Санкт-Петербургский Государственный Политехнический Университет

Институт информационных технологий и управления

Кафедра Компьютерных Систем и Программных Технологий

Отчет по лабораторной работе № 4

По дисциплине «Проектирование ОС и компонентов»

По теме «Управление памятью»

**Работу выполнила студентка группы № 13541/3**

Шаляпин Н.С. \_\_\_\_\_\_\_\_\_\_\_

**Работу принял преподаватель:**

Душутина Е.В. \_\_\_\_\_\_\_\_\_\_\_

.

Санкт-Петербург

2017

## Программа работы

Управление памятью. Создание библиотек разных типов и их размещение. Загрузка программ и библиотек, анализ дампа. Необходимо провести:

1. Провести анализ загрузки приложения в ОЗУ
2. Профилирование функционирования приложения

Используются в качестве приложений: утилита, выводящая информацию о файле для Windows и утилита для Linux, выводящая информацию о файле. Утилиты были разработаны в прошлом семестре.

## Используемые системы

Лабораторная работа выполнялась в ОС: Windows 7 и Ubuntu 14.04.

Windows

|  |
| --- |
| Имя узла: ADMIN-ПК  Название ОС: Microsoft Windows 7 Максимальная  Версия ОС: 6.1.7601 Service Pack 1 сборка 7601  Изготовитель ОС: Microsoft Corporation  Параметры ОС: Изолированная рабочая станция  Сборка ОС: Multiprocessor Free  Зарегистрированный владелец: Admin  Зарегистрированная организация:  Код продукта: 00426-OEM-8992662-00173  Дата установки: 02.04.2017, 14:53:54  Время загрузки системы: 27.05.2017, 21:26:27  Изготовитель системы: ASUSTeK Computer Inc.  Модель системы: K53SM  Тип системы: x64-based PC  Процессор(ы): Число процессоров - 1.  [01]: Intel64 Family 6 Model 42 Stepping 7 GenuineIntel ~2501 МГц  Версия BIOS: American Megatrends Inc. K53SM.214, 15.08.2012 |

Linux

|  |
| --- |
| nikita@nikita-K53SM:~$ uname -a  Linux nikita-K53SM 3.19.0-80-generic #88~14.04.1-Ubuntu SMP Fri Jan 13 14:54:07 UTC 2017 x86\_64 x86\_64 x86\_64 GNU/Linux |

1. **Анализ адресного пространства в Windows**

Для анализа будет использоваться утилита, выводящая информацию о файле, написанная в предыдущем семестре.

У каждого процесса есть свое виртуальное адресное пространство, в котором расположены все необходимые данные и исходный код программ.

Найдем точу входа программы. Для этого воспользуемся утилитой Dumpbin, которая поставляется с Visual Studio. Dumper двоичных файлов в формате COFF от Microsoft (DUMPBIN.EXE) отображает информацию о двоичных файлах в формате COFF.  Можно использовать DUMPBIN для исследования объектных файлов COFF, стандартных библиотек объектов COFF, исполняемых файлов и библиотек динамической компоновки (DLL).

Запустим утилиту из консоли командой **C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\bin\dumpbin.exe** , указав опцию **/headers** и нужный файл формата **.exe.**

Полный лог информации, выведенной утилитой приведен в файле Log\_dumpbin.txt.

|  |
| --- |
| C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\bin>dumpbin.exe /headers  "D:\!\13541\_3\OS\FullInfoFile\FullInfoFile2Old\Debug\FullInfoFile2.exe"  Microsoft (R) COFF/PE Dumper Version 12.00.21005.1  Copyright (C) Microsoft Corporation. All rights reserved.  Dump of file D:\!\13541\_3\OS\FullInfoFile\FullInfoFile2Old\Debug\FullInfoFile2.exe  PE signature found  File Type: EXECUTABLE IMAGE  FILE HEADER VALUES  14C machine (x86)  7 number of sections  59297ED5 time date stamp Sat May 27 16:27:49 2017  0 file pointer to symbol table  0 number of symbols  E0 size of optional header  102 characteristics  Executable  32 bit word machine  OPTIONAL HEADER VALUES  10B magic # (PE32)  12.00 linker version  11A00 size of code  8800 size of initialized data  0 size of uninitialized data  **111F9 entry point (004111F9) @ILT+500(\_wmainCRTStartup)**  1000 base of code  1000 base of data  **400000 image base (00400000 to 0042DFFF**)  1000 section alignment  … |

Где image base – предпочитаемый адрес в памяти, начиная с которого программа загружена в память (400000), entry point – точка входа программы. Первым значением указана относительная точка входа (111F9), вторым значением абсолютный адрес точки входа (004111F9). Заметим что

004111F9 = 111F9 + 400000, то есть entry point (абс) = entry point (отн) + image base.

Таким образом, был найден относительный адрес точки входа программы, что является смещением от адреса фактической загрузки программы.

Чтобы найти адрес фактической загрузки программы непосредственно в процессе ее выполнения необходимо в среде Visual Studio запустить программу и остановить с помощью breakpoint. Далее открыв окно Debug->Windows->Modules, просмотрим информацию об исполняемом файле программы и используемых библиотеках dll.

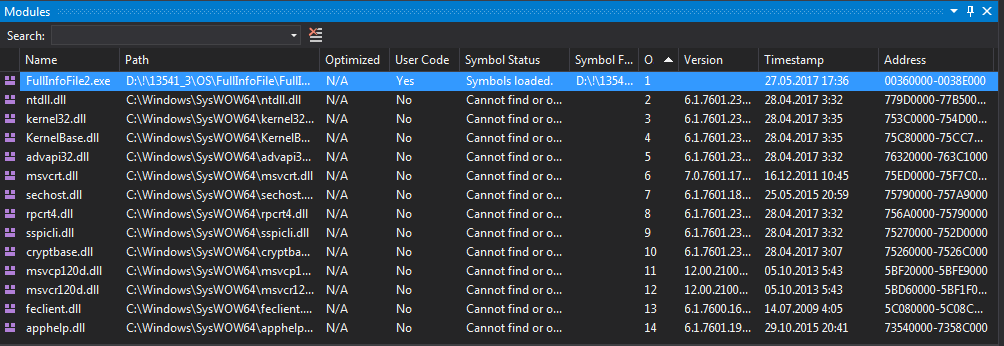


Рис. 1. Окно Modules

Исполняемый файл программы chmod.exe занимает следующий диапазон адресов 00360000-0038E000 .

Наидем реальную точку входа entry point (отн) + image base = entry point (абс).

111F9 + 00360000 = **003711F9.**

Адрес загрузки программы не является постоянным, однако размер диапазона при перезапуске не изменяется (рис. 2).

