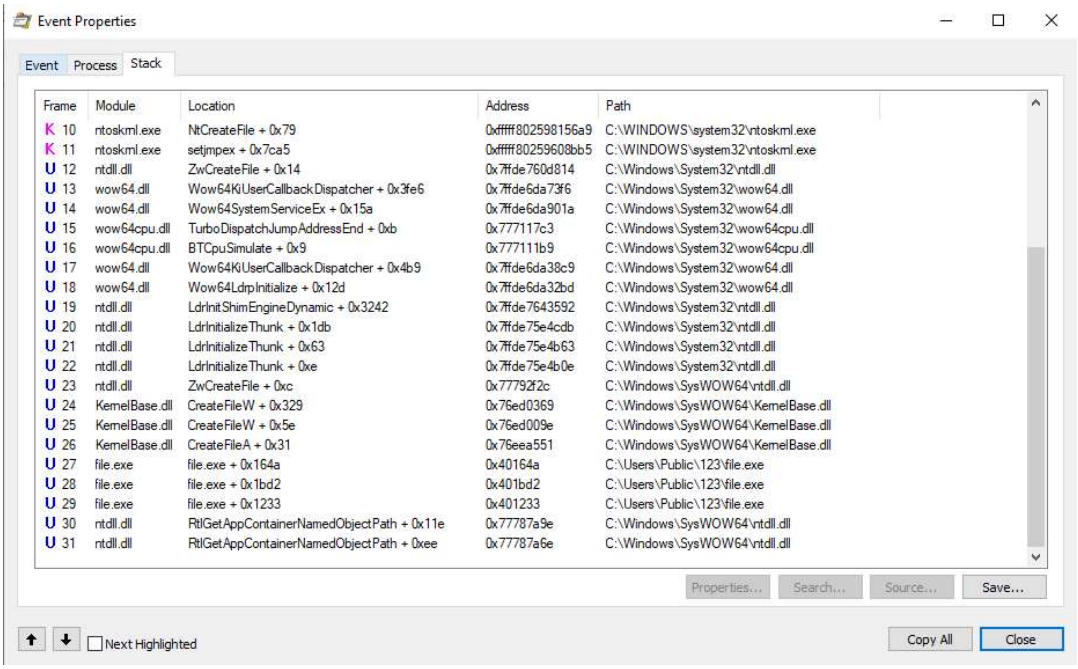

Рисунок 12 – Просмотр событий программы «file.exe» в Process Monitor

1.5.2 API-функции ОС Linux

Для мониторинга системных вызовов в ОС Linux можно использовать утилиту **strace** (Рисунок 13). Она использует библиотеку *ptrace* для перехвата системных вызовов. О том, как можно использовать утилиту, приведено в п. **2.4.2**.

```
ubuntu@ubuntu-VirtualBox:~$ strace -f ./file.out
execve("./file.out", ["./file.out"], 0x7fff3806c8e8 /* 45 vars */) = 0
[ Process PID=4396 runs in 32 bit mode. ]
brk(NULL)                               = 0x57ecc000
arch_prctl(0x3001 /* ARCH_??? */, 0xffb81678) = -1 EINVAL (Недопустимый аргумент)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xf7f73000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (Нет такого файла или каталога)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (Нет такого файла или каталога)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
statx(3, "", AT_STATX_SYNC_AS_STAT|AT_NO_AUTOMOUNT|AT_EMPTY_PATH, STATX_BASIC_STATS, {stx_mask=STATX_BASIC_STATS|STATX_MNT_ID, stx_
attributes=0, stx_mode=S_IFREG|0644, stx_size=63775, ...}) = 0
mmap2(NULL, 63775, PROT_READ, MAP_PRIVATE, 3, 0) = 0xf7f63000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (Нет такого файла или каталога)
openat(AT_FDCWD, "/lib32/libc.so.6", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0`\27\2\0004\0\0\0"..., 512) = 512
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\23\25 [2>o\247\313Y\301F;\306wQ"..., 96, 468) = 96
statx(3, "", AT_STATX_SYNC_AS_STAT|AT_NO_AUTOMOUNT|AT_EMPTY_PATH, STATX_BASIC_STATS, {stx_mask=STATX_BASIC_STATS|STATX_MNT_ID, stx_
attributes=0, stx_mode=S_IFREG|0755, stx_size=2258024, ...}) = 0
mmap2(NULL, 2295740, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xf7c00000
mprotect(0xf7c20000, 2113536, PROT_NONE) = 0
mmap2(0xf7c20000, 1564672, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x20000) = 0xf7c20000
mmap2(0xf7d9e000, 544768, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x19e000) = 0xf7d9e000
mmap2(0xf7e24000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x223000) = 0xf7e24000
mmap2(0xf7e27000, 38844, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xf7e27000
close(3)                                = 0
set_thread_area({entry_number=-1, base_addr=0xf7f74500, limit=0x0fffff, seg_32bit=1, contents=0, read_exec_only=0, limit_in_pages=1
, seg_not_present=0, useable=1}) = 0 (entry_number=12)
set_tid_address(0xf7f74568)             = 4396
set_robust_list(0xf7f74570, 12)         = 0
rseq(0xf7f74a20, 0x20, 0, 0x53053053)   = 0
mprotect(0xf7e24000, 8192, PROT_READ)   = 0
mprotect(0x565a7000, 4096, PROT_READ)   = 0
mprotect(0xf7fb0000, 8192, PROT_READ)   = 0
ugetrlimit(RLIMIT_STACK, {rlim_cur=8192*1024, rlim_max=RLIM_INFINITY}) = 0
```

Рисунок 13 – Просмотр событий при создании нового процесса с помощью **strace**

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**/**edb-debugger**).
2. Компилятор с языка **C** (**Visual Studio**, **MinGW**, **gcc**).
3. **Process Monitor**/**strace**.
4. **Process Hacker**/**Process Explorer**/**ps aux**.

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

2.2 Требования к отчету и критерии оценивания

Стоит отметить, что выполнение пункта 2.5 («Индивидуальное задание») позволяет получить максимально **только 6 баллов из 10**. Для оценки **7 и более** за практическую работу нужно выполнить пункт 2.5 («Индивидуальное задание») и **защитить ПР**.

Примечание. На защите возможны вопросы не только по ходу работы, но и по соответствующему лекционному материалу или прочитанной литературе по соответствующей теме.

Критерии оценки работ:

1. **Правильно приведены все требуемые этапы работы по заданию из п. 2.5** (4 из 10 баллов за этот пункт) в соответствии с данной инструкцией к выполнению ПР в логически структурированном отчёте.
2. **Отчет соответствует ГОСТ 7.32** (2 из 10 баллов за этот пункт). В нем есть обязательные разделы:
    - титульный лист;
    - введение;
    - основная часть;
    - заключение.

Все рисунки в виде скриншотов пронумерованы и подписаны. Каждый скриншот является информативным.

3. **Необязательно, по желанию студента**: возможна защита работы для получения дополнительных 4 из 10 баллов за ПР.

Итого: при **выполнении всех трех пунктов возможно** получение **10 баллов** за ПР.

2.3 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/Linux.
4. Цикл выполнения одной из 5 команд о тех пор, пока не введен «end».

2.4 Ход работы

Примечание. Не обязательно приводить все скриншоты, как в данном разделе. Обязательно привести скриншоты, которые выделены **полужирным** шрифтом в соответствии с требованиями пункта 2.5.

2.4.1 Работа в ОС Windows

Для получения исполняемого образа нужно скомпилировать файл «file.c», например, с использованием установленного компилятора **MinGW** командой (Рисунок 14):

```
gcc –m32 file.c -o file.exe
```

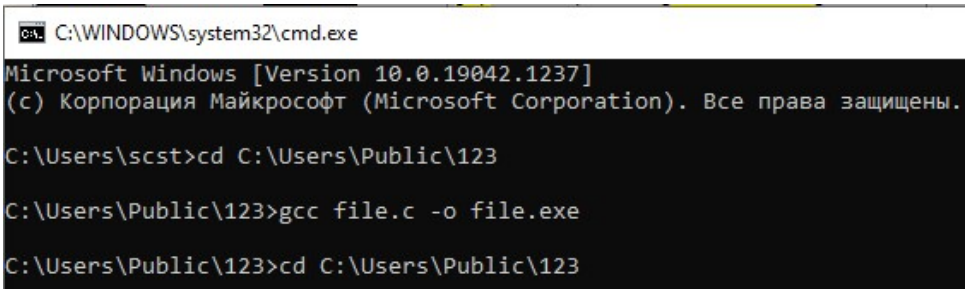

Рисунок 14 – Компиляция и сборка исполняемого файла file.exe

Полученный скомпилированный PE-файл «file.exe» загрузим в отладчик **x32dbg**. Далее убедимся, что в параметрах отладчика (окно «Параметры», Рисунок 15) 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.