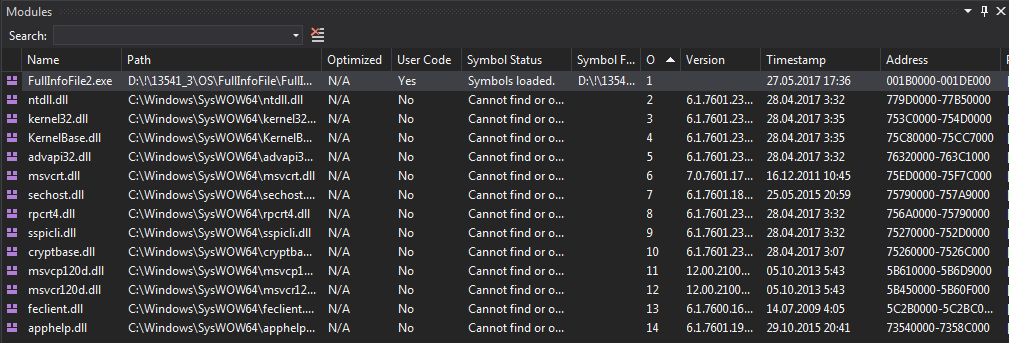


Рис. 2. Окно Modules после перекомпиляции

Найдем точу входа утилиты chmod, реализованной в виде сервиса.

Посмотреть состояние виртуальной памяти процесса можно с помощью отладчика. В данной работе использовалась среда разработки Visual Studio 2015 и встроенный отладчик этой среды.

Для просмотра адресного пространства процесса необходимо установить точку останова в любой достижимой строчке программы и запустить отладку. Далее через меню Debug->Windows->Memory можно открыть 4 окна просмотра памяти программы, каждое из которых работает независимо. Данные окна позволяют просматривать актуальное состояние памяти программы во время ее выполнения, редактировать какие-либо ячейки памяти и искать ячейки памяти, в которых расположены различные объекты программы (переменные, функции и т.д.) по их имени.

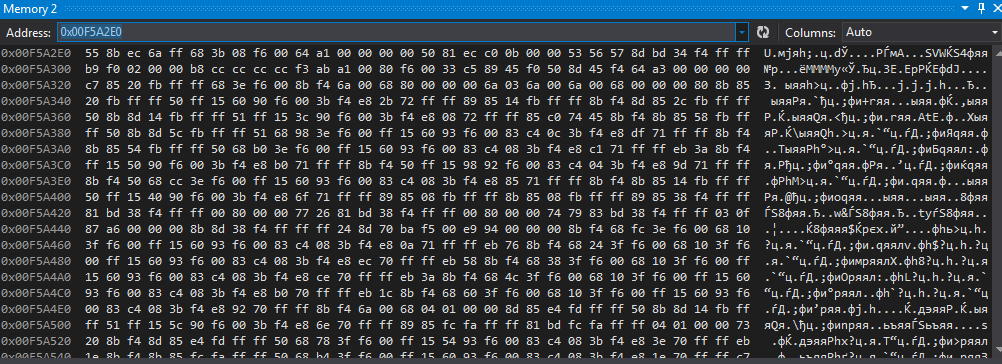


Рис. 3. Адрес функции main в памяти

На рис. 3 приведен пример поиска адреса функции main в памяти процесса. Как видно, функция расположена по адресу 0x00F5A2E0. При перекомпиляции проекта адрес функции main изменяется (рис. 4).

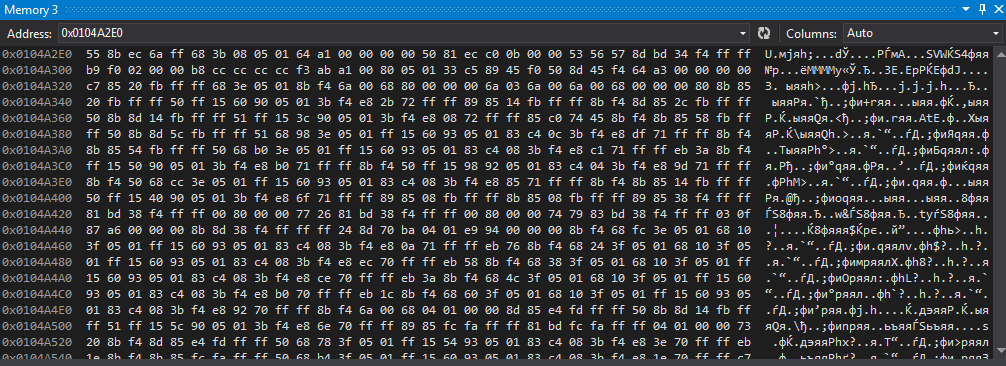


Рис. 4. Адрес функции main после перекомпиляции проекта

Так же, посмотреть расположение исходного кода программы в памяти можно с помощью инструмента дизассемблирования отладчика (Debug->Windows->Disassembly). Адрес функции main был найден в Memory 4 и Disassembly (рис. 5).

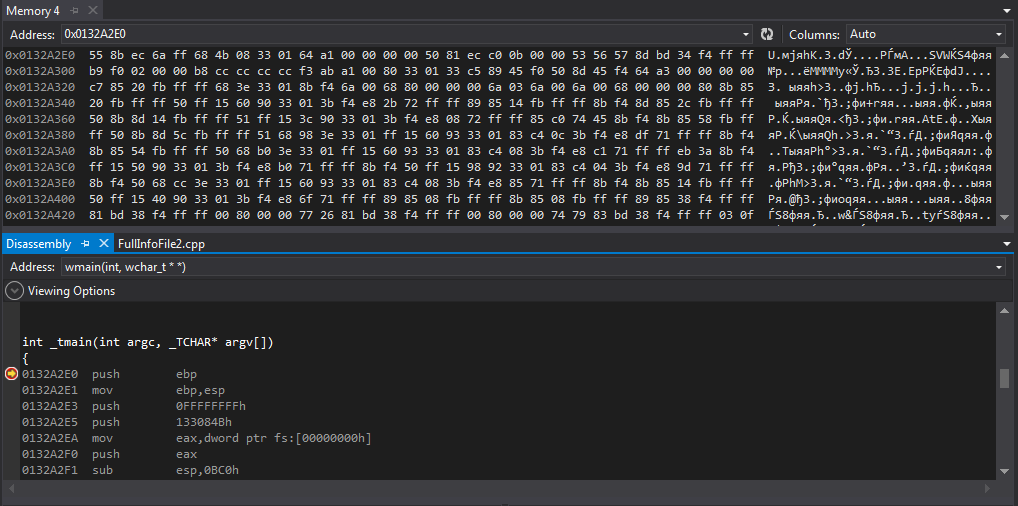


Рис. 5. Дизассемблированный код программы в памяти

1. **Профилирование в Windows**

Запустим тест производительности в Debug->Profiler->Performance profiler->Performance Wizard->Instrumentation. Желательно собрать проект в режиме «Release», потому что при сборке в режиме отладки код программы сопровождается дополнительными инструкциями и меньше оптимизируется. Отчет о производительности утилиты выводящей информацию о файле приведен на рис. 6.