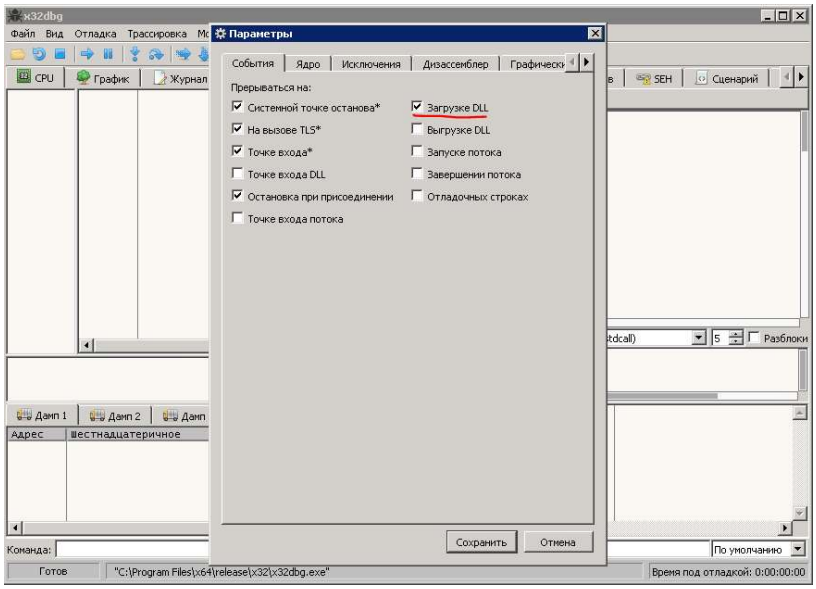

Рисунок 15 – Параметры событий останова отладчика

Нажмем F9 до тех пор, пока отладчик не остановится в EntryPoint (Рисунок 16).

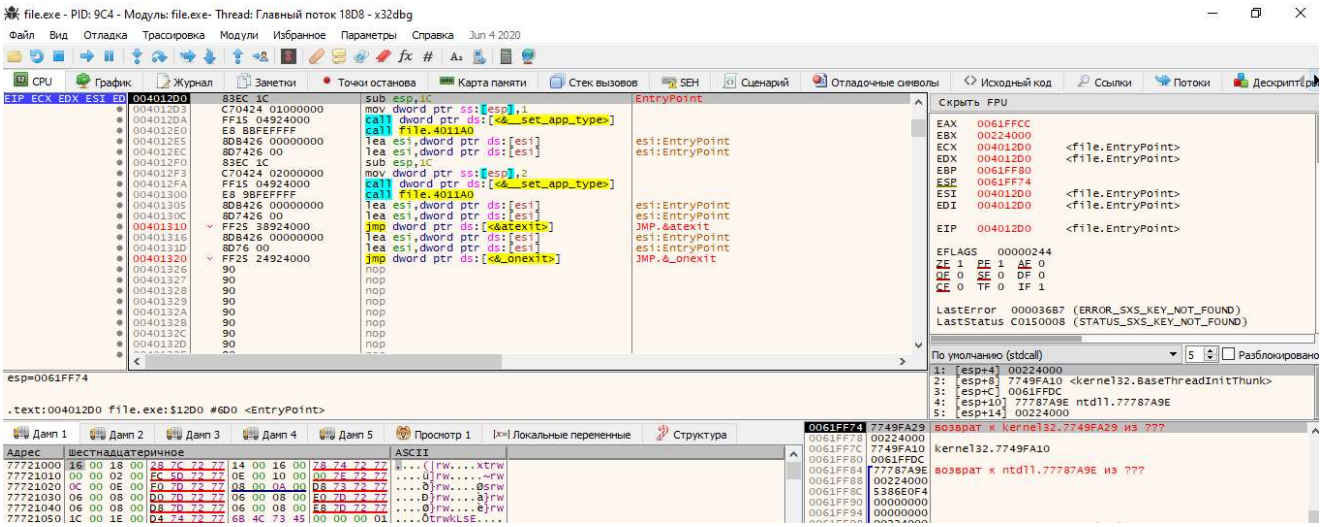

Рисунок 16 – EntryPoint программы

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 17). В **нашем случае этот *call* имеет адрес 40122E** и переходит на адрес 4019DF. Здесь можно установить программную точку останова (F2).

Примечание. Адреса функций в примере и при выполнении ПР могут отличаться, т.к. они зависят от компилятора и опций компиляции и сборки.

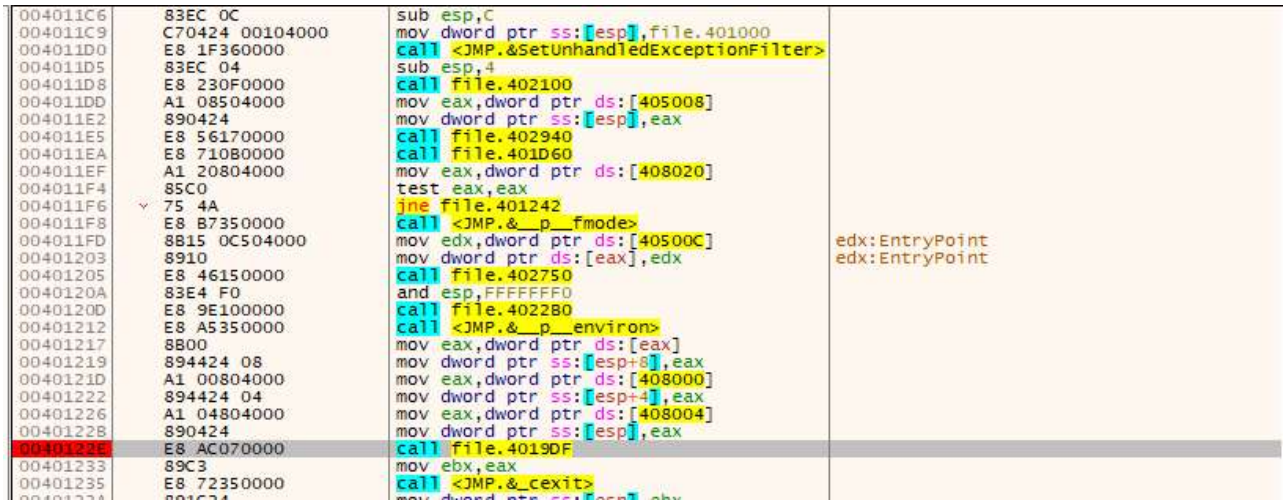

Рисунок 17 – Поиск функции *main* в исполняемом файле

Если перейти в функцию по адресу 4019DF, то можно увидеть такую картину (Рисунок 18).

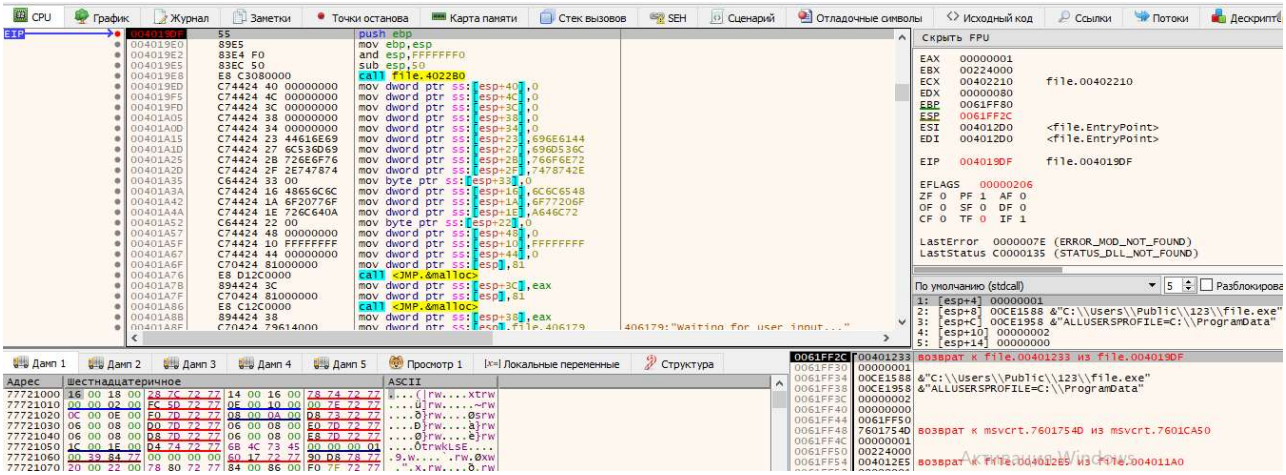

Рисунок 18 – Функция *main*

Дойдем до функций выделения памяти с помощью *malloc* (**Рисунок 19**) в *main*. Сразу за соответствующими *call malloc* поставим программные точки останова с целью изучения адресов, которые выдаст ОС Windows после вызова *malloc* (адрес возвращается функцией через *EAX*, если нажать на него ПКМ и «Перейти к дампу» –> Дамп 2).

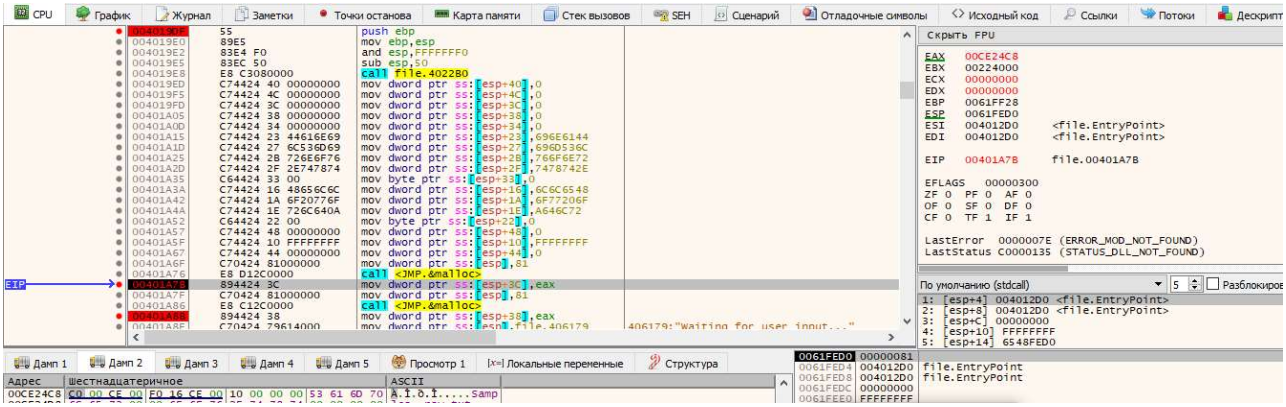

Рисунок 19 – Адреса после вызовов *malloc*

В функции *main* вызывается *malloc* дважды (**Рисунок 20**): для выделения памяти под переменную *buf* (куда попадают вводимые данные с консоли) и переменную *command* (первое слово из *buf*).

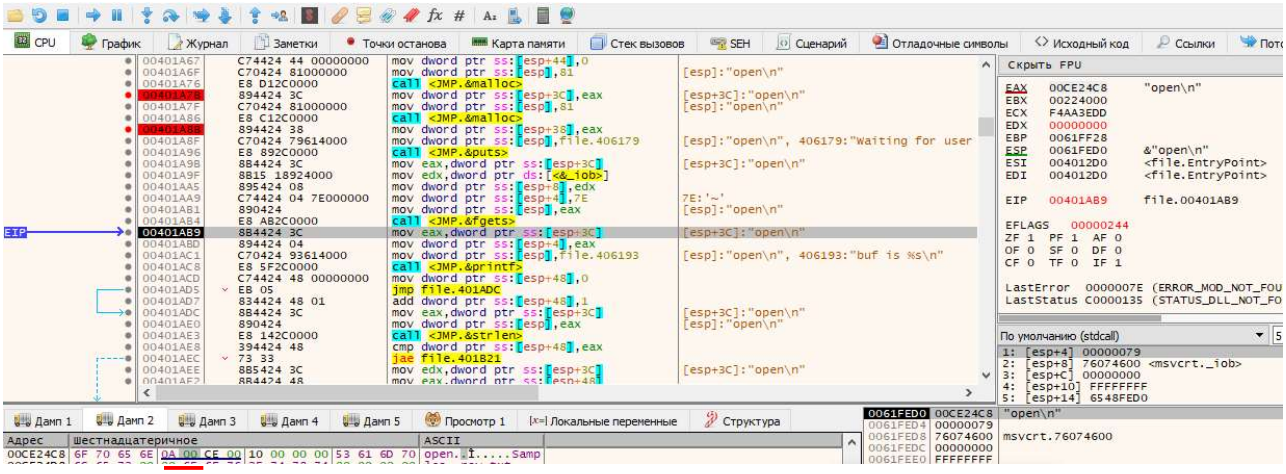

Рисунок 20 – Содержимое памяти по указателю *buf* после *fgets*

Стоит отметить, что после вызова *fgets* функция добавит нуль-терминатор в строку автоматически после спецсимвола «\n» (**Рисунок 20**). 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 21). Тогда его надо записать отдельным действием, либо учесть при копировании.

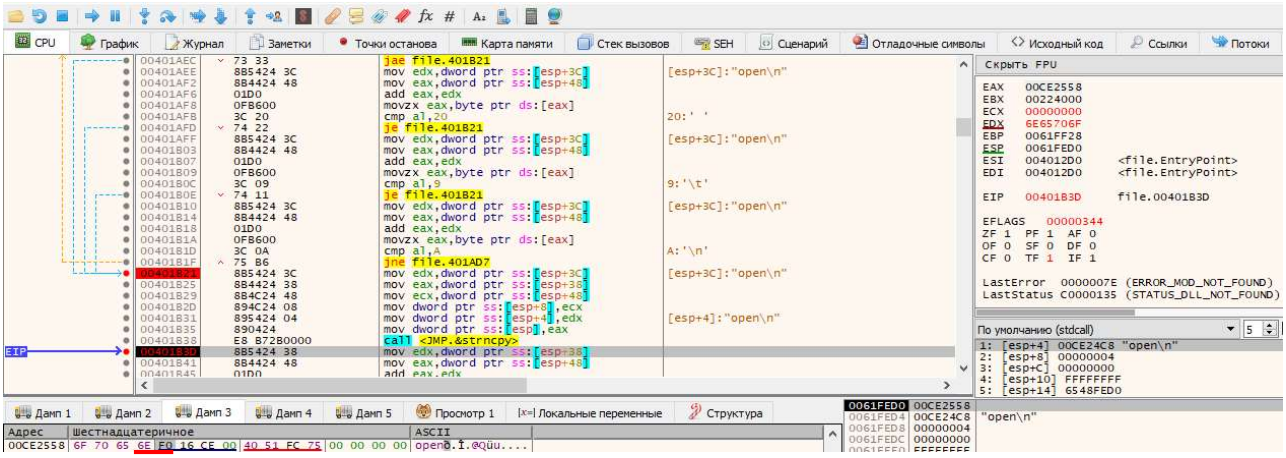

Рисунок 21 – Содержимое памяти по указателю *command* после копирования из *buf*

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 22**).

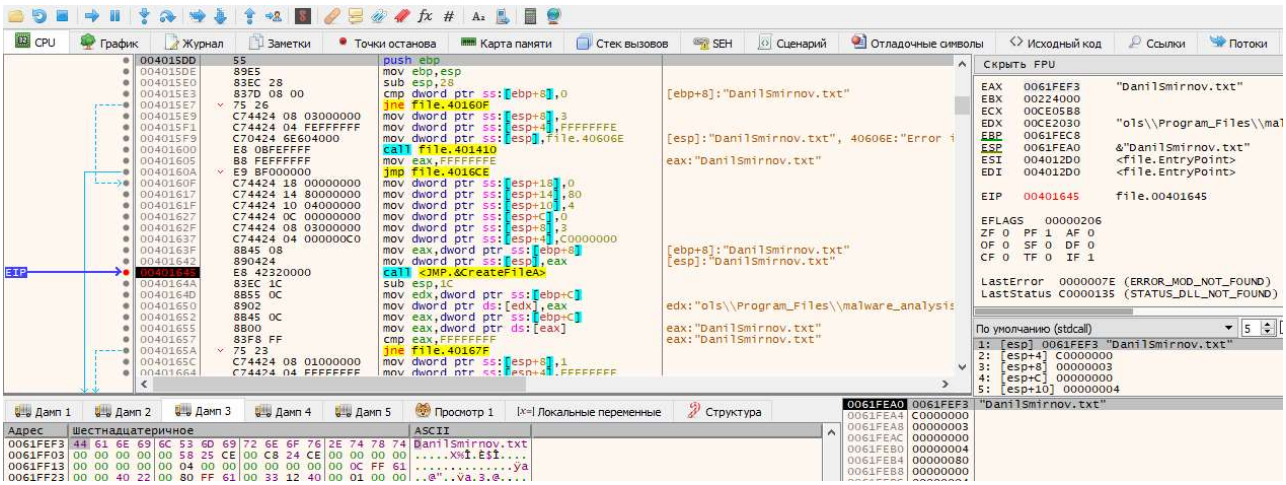

Рисунок 22 – Состояние стека и регистра EAX перед вызовом *CreateFileA*

После вызова API-функции *CreateFileA* можно увидеть, что в *EAX* записано значение *000000F4* (согласно спецификации с MSDN [12], это и есть хендл открытого файла, **Рисунок 23**).