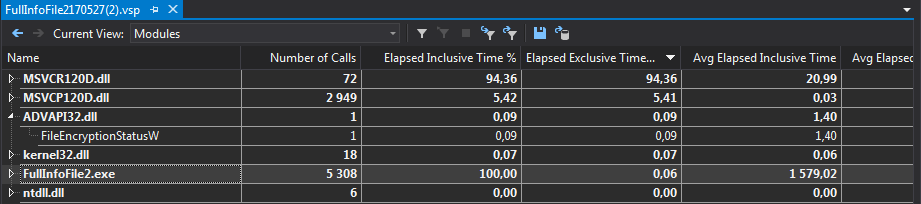


Рис. 6. Отчёт о производительность утилиты

Из рис. 6 видно, что большинство ресурсов тратится на вызовы внешних библиотек, например, на внешний код из библиотеки advapi32.dll, в которой представлены функции для работы с DACL.



Рис.7. Профайлер Visual Studio. Окно Functions

На рисунке 7 можно увидеть список выполняемых функций.

Для анализа используемой памяти можно так же использовать отладчик Visual Studio (Debug->Windows->Diagnostic Tools). Он позволяет отслеживать использование памяти программой в реальном времени, делать снимки памяти, а так же следить за состоянием кучи.

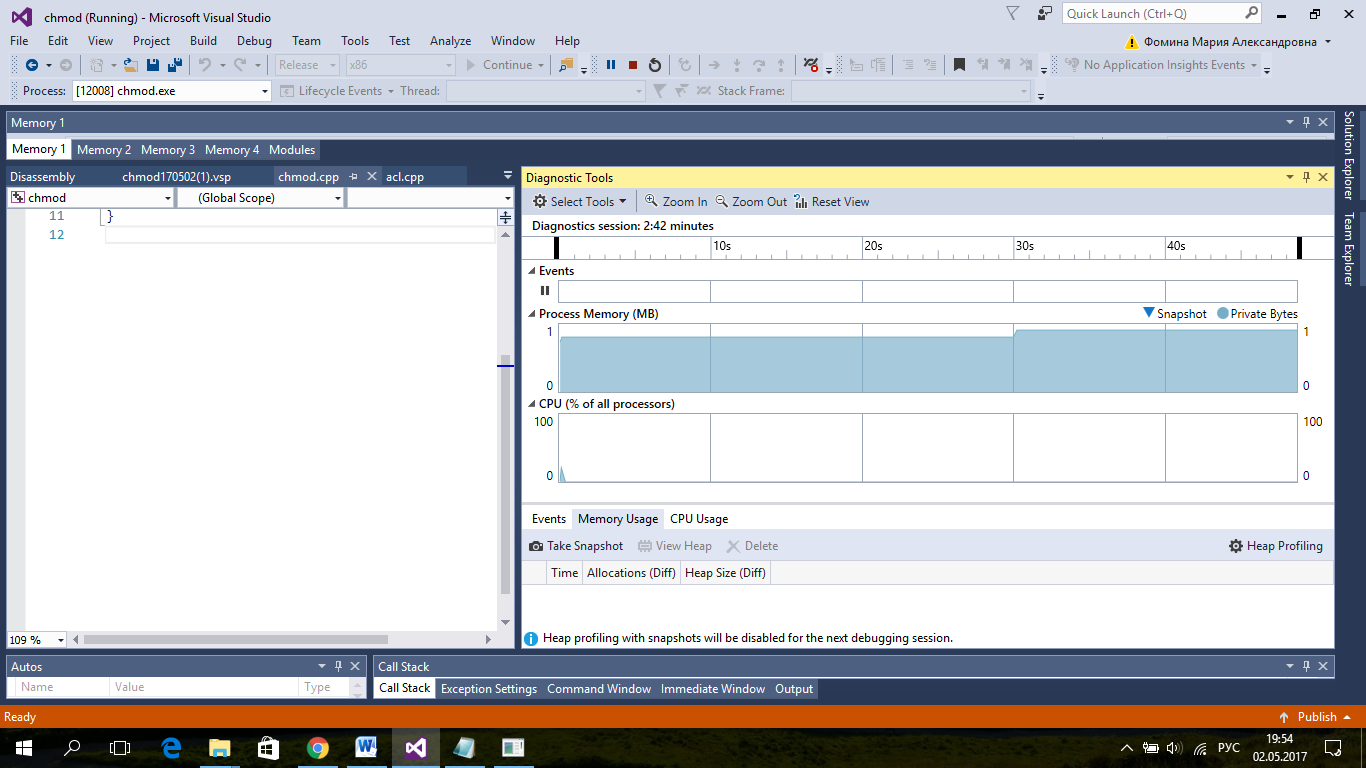


Рис. 8. Профилирование памяти

Из рис. 8 видно, что утилита требует около 1,2 Мб памяти. Куча пустая, т.к. написанная утилита не выделяет память в куче.

1. **Анализ адресного пространства библиотек в Windows**

**Visual Studio**

Проанализируем адресное пространство AppWithLibFullInfoFile и AppWithLibFullInfoFileDll.

Статическая библиотека связывается с программой на этапе компиляции, поэтому она сразу включена в адресное пространство программы. Проверить это можно, установив точку останова на первую же строчку программы и проанализировав адресное пространство.

Рассмотрим адресное пространство процесса при работе со статической библиотекой (проект AppWithLibFullInfoFile). На рис. 9 показано, что исходный код библиотеки изначально расположен в памяти программы.

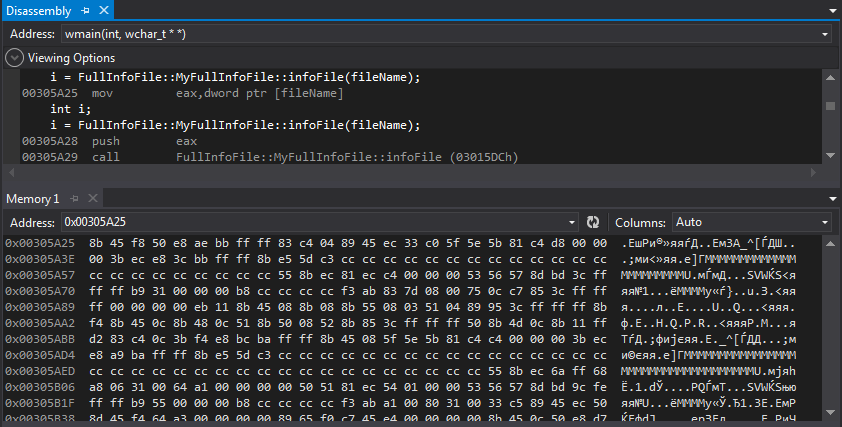
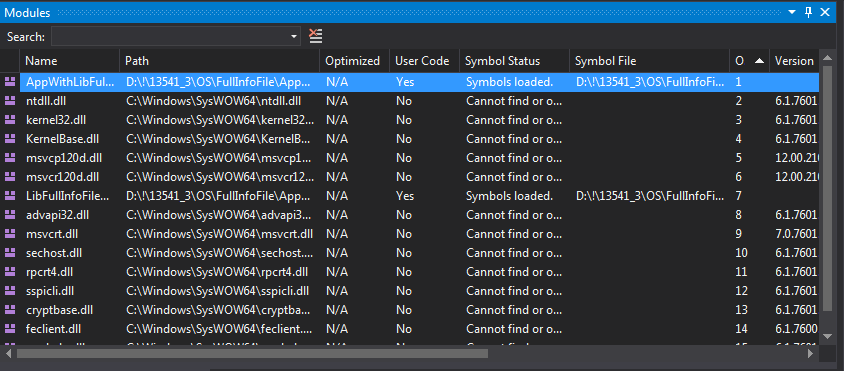


Рис. 9. Функция infoFile(fileName) статической библиотеки в адресном пространстве процесса AppWithLibFullInfoFile

Рассмотрим адресное пространство процесса при работе с динамической библиотекой (проект AppWithLibFullInfoFileDll). У динамической библиотеки LibFullInfoFileDLL.dll есть свое адресное пространство (рис. 10), функция infoFile(fileName) не находится в адресном пространстве процесса AppWithLibFullInfoFileDll (рис. 11).



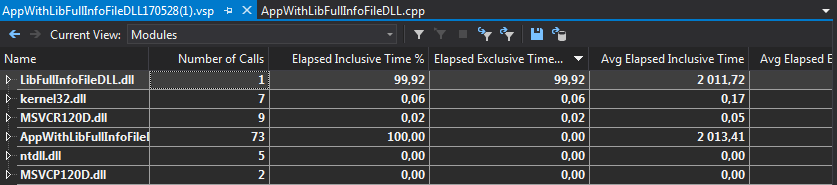


Рис. 10. Адресное пространство динамической библиотеки LibFullInfoFileDLL.dll

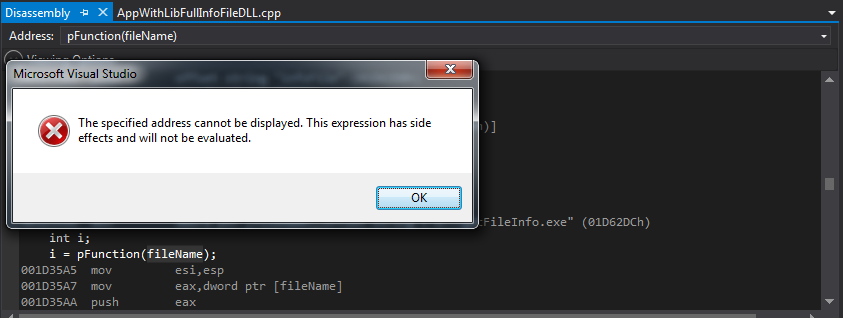


Рис. 11. Функция pFunction(fileName) не в адресном пространстве процесса AppWithLibFullInfoFileDll

**Утилита CodeAnalyst**

AMD CodeAnalyst — профилировщик кода с поддержкой графического интерфейса пользователя. Несмотря на то, что CodeAnalyst разрабатывался только для работы с процессорами производства AMD, он также может ограниченно (без поддержки аппаратных событий) работать и с процессорами производства Intel. CodeAnalyst имеет схожий внешний вид и поведение как на платформе Linux, так и на платформе Microsoft Windows.[1]

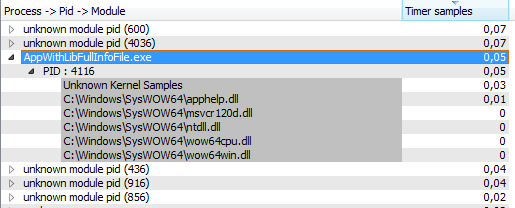


Рис.12 Анализ программы со статической библиотекой

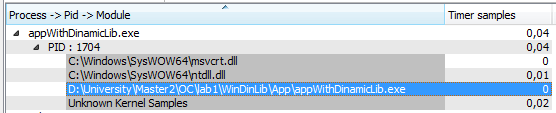


Рис. 13. Анализ программы c динамической библиотекой

Данная программа показывает какие библиотеки были использованы при запуске программы. Эти библиотеки автоматически добавляются в программу при компиляции. Например, из вывода видно, что используются системные динамические библиотеки с расширением .dll из папки SysWOW64.

Папка SysWOW64 — папка системных библиотек для запуска 32-разрядных приложений на 64-разрядной Windows. Все запускаемые 32-разрядные приложения используют библиотеки из этой папки.

Таким образом можно сделать вывод, что приложение скомпилировано под 32-х разрядную ОС

Программа также показывает время, затраченное на исполнение функций из системных библиотек.

Данная утилита предназначена для процессоров от AMD, в то время как на тестируемой машине установлен Intel процессор. Возможно, это является причиной того, что утилита предоставляет малое количество информации.

**4. Анализ адресного пространства в Linux**

Для анализа адресного пространства воспользуемся утилитой readelf. Утилита отображает информацию об одном или нескольких объектных файлов в формате ELF. Для начала убедимся, что файл относится к формату elf. Для этого воспользуемся утилитой file. Видим, что исполняемый файл программы относится к формату elf, значит, к нему можно применить утилиту readelf.

|  |
| --- |
| nikita@nikita-K53SM:~/labs/2 semester/lab4$ file /home/nikita/labs/lab6/infofile  /home/nikita/labs/lab6/infofile: **ELF 64-bit LSB executable**, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=d347e7d8ba6dc6295f77b98a5f281911291c4a76, not stripped  nikita@nikita-K53SM:~/labs/2 semester/lab4$ readelf -e /home/nikita/labs/lab6/infofile  Заголовок ELF:  Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00  Класс: ELF64  Данные: дополнение до 2, little endian  Версия: 1 (current)  OS/ABI: UNIX - System V  Версия ABI: 0  Тип: EXEC (Исполняемый файл)  Машина: Advanced Micro Devices X86-64  Версия: 0x1  **Адрес точки входа: 0x400570**  Начало заголовков программы: 64 (байт в файле)  Начало заголовков программы: 11184 (байт в файле)  ...  Заголовки программы:  Тип Смещ. Вирт.адр Физ.адр  Рзм.фйл Рзм.пм Флаги Выравн  PHDR 0x0000000000000040 0x0000000000400040 0x0000000000400040  0x00000000000001f8 0x00000000000001f8 R E 8  INTERP 0x0000000000000238 0x0000000000400238 0x0000000000400238  0x000000000000001c 0x000000000000001c R 1  [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]  **LOAD 0x0000000000000000 0x0000000000400000 0x0000000000400000**  **0x000000000000122c 0x000000000000122c R E 200000**  LOAD 0x0000000000001e10 0x0000000000601e10 0x0000000000601e10  0x0000000000000248 0x0000000000000250 RW 200000  ... |

Адрес точки входа программы **0x400570**. Рассмотрим подробнее одну из строк (выделенную) таблицы заголовков программы. Таблица состоит из следующих полей: смещение, виртуальный адрес, физический адрес, размер файла, размер памяти, флаги и выравнивание. Заголовок программы LOAD размещен по виртуальному адресу, равному физическому, **0x400000** со смещением **0x0**. Флаги разрешают чтение и исполнение, этот раздел содержит непосредственно код программы.