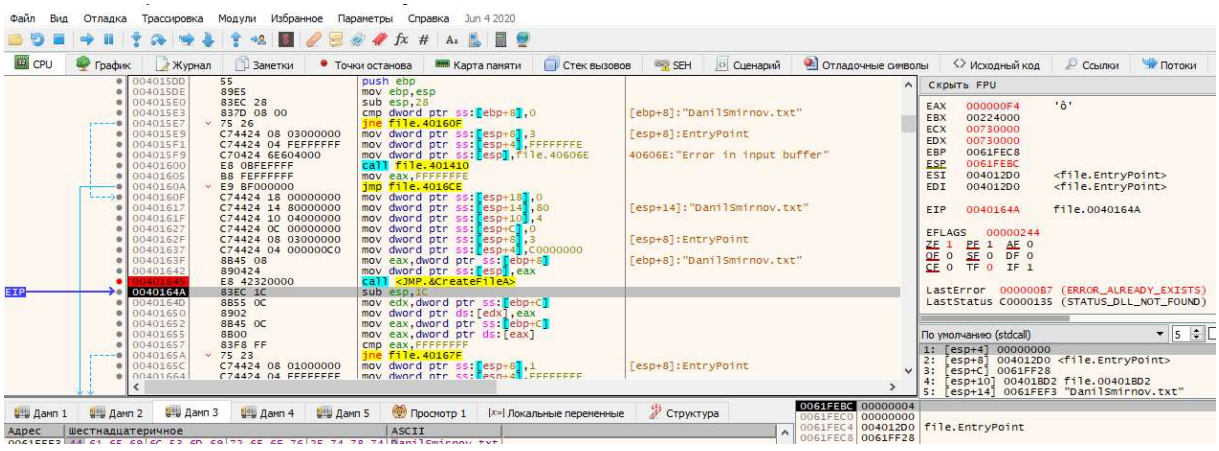

Рисунок 23 – Состояние стека и регистра EAX после вызова *CreateFileA*

То же значение хендла можно увидеть и в программе Process Hacker [16] (или Process Explorer из набора Sysinternals), если выбрать процесс-потомок от отладчика x32dbg.exe и выбрать его свойства и вкладку «Handles» (**Рисунок 24**).

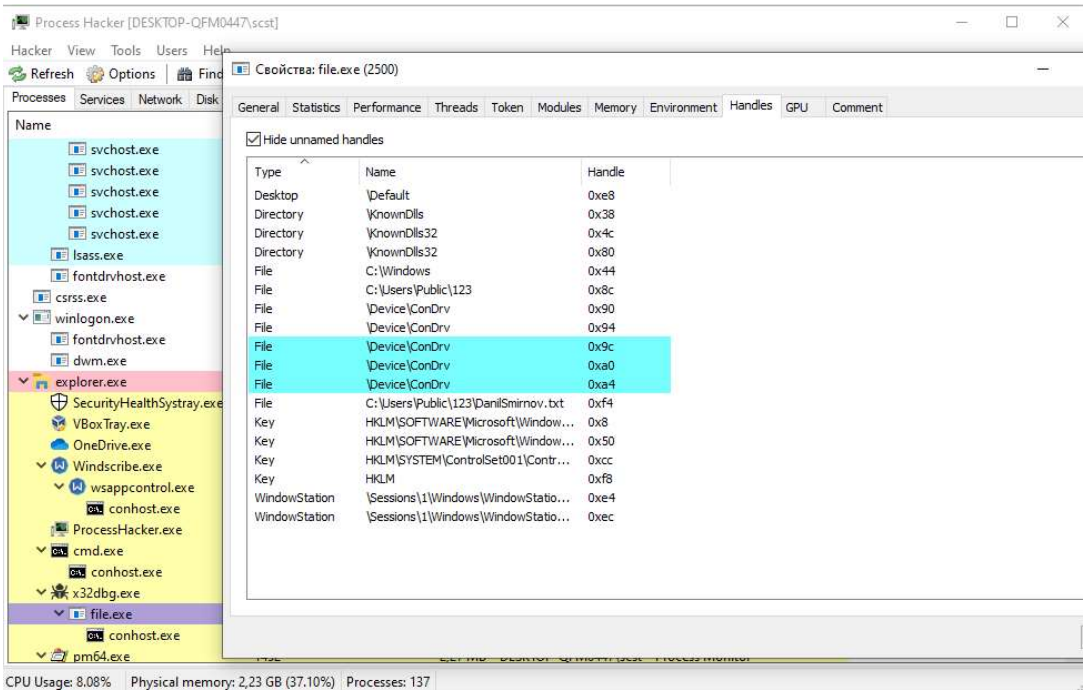

Рисунок 24 – Хендлы исследуемого процесса в Process Hacker

Для мониторинга всех открываемых файлов конкретной программой можно использовать уже известную утилиту **Process Monitor** (**Рисунок 25**).

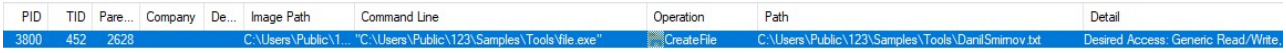

Рисунок 25 – Открытие файла в Process Monitor

Аналогично по отладчику можно отследить и операции чтения/записи данных из файла/в файл. Скриншоты по операциям чтения и записи рекомендуется делать и до вызова API-функций (**Рисунок 26**) и после (**Рисунок 27**), чтобы можно было понять адрес буфера для чтения из файла (в данном случае память чуть выше выделялась с помощью функции *malloc()* в куче) и отследить, что в него считалось.

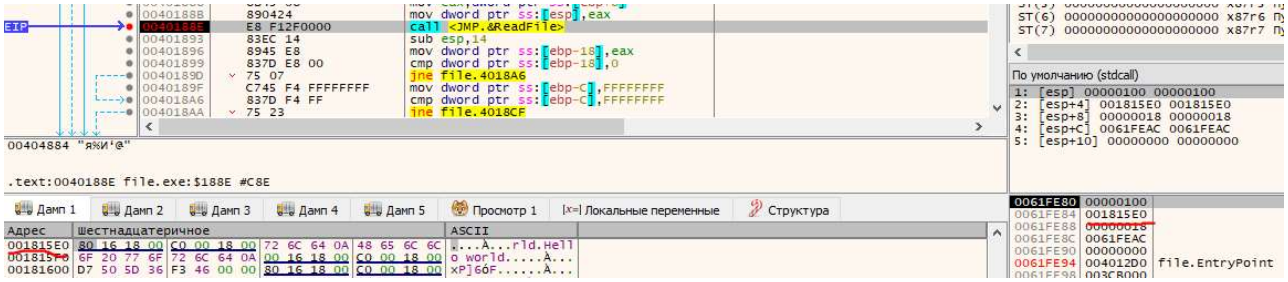

Рисунок 26 – Состояние стека перед вызовом ReadFile

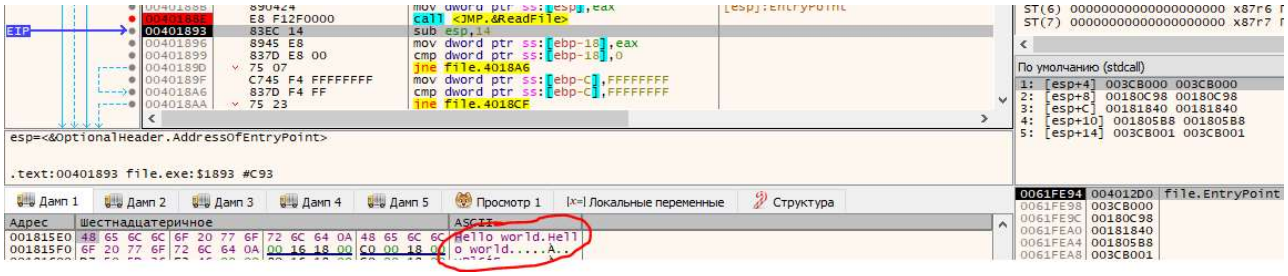

Рисунок 27 – Состояние кучи после вызова ReadFile, куда записался буфер из файла

Также важно зафиксировать параметры чтения из файла и с помощью Process Monitor (**Рисунок 28**).