Разберем загрузку программы в память более подробно. Для этого воспользуемся утилитой **objdump** с флагом **-d** для того, чтобы получить ассемблерный код. Обратим особое внимание на адрес **0x400570** – точка входа программы.

|  |
| --- |
| nikita@nikita-K53SM:~/labs/2 semester/lab4$ objdump -d /home/nikita/labs/lab6/infofile  …  Дизассемблирование раздела .text:  0000000000400570 <\_start>:  400570: 31 ed xor %ebp,%ebp  400572: 49 89 d1 mov %rdx,%r9  400575: 5e pop %rsi  400576: 48 89 e2 mov %rsp,%rdx  400579: 48 83 e4 f0 and $0xfffffffffffffff0,%rsp  40057d: 50 push %rax  40057e: 54 push %rsp  40057f: 49 c7 c0 60 0c 40 00 mov $0x400c60,%r8  400586: 48 c7 c1 f0 0b 40 00 mov $0x400bf0,%rcx  40058d: 48 c7 c7 66 06 40 00 mov $0x400666,%rdi  400594: e8 a7 ff ff ff callq 400540 <\_\_libc\_start\_main@plt>  400599: f4 hlt  40059a: 66 0f 1f 44 00 00 nopw 0x0(%rax,%rax,1)  ... |

Первой запускается процедура "\_start". Она очищает регистр ebp, записывает какие-то значения в стек и вызывает подпрограмму.

***0x400666 - адрес функции main.***

***0x400c60 - адрес функции \_\_libc\_csu\_fini.***

***0x400bf0 - адрес функции \_\_libc\_csu\_init.***

***0x4004d0 - адрес функции \_init.***

***0x400c80 - адрес функции \_fini.***

Функции \_init и \_fini это функции инициализации и завершения приложения, генерируемые компилятором GCC.

Просмотрим все косвенные ссылки:

|  |
| --- |
| nikita@nikita-K53SM:~/labs/2 semester/lab4$ objdump -R /home/nikita/labs/lab6/infofile  /home/nikita/labs/lab6/infofile: формат файла elf64-x86-64  DYNAMIC RELOCATION RECORDS  OFFSET TYPE VALUE  0000000000601ff8 R\_X86\_64\_GLOB\_DAT \_\_gmon\_start\_\_  0000000000602018 R\_X86\_64\_JUMP\_SLOT puts@GLIBC\_2.2.5  0000000000602020 R\_X86\_64\_JUMP\_SLOT ctime@GLIBC\_2.2.5  0000000000602028 R\_X86\_64\_JUMP\_SLOT \_\_stack\_chk\_fail@GLIBC\_2.4  0000000000602030 R\_X86\_64\_JUMP\_SLOT printf@GLIBC\_2.2.5  **0000000000602038 R\_X86\_64\_JUMP\_SLOT \_\_libc\_start\_main@GLIBC\_2.2.5**  0000000000602040 R\_X86\_64\_JUMP\_SLOT \_\_xstat@GLIBC\_2.2.5 |

По указанному адресу 0x602038 расположена функция \_\_libc\_start\_main. В задачу этой функции входят некоторые действия по инициализации среды исполнения и вызов функции main().

Для того чтобы посмотреть динамические объекты программы, используем флаг d c утилитой readelf.

|  |
| --- |
| nikita@nikita-K53SM:~/labs/2 semester/lab4$ readelf -d /home/nikita/labs/lab6/infofile  Dynamic section at offset 0x1e28 contains 24 entries:  Тег Тип Имя/Знач  0x0000000000000001 (NEEDED) Совм. исп. библиотека: [libc.so.6]  0x000000000000000c (INIT) 0x4004d0  0x000000000000000d (FINI) 0x400c80  ... |

С помощью gdb можно также получить информацию об адресном пространстве.

|  |
| --- |
| nikita@nikita-K53SM:~/labs/2 semester/lab4$ gdb /home/nikita/labs/lab6/infofile  (gdb) print \_init  $1 = {<text variable, no debug info>} 0x4004d0 <\_init>  (gdb) print \_fini  $2 = {<text variable, no debug info>} 0x400c80 <\_fini>  (gdb) print main  $3 = {<text variable, no debug info>} 0x400666 <main>  (gdb) print \_start  $4 = {<text variable, no debug info>} **0x400570** <\_start> |

Из результатов видно, что адрес точки входа совпадает с тем, что показал дизассемблированный код. Убедимся в том, что статические библиотеки сразу связываются с исполняемым файлом (исполняемый файл infofile1 использует для работы статическую библиотеку):

|  |
| --- |
| nikita@nikita-K53SM:~/labs/2 semester/lab4$ gdb /home/nikita/labs/lab6/infofile1  (gdb) print ftypelet  $1 = {<text variable, no debug info>} **0x40071a** <ftypelet(unsigned int)>  (gdb) print strmode  $2 = {<text variable, no debug info>} **0x4007c5** <strmode(unsigned int, char\*)>  (gdb) print printFileType  $3 = {<text variable, no debug info>} **0x400991** <printFileType(stat)>  (gdb) print printInfo  $4 = {<text variable, no debug info>} **0x400a3b** <printInfo(stat)>  …  nikita@nikita-K53SM:~/labs/2 semester/lab4$ readelf -d /home/nikita/labs/lab6/infofile1  Dynamic section at offset 0x1e28 contains 24 entries:  Тег Тип Имя/Знач  0x0000000000000001 (NEEDED) Совм. исп. библиотека: [libc.so.6]  0x000000000000000c (INIT) 0x4004d0  0x000000000000000d (FINI) 0x400c80  ... |

Из результатов видно, что удалось получить адреса всех функций из статической библиотеки.