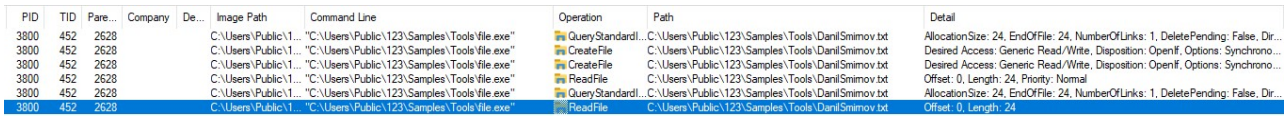

Рисунок 28 – Операция чтения из файла в Process Monitor

Просьба обратить внимание, что на примерах выше приведено только чтение файлов, то есть скриншоты по **записи данных** в файл нужно сделать **самостоятельно** по аналогии.

2.4.2 Работа в ОС Linux

Работа в ОС Linux с программой аналогична работе в ОС Windows. Сначала необходимо скомпилировать программу командой:

```
gcc -m32 file.c -o file.out
```

Затем запустить отладку с использованием **edb-debugger**, например, командой:

```
edb --run file.out
```

Далее нужно проверить предпочтения по первому останову программы. Опция «Initial Breakpoint» должна соответствовать символу *main* (Рисунок 29). Если не соответствует, то нужно изменить параметр.

Рисунок 29 – Проверка предпочтений по первому останову программы

После нажатия F9 попадаем в функцию *main* (Рисунок 30).

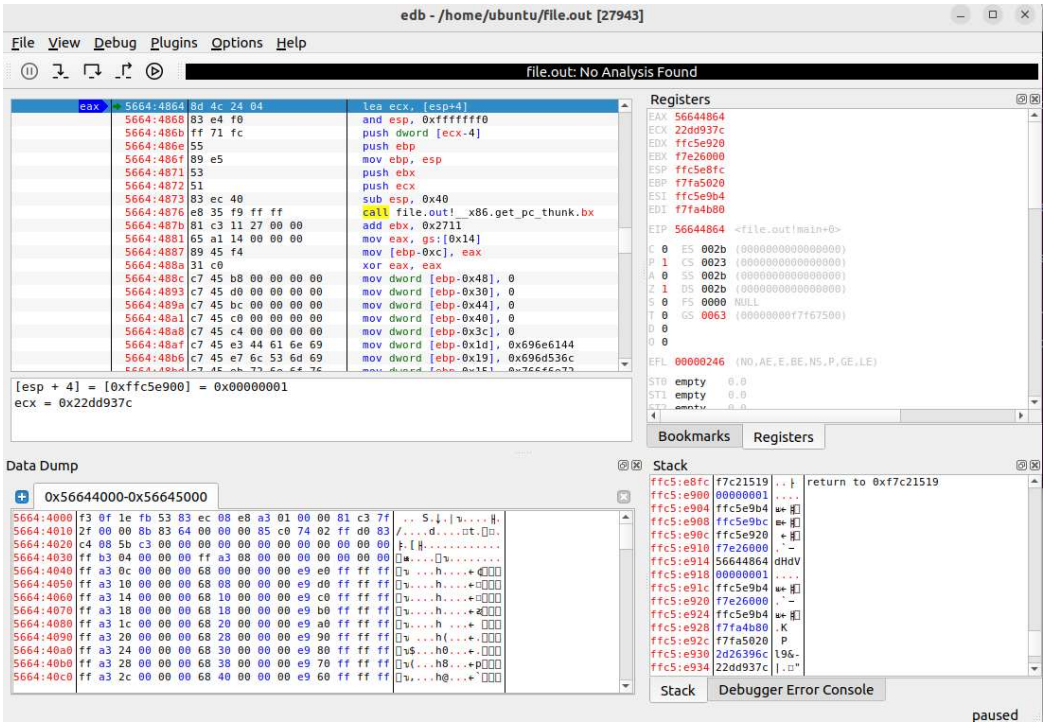

Рисунок 30 – Функция *main*

Дойдем до функций выделения памяти с помощью *malloc* в *main*. Сразу за соответствующими *call malloc* поставим программные точки останова с целью изучения адресов, которые выдаст ОС Windows после вызова *malloc* (адрес возвращается функцией через *EAX*, если нажать на него ПКМ и «Follow in Dump», **Рисунок 31**).

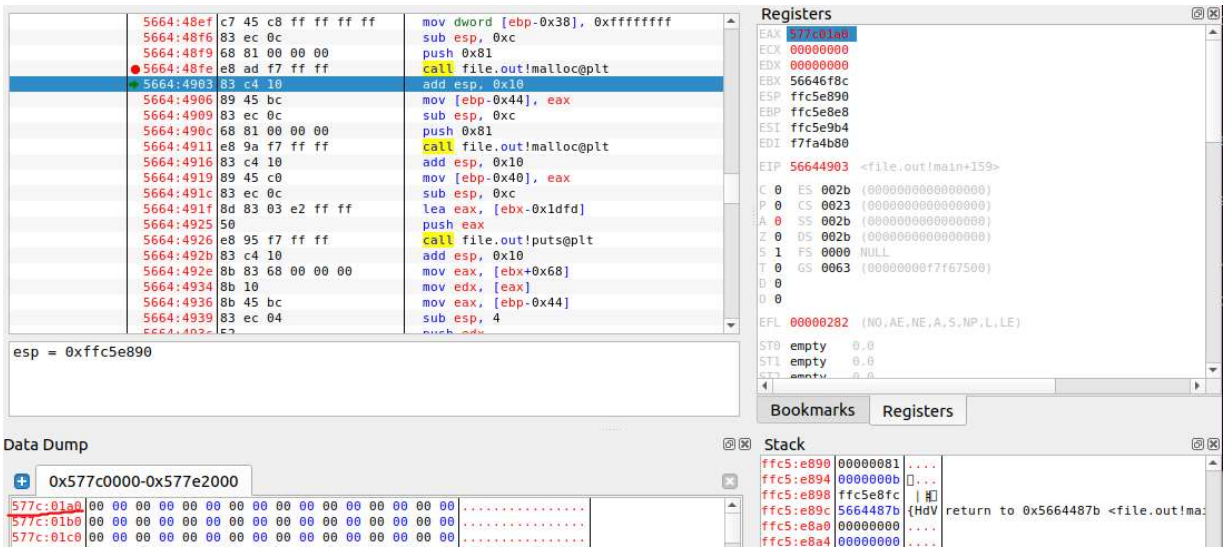

Рисунок 31 – Адреса после вызовов *malloc*

В функции *main* вызывается *malloc* дважды (**Рисунок 32**): для выделения памяти под переменную *buf* (куда попадают вводимые данные с консоли) и переменную *command* (первое слово из *buf*).