Теперь проанализируем программу infofile2, в которой подключается динамическая библиотека.

|  |
| --- |
| nikita@nikita-K53SM:~/labs/2 semester/lab4$ gdb /home/nikita/labs/lab6/dynamic/infofile2  **(gdb) print printFileType**  **Таблица символов не загружена. Используйте меню файл**  (gdb) b main.c:13  Make breakpoint pending on future shared library load? (y or [n]) y  Breakpoint 1 (main.c:13) pending.  (gdb) r  Starting program: /home/nikita/labs/lab6/dynamic/infofile2  [Inferior 1 (process 3905) exited with code 01]  **(gdb) print printFileType**  **$1 = {<text variable, no debug info>} 0x7ffff7bd4ab7 <printFileType>**  (gdb) print main  $2 = {<text variable, no debug info>} 0x400736 <main>  (gdb) print \_start  $3 = {<text variable, no debug info>} 0x400640 <\_start>  ...  nikita@nikita-K53SM:~/labs/2 semester/lab4$ readelf -d /home/nikita/labs/lab6/dynamic/infofile2  Dynamic section at offset 0xe18 contains 25 entries:  Тег Тип Имя/Знач  **0x0000000000000001 (NEEDED) Совм. исп. библиотека: [lib2.so]**  0x0000000000000001 (NEEDED) Совм. исп. библиотека: [libc.so.6]  …  //Получение информации о загруженных библиотеках с помощью gdb  nikita@nikita-K53SM:~/labs/2 semester/lab4$ gdb /home/nikita/labs/lab6/dynamic/infofile2  **(gdb) info sharedlibrary**  **No shared libraries loaded at this time.**  (gdb) r  Starting program: /home/nikita/labs/lab6/dynamic/infofile2  [Inferior 1 (process 7390) exited with code 01]  **(gdb) info sharedlibrary**  From To Syms Read Shared Object Library  0x00007ffff7dd7ac0 0x00007ffff7df5640 Yes /lib64/ld-linux-x86-64.so.2  **0x00007ffff7bd4740 0x00007ffff7bd4d38 Yes (\*) /home/nikita/labs/lab6/dynamic/lib2.so**  0x00007ffff782a8b0 0x00007ffff797cf14 Yes /lib/x86\_64-linux-gnu/libc.so.6  (\*): Shared library is missing debugging information.  (gdb) q |

Функция printFileType из динамической библиотеки не находятся в адресном пространстве процесса, при попытке получить её адрес отладчик выводит ошибку. Была установлена точка останова на строчку программы, в которой происходит вызов функции библиотеки. Была запущена программа, в точке останова можно получить адрес функции динамической библиотеки (0x7ffff7bd4ab7).

Найдем точку входа программы, реализованной в виде демона.

|  |
| --- |
| nikita@nikita-K53SM:~$ cd labs/2\ semester/lab1/daemon/  nikita@nikita-K53SM:~/labs/2 semester/lab1/daemon$ readelf -e ./infofile  Заголовок ELF:  Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00  Класс: ELF64  Данные: дополнение до 2, little endian  Версия: 1 (current)  OS/ABI: UNIX - System V  Версия ABI: 0  Тип: EXEC (Исполняемый файл)  Машина: Advanced Micro Devices X86-64  Версия: 0x1  Адрес точки входа: **0x4009b0**  Начало заголовков программы: 64 (байт в файле)  Начало заголовков программы: 12184 (байт в файле)  ...  **LOAD 0x0000000000000000 0x0000000000400000 0x0000000000400000**  **0x0000000000001c8c 0x0000000000001c8c R E 200000**  ... |

Адрес точки входа программы **0x4009b0**. Программный код находится также по физическому и виртуальному адресах **0x400000** со смещением **0x0**.

Из вывода утилиты **readelf** для обычной программы и программы-демона видно, что программы-демоны загружаются в память ОС аналогично тому, как загружаются обычные программы.

**5. Профилирование в Linux**

Для профилирования программы воспользуемся утилитой valgrind. Данная утилита включает в себя набор анализаторов, в том числе анализатор производительности и анализатор использования памяти.

Запустим анализ производительности:

|  |
| --- |
| nikita@nikita-K53SM:~/labs/2 semester/lab4$ valgrind --tool=callgrind /home/nikita/labs/lab6/infofile log.linux.txt  ==18217== Callgrind, a call-graph generating cache profiler  ==18217== Copyright (C) 2002-2015, and GNU GPL'd, by Josef Weidendorfer et al.  ==18217== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info  ==18217== Command: /home/nikita/labs/lab6/infofile log.linux.txt  ==18217==  ==18217== For interactive control, run 'callgrind\_control -h'.  ...  ==18217==  ==18217== Events : Ir  ==18217== Collected : 205028  ==18217==  ==18217== I refs: 205,028  nikita@nikita-K53SM:~/labs/2 semester/lab4$ ls -l  итого 2772  -rw------- 1 nikita nikita 66138 май 3 04:06 callgrind.out.18217  -rw-rw-r-- 1 nikita nikita 62503 май 3 04:04 log.linux.txt  -rw------- 1 nikita nikita 8052 май 3 03:59 massif.out.16719 |

Данная утилита генерирует выходной файл с названием callgrind.out.18217. В текстовом виде просматривать его неудобно. Для визуализации воспользуемся утилитой KCacheGrind (рис. 14) в режиме профилирования ELF-объектов.

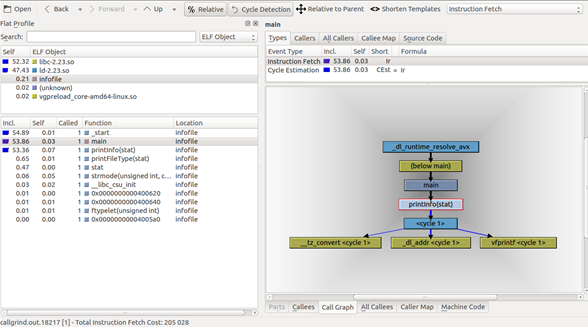


Рис. 14. Анализ производительности в KCacheGrind

В верхней левой части окна можно выбрать анализируемый ELF-объект. Слева снизу находится список функций в данном объекте, в левом столбце отображена ресурсоемкость функций. Выбрав определенную функцию, справа можно увидеть граф вызовов функций.

Для анализа используемой памяти используется инструмент massif.

|  |
| --- |
| nikita@nikita-K53SM:~/labs/2 semester/lab4$ valgrind --tool=massif /home/nikita/labs/lab6/infofile log.linux.txt  ==16719== Massif, a heap profiler  ==16719== Copyright (C) 2003-2015, and GNU GPL'd, by Nicholas Nethercote  ==16719== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info  ==16719== Command: /home/nikita/labs/lab6/infofile log.linux.txt  ==16719==  ...  nikita@nikita-K53SM:~/labs/2 semester/lab4$ massif-visualizer massif.out.16719  massif-visualizer(16804)/kdeui (kdelibs): Attempt to use QAction "toggleDataTree" with KXMLGUIFactory!  massif-visualizer(16804)/kdeui (kdelibs): Attempt to use QAction "toggleAllocators" with KXMLGUIFactory!  description: "(none)"  command: "/home/nikita/labs/lab6/infofile log.linux.txt"  time unit: "i"  snapshots: 34  peak: snapshot # 6 after "130512i"  peak cost: "6,6 KiB" heap "59 B" heap extra "0 B" stacks |

Для просмотра отчета massif можно использовать инструмент massif-vizualizer (рис. 15). На графике слева видно, что программа использует около 6,6 Кб памяти. График при этом цветом выделяет, какой объем в памяти занимает каждый объект. Справа можно выбрать один из снимков памяти (который делает massif во время анализа) и более подробно посмотреть какие объекты и где занимают память.