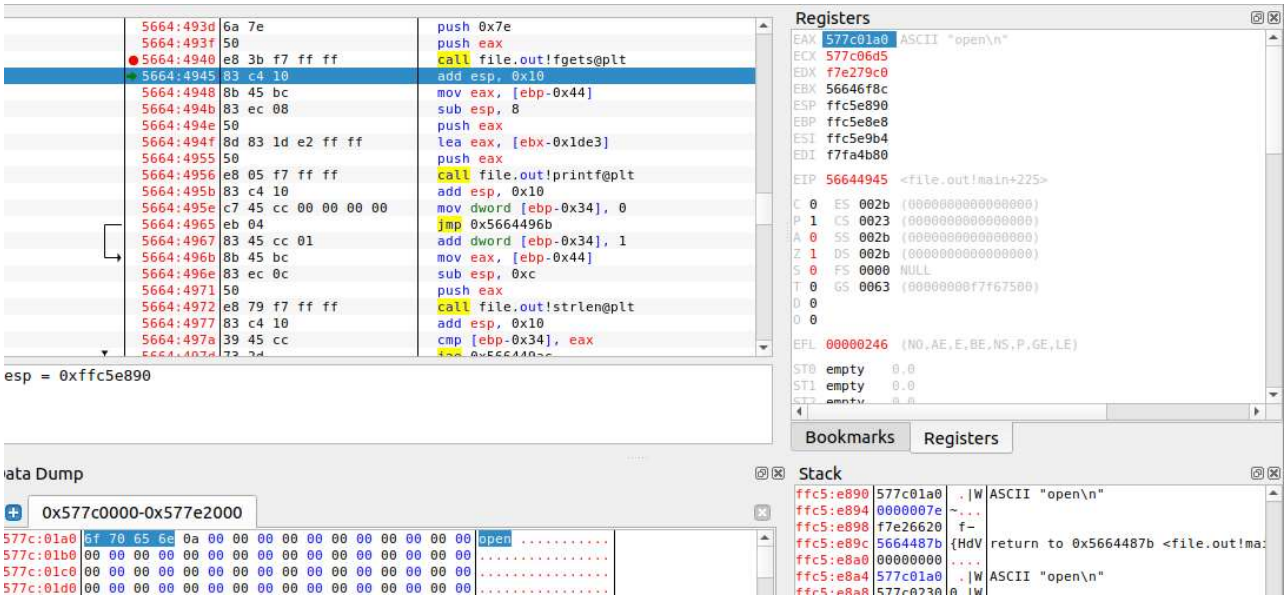

Рисунок 32 – Содержимое памяти по указателю *buf* после *fgets*

Если в консоль записать значение «open», то после проверки условий будет вызвана функция с символом «LabOpenFile». В ней будет в самом начале произведена проверка валидности строки с именем файла. Затем производится вызов *open* (**Рисунок 33**).

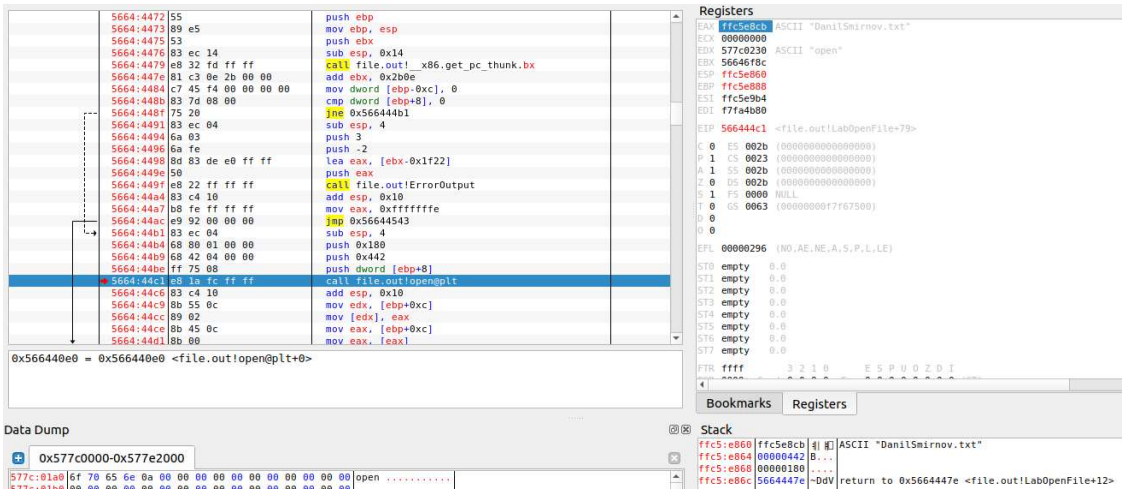

Рисунок 33 – Состояние стека и регистра EAX перед вызовом *open*

После вызова *open* можно увидеть, что в *EAX* записано значение *00000003* (согласно команде «`man 2 open`» это и есть дескриптор открытого файла, **Рисунок 34**).

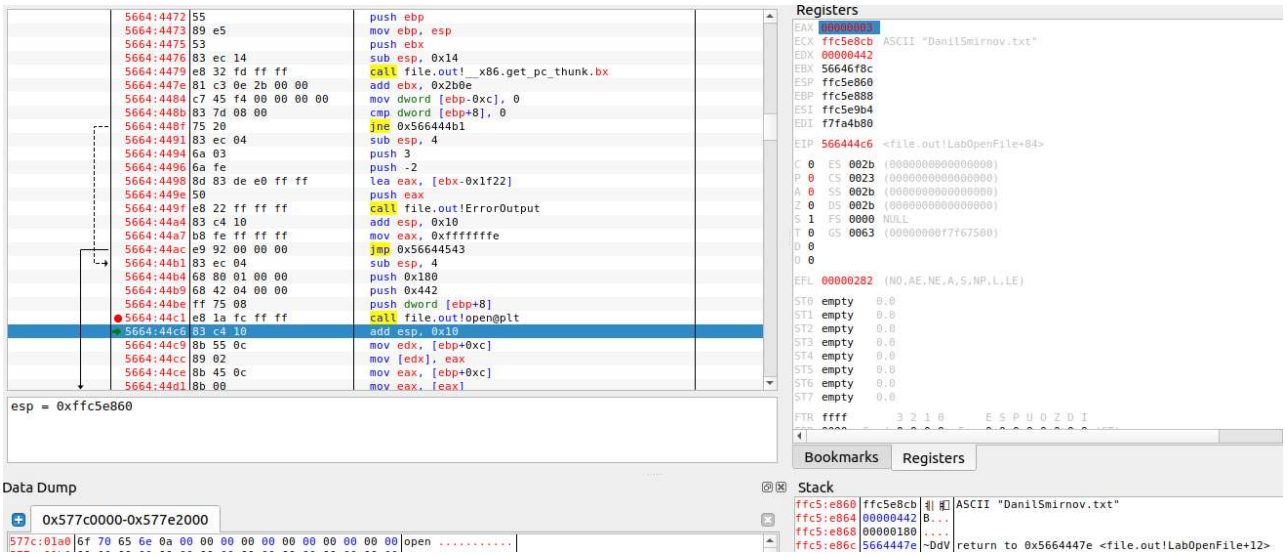

Рисунок 34 – Состояние стека и регистра EAX после вызова *open*

То же значение дескриптора можно увидеть и путем вывода содержимого каталога *fd* для соответствующего процесса по его *pid*. Для этого нужно сначала получить список всех процессов, например, следующей командой:

```
ps aux | grep -i "file.out"
```

Далее нужно выяснить *pid* интересующего процесса и выполнить команду:

```
ls -la /proc/<PID>/fd/
```

В конкретном примере *pid* имеет значение 27943, поэтому обращение в псевдофайловую систему */proc* будет к этому *pid* (**Рисунок 35**). Видно, что дескриптор из отладчика и из */proc* совпали.

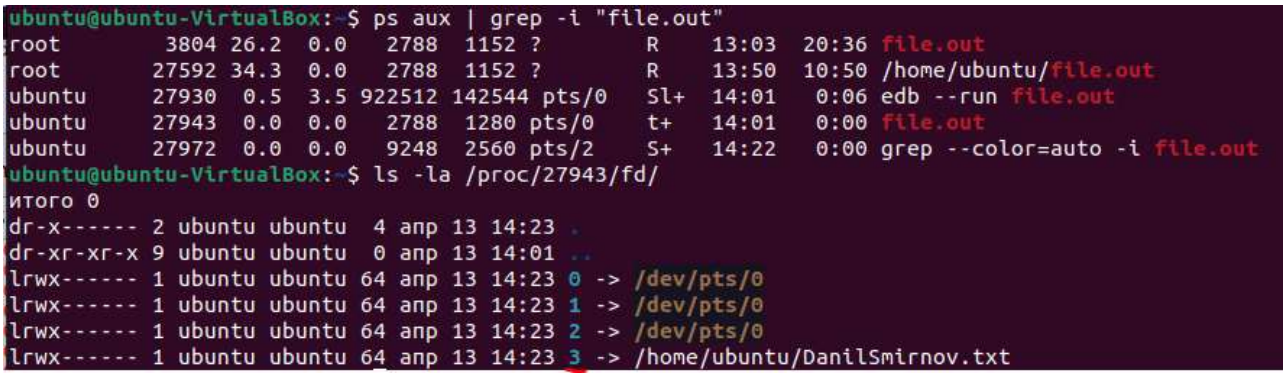

Рисунок 35 – Вывод всех дескрипторов для соответствующего процесса

Аналогично по отладчику можно отследить и операции чтения/записи данных из файла/в файл. Скриншоты по операциям чтения и записи рекомендуется делать и до системных вызовов (**Рисунок 36**) и после (**Рисунок 37**), чтобы можно было понять адрес буфера для чтения из файла (в данном случае память чуть выше выделялась с помощью функции *malloc* в куче) и отследить, что в него считалось.