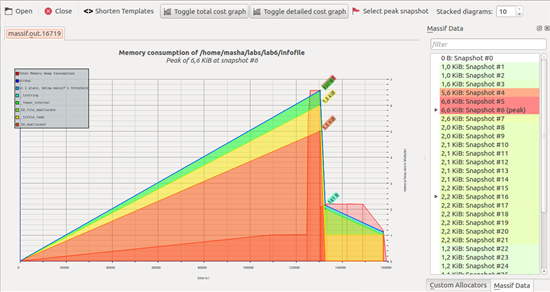


Рис. 15. Анализ памяти

Были определены адреса точек входа программ, функции main и других объектов программы. Определено, что статические библиотеки располагаются в адресном пространстве процесса, а динамические загружаются во время выполнения программ. Программы-демоны и службы загружаются в память ОС аналогично тому, как загружаются обычные программы.

Профилирование показало, что наиболее ресурсоемкими функциями в программах являются те, которые производят вызовы функций WinAPI или системных вызовов Linux.

**Дополнение**

**6. Создание дампа памяти в Windows**

В диспетчере задач можно сделать дамп памяти выполняющейся программы. Был сделан дамп FullInfoFile2.exe (рис. 16 и рис. 17).

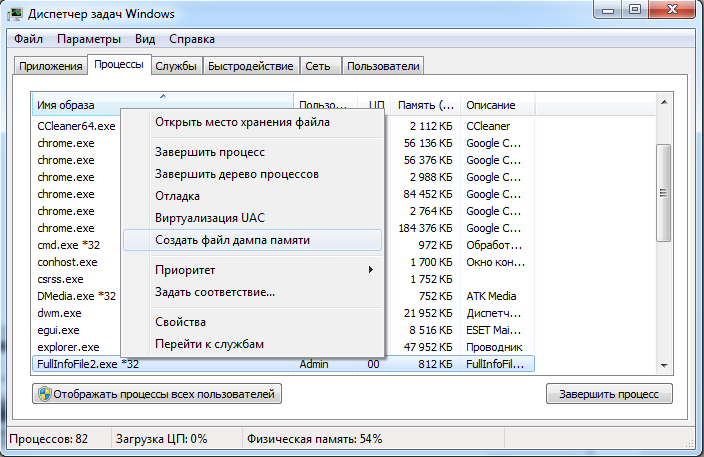


Рис. 16. Создание дампа памяти FullInfoFile2.exe

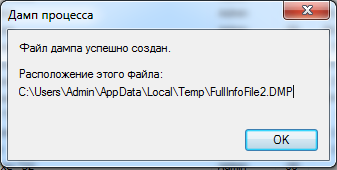


Рис. 17. Создание дампа памяти FullInfoFile2.exe

В Visual Studio был просмотрен FullInfoFile2.DMP и запущен Debug with Native Only (рис. 18 и рис. 19).

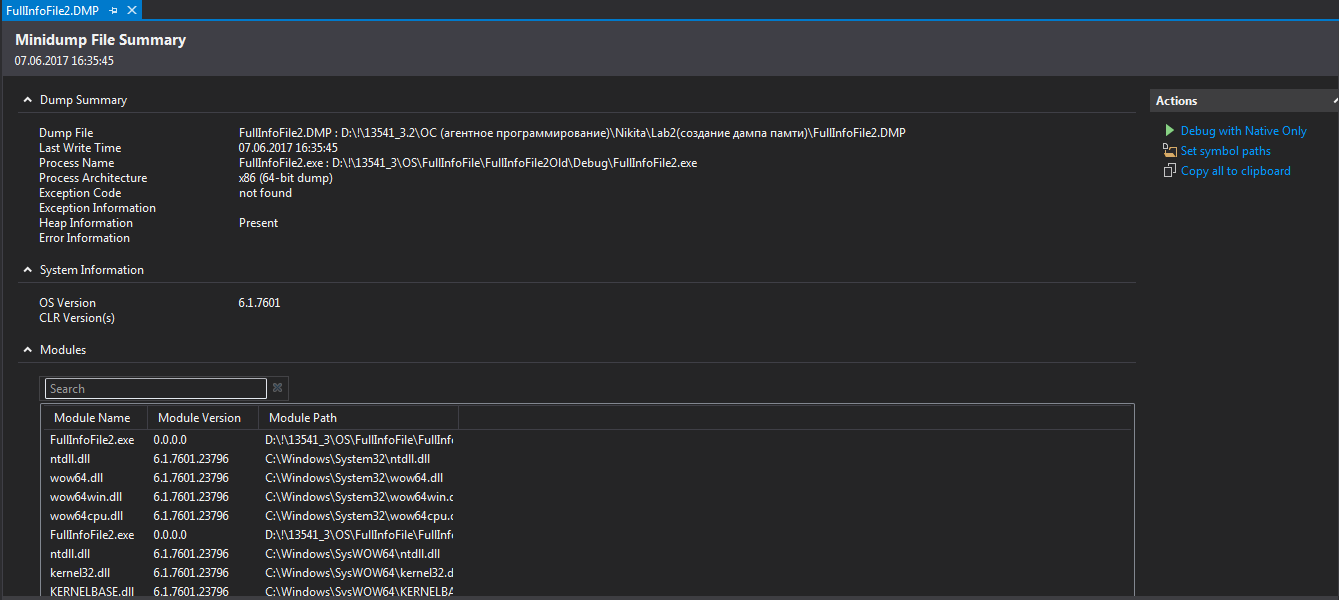


Рис. 18. Просмотр информации о дампе

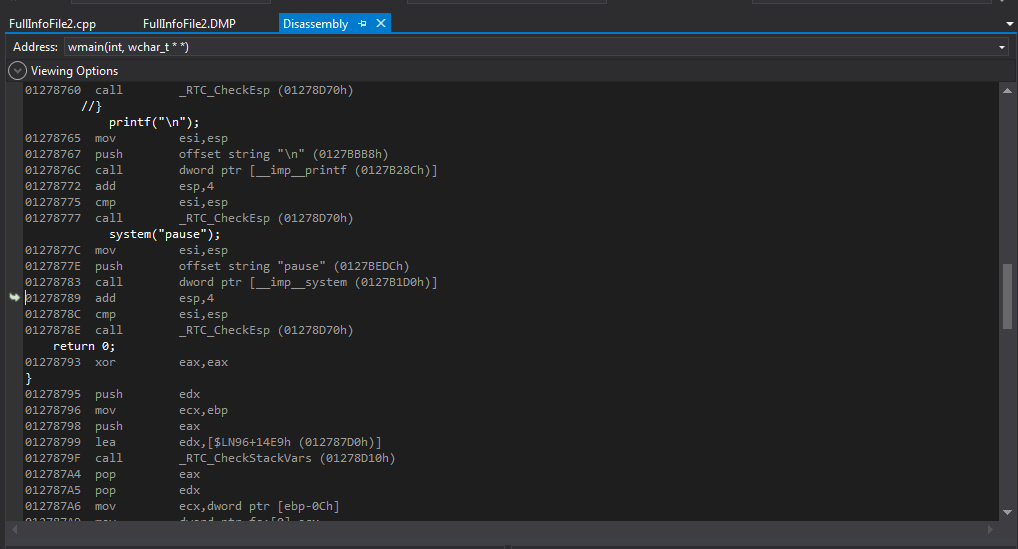


Рис. 19. Поиск адреса функции main в дампе

Также дамп памяти можно сделать с помощию программы procdump [5]. Опция **ma** позволяет записать полный дамп-файл процесса.