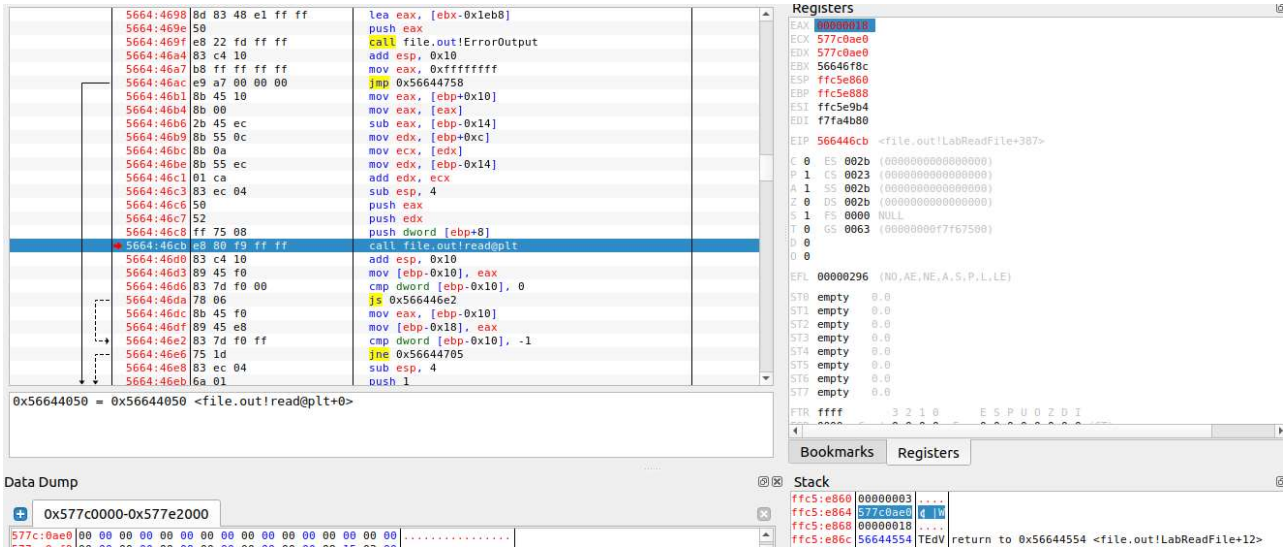

Рисунок 36 – Состояние стека перед вызовом *read*

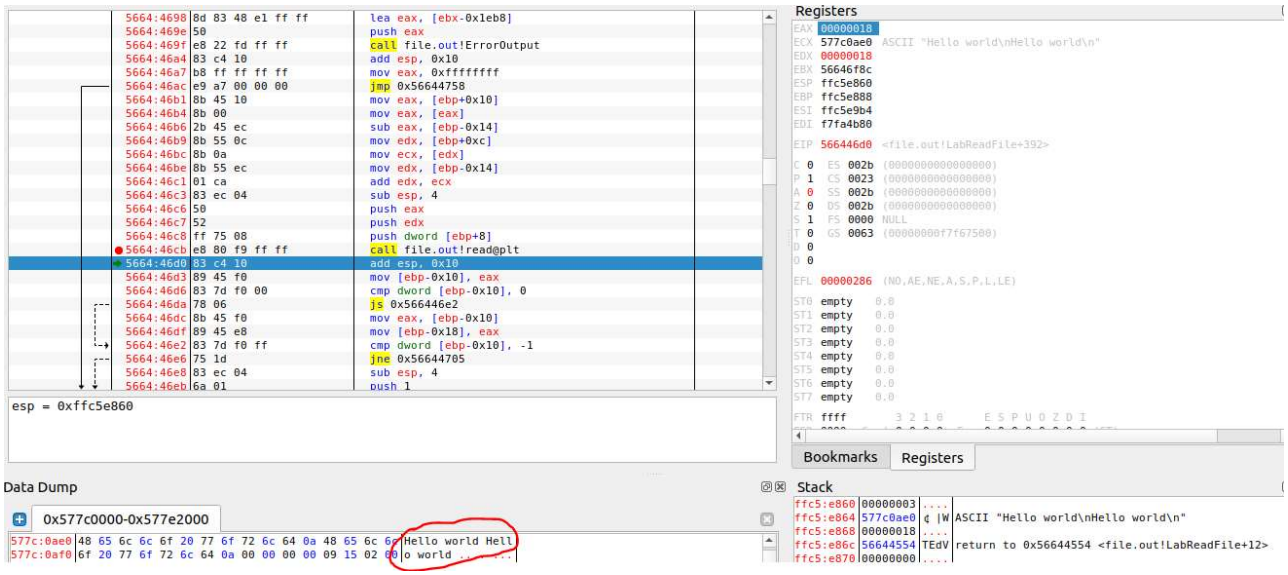

Рисунок 37 – Состояние кучи после вызова *read*, куда записался буфер из файла

Просьба обратить внимание, что на примерах выше приведено только чтение файлов, то есть скриншоты по **записи данных** в файл нужно сделать **самостоятельно** по аналогии.

Утилита **strace** позволяет не только отслеживать системные вызовы создаваемых дочерних процессов, но и отлаживать существующие процессы в системе. Из-за особенности работы утилиты **strace** через перехват системных вызовов в моменте отладки другим отладчиком (в данном случае **edb-debugger**) одновременно отлаживать в **edb-debugger** и отслеживать системные вызовы утилитой **strace** не получится. Поэтому на этом работа с отладчиком **edb-debugger** завершается.

Для дальнейшего мониторинга работы программы студенту предлагается запустить программу через консоль:

```
./file.out
```

В другом консольном окне нужно выяснить *pid* интересующего процесса и присоединиться к нему утилитой **strace** от имени привилегированного пользователя командой:

```
sudo strace -e trace=read,write,memory -p <PID>
```

На примере выше приведен мониторинг операций, использующих системные вызовы чтения, записи и обращений к памяти. В консоли, где запущена программа *file.out*, выполним последовательность команд *open*, *write*, *read* и *end*. Вывод **strace** приведен на рисунке (**Рисунок 38**).

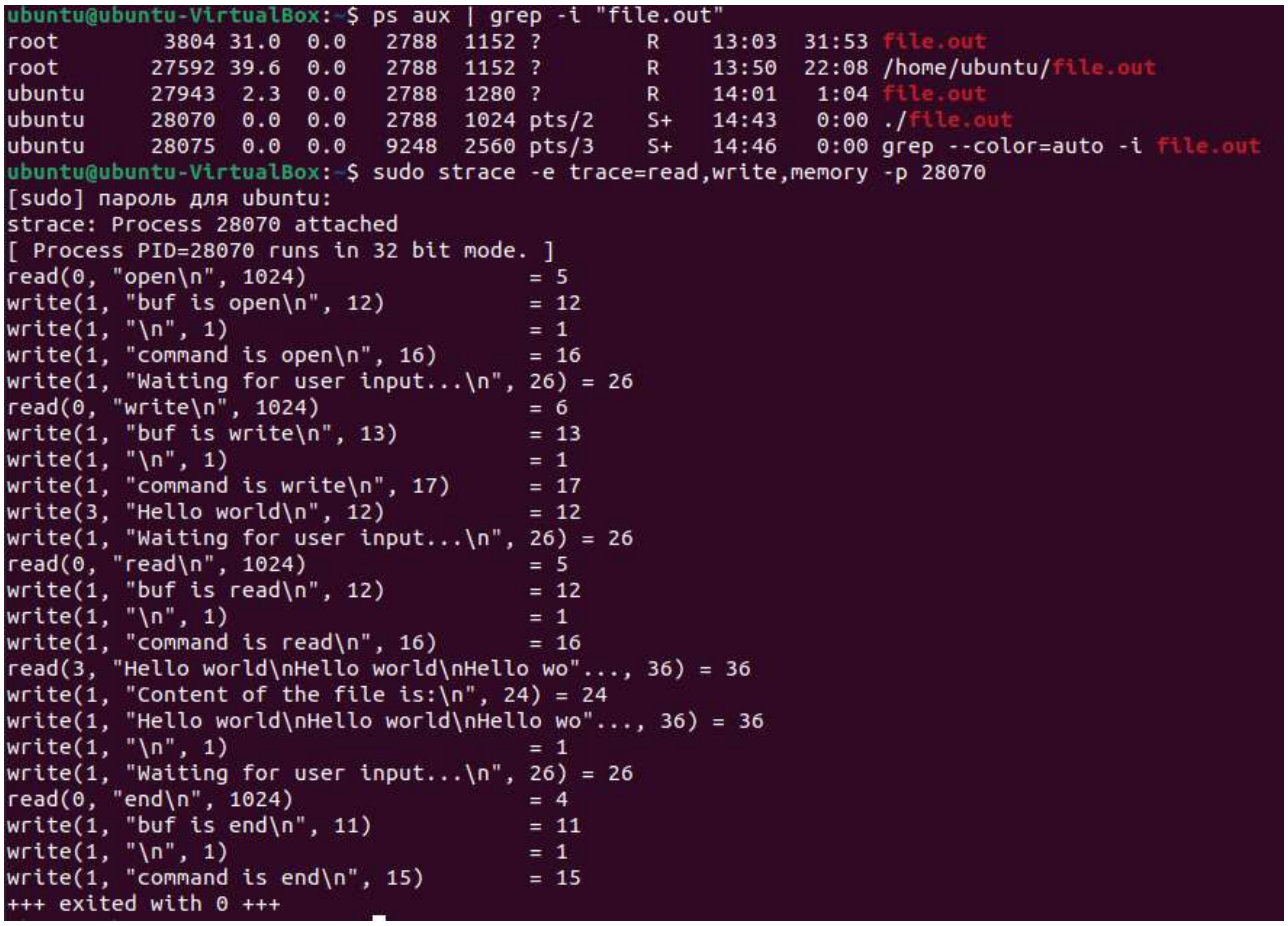

Рисунок 38 – Пример вывода strace для последовательности команд

2.5 Индивидуальное задание

Каждому студенту предоставляется возможность выбора операционной системы (ОС Windows или ОС Linux), на которой будет выполняться практическая работа. Оценка не зависит от выбора ОС.

2.5.1 Задание по ОС Windows

Индивидуальное задание:

1. Получить файлы «file.c» и «file.h».
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.
6. Написать отчет, в котором отразить основные этапы со скриншотами:
    a. **Выделение памяти** с помощью *malloc* (после вызова *malloc*, как на **Рисунок 19**) 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 скриншоте (как на **Рисунок 20**).
    c. **Вызов API-функции** *CreateFileA* после написания команды «*open*» в консоль, где указаны все аргументы функции в стеке – в частности, выделить на скриншоте имя файла, которое должно быть индивидуальным. Сделать скриншоты до (**Рисунок 22**) и после вызова *CreateFileA* (**Рисунок 23**). На скриншоте 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 24**). Также сравнить имя открываемого файла в отладчике со значением в Process Monitor (**Рисунок 25**).
    d. **Вызов API-функции** *WriteFile* после написания команды «*write*» в консоль, где указаны все аргументы функции в стеке. До (**Рисунок 26**) и после (**Рисунок 27**) 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 в Process Monitor (**Рисунок 28**). Обратите внимание, что отдельного примера скриншота для записи в п. 2.4. **не было приведено**.

e. **Вызов API-функции** *ReadFile* после написания команды «*read*» в консоль, где указаны все аргументы функции в стеке. До (**Рисунок 26**) и после (**Рисунок 27**) 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 в Process Monitor (**Рисунок 28**).

2.5.2 Задание по ОС Linux

Индивидуальное задание:

7. Получить файлы «file.c» и «file.h».
8. Скачать и настроить необходимые инструменты в соответствии с инструкцией из раздела 2.1, затем запустить их.
9. **Изменить** значение в переменной *chFileName* на свои «**ИмяФамилия.txt**» без пробелов на английском языке.
10. Перекомпилировать файл.
11. Запустить файл в отладчике.
12. Написать отчет, в котором отразить основные этапы со скриншотами:

    a. **Выделение памяти** с помощью *malloc* (после вызова *malloc*, как на **Рисунок 31**) 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 скриншоте (как на **Рисунок 32**).

    c. **Системный вызов** *open* после написания команды «*open*» в консоль, где указаны все аргументы функции в стеке – в частности, выделить на

скриншоте имя файла, которое должно быть индивидуальным. Сделать скриншоты до (**Рисунок 33**) и после вызова *open* (**Рисунок 34**). На скриншоте после вызова показать возвращаемое значение в регистре *EAX* и интерпретировать, что оно означает (дескриптор, количество байт, область памяти и т.д.). Сравнить со значением команды `ls -la /proc/<PID>/fd/` (как на **Рисунок 35**).

d. **Системный вызов** *write* после написания команды «*write*» в консоль, где указаны все аргументы функции в стеке. До (**Рисунок 36**) и после (**Рисунок 37**) 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 (дескриптор, количество байт, область памяти и т.д.) и привести на скриншоте. Обратите внимание, что примера скриншота в п.2.4. **не было приведено**.

e. **Системный вызов** *read* после написания команды «*read*» в консоль, где указаны все аргументы функции в стеке. До (**Рисунок 36**) и после (**Рисунок 37**) вызова показать *в дампе то место, какие данные были считаны из файла* и *показать возвращаемое значение в регистре EAX* и интерпретировать, что оно означает (дескриптор, количество байт, область памяти и т.д.) и привести на скриншоте.

f. **Выполнить** отдельный запуск программы. Затем подключиться к ней с использованием утилиты strace и выполнить последовательность операций *open*, *write*, *read*, *end* (**Рисунок 38**).

2.6 Дополнительное задание

Для получения дополнительных баллов в практической части модуля (с весом **0,1 от всех практических работ данного модуля**, то есть к примеру, вместо ПР 4.4) можно выполнить дополнительное задание (**при условии защиты ПР 4.1!**). Доп. задание состоит из двух обязательных частей:

1. Найти **утечку ресурсов** в программе, вывести проблемное место на скриншоте и предложить несколько путей решения проблемы.

2. **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/дескриптор** и записываемое значение, а для закрытия файла – **хендл/дескриптор**).

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. Таненбаум, Э. Архитектура компьютера. 6-е издание / Э. Таненбаум, Т. Остин – 2018.
3. Керниган, Б. Язык программирования Си. / Б. Керниган, Д. Ритчи. – 2017.
4. Сикорски, М. Вскрытие покажет! Практический анализ вредоносного ПО. / М. Сикорски, Х. Эндрю. – 2018.
5. Монаппа, К. Анализ вредоносных программ. / К. Монаппа. – 2019.
6. Hale Ligh, M. The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory. / A. Case, J. Levy, A. Walters
7. Пошаговое руководство к исполняемым файлам (EXE) Windows [Электронный ресурс]. URL: https://habr.com/ru/articles/148194/ (дата обращения 08.03.2024)
8. pestudio [Электронный ресурс]. URL: https://www.winitor.com/download/ (дата обращения 08.03.2024)
9. Explorer Suite [Электронный ресурс]. URL: https://ntcore.com/explorer-suite/ (дата обращения 08.03.2024)
10. x64dbg. An open-source x64/x32 debugger for windows. [Электронный ресурс]. URL: https://x64dbg.com/#start (дата обращения 08.03.2024)
11. Пролог и эпилог функции [Электронный ресурс]. URL: http://datadump.ru/function-prologue-epilogue/(дата обращения 08.03.2024)
12. CreateFileA function (fileapi.h) [Электронный ресурс]. URL: https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea (дата обращения 08.10.2023)
13. Process Monitor v3.85 [Электронный ресурс]. URL: https://docs.microsoft.com/en-us/sysinternals/downloads/procmon (дата обращения 08.10.2023)

14. PsSetCreateProcessNotifyRoutine function (ntddk.h) [Электронный ресурс]. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-pssetcreateprocessnotifyroutine (дата обращения 08.10.2023)
15. PsSetLoadImageNotifyRoutine function (ntddk.h) [Электронный ресурс]. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-pssetloadimagenotifyroutine (дата обращения 08.10.2023)
16. Process Hacker 2.39 r124 [Электронный ресурс]. URL: https://processhacker.sourceforge.io/downloads.php (дата обращения 08.10.2023)