|  |
| --- |
| D:\!\13541\_3.2\ОС (агентное программирование)\Nikita\Lab2(создание дампа памти)\  Procdump>procdump.exe -ma FullInfoFile2.exe FullInfoFile2-procdump.dmp  ProcDump v9.0 - Sysinternals process dump utility  Copyright (C) 2009-2017 Mark Russinovich and Andrew Richards  Sysinternals - www.sysinternals.com  [17:21:03] Dump 1 initiated: D:\!\13541\_3.2\[17:21:03] Dump 1 writing: Estimated  dump file size is 12 MB.  [17:21:03] Dump 1 complete: 12 MB written in 0.3 seconds  [17:21:04] Dump count reached. |

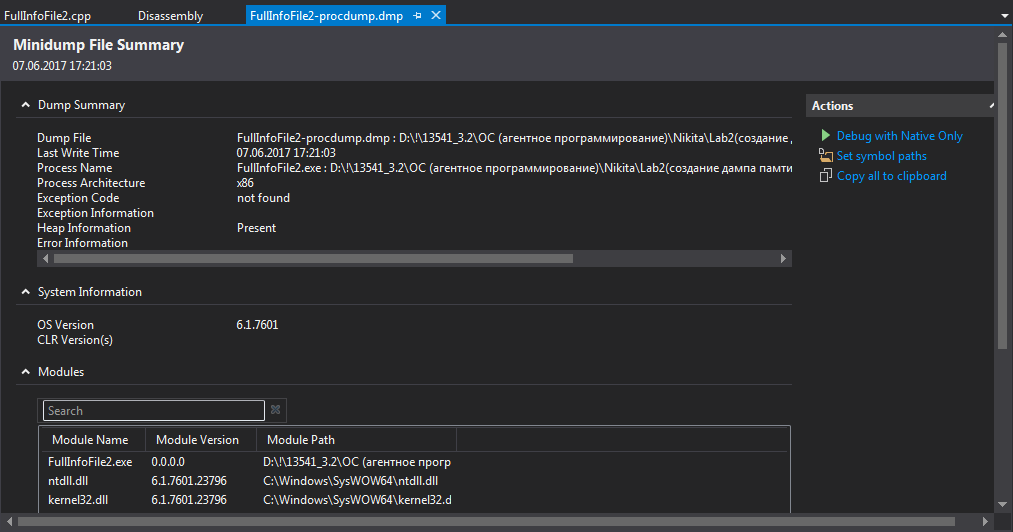


Рис. 20. Просмотр дампа памяти, созданного с помощью procdump

**7. Создание дампа памяти в Linux**

**/proc/$pid/maps** отображает список виртуальных адресов для процесса с заданным $pid.

**/proc/$pid/pagemap** отображает более подробную информацию о каждой странице, включая физические адреса, если они существуют.

Утилита **gcore** позволяет сделать полный дамп исполняемого процесса:

***gcore [-o filename] pid***

Команда **GDB: dump memory** позволяет сделать дамп содержимого памяти процесса в диапазоне адресов. Дамп можно сделать в различных форматах. Команда **restore** позволяет восстановить содержимое памяти, если передан дамп памяти в бинарном формате, то нужно после названия файла писать слово **binary**.

|  |
| --- |
| **//Скомпилируем демон с опцией отладки (-g)**  nikita@nikita-K53SM:~/labs/2semester/lab1/daemon$ g++ -g -o infofile main.c  …  **//Запущен демон, записывающий информацию о файле каждую минуту**  nikita@nikita-K53SM:~/labs/2semester/lab1/daemon$ ./infofile "/home/nikita/labs/2semester/lab1/daemon/temp.txt"  nikita@nikita-K53SM:~/labs/2semester/lab1/daemon$ ps -aux | grep infofile  nikita 3804 0.0 0.0 4360 92 ? Ss 17:07 0:00 ./infofile /home/nikita/labs/2semester/lab1/daemon/temp.txt  nikita 3843 0.0 0.0 15472 960 pts/4 S+ 17:07 0:00 grep --color=auto infofile  nikita@nikita-K53SM:~/labs/2semester/lab1/daemon$ cd /home/nikita/labs/2semester/lab4/  **//Сделаем дамп процесса демона**  nikita@nikita-K53SM:~/labs/2semester/lab4$ **sudo gcore -o infofile-daemon.dump 3804**  0x00007fd6d372d740 in \_\_nanosleep\_nocancel () at ../sysdeps/unix/syscall-template.S:84  84 ../sysdeps/unix/syscall-template.S: Нет такого файла или каталога.  warning: target file /proc/3804/cmdline contained unexpected null characters  Saved corefile infofile-daemon.dump.3804  **//Запускаем gdb с исполняемым файлом и дампом**  nikita@nikita-K53SM:~/labs/2semester/lab4$ **gdb /home/nikita/labs/2semester/lab1/daemon/infofile infofile-daemon.dump.3804**  …  Core was generated by `./infofile'.  …  **//Детальная трассировка**  **(gdb) bt full**  #0 0x00007fd6d372d740 in \_\_nanosleep\_nocancel ()  at ../sysdeps/unix/syscall-template.S:84  No locals.  #1 0x00007fd6d372d6aa in \_\_sleep (seconds=0) at ../sysdeps/posix/sleep.c:55  save\_errno = 0  ts = {tv\_sec = 23, tv\_nsec = 917475280}  #2 0x0000000000400d8b in Daemon (argc=2, argv=0x7ffe7e2870e8) at main.c:53  log = 0x602160 <printInfo(stat)::str> "Размер файла: \n\t808 байт\nЧисло жёстких ссылок: \n\t1\ninode файла: \n\t789900\nID устройства, содержащего файл: \n\t2055\nID пользоват", <incomplete sequence \320>...  fStat = {st\_dev = 2055, st\_ino = 789900, st\_nlink = 1,  st\_mode = 33204, st\_uid = 1000, st\_gid = 1000, \_\_pad0 = 0,  st\_rdev = 0, st\_size = 808, st\_blksize = 4096, st\_blocks = 8,  st\_atim = {tv\_sec = 1493929426, tv\_nsec = 751658596}, st\_mtim = {  tv\_sec = 1493929426, tv\_nsec = 751658596}, st\_ctim = {  tv\_sec = 1493929426, tv\_nsec = 819660101}, \_\_glibc\_reserved = {0,  0, 0}}  #3 0x0000000000400e3a in main (argc=2, argv=0x7ffe7e2870e8) at main.c:79  parpid = 0  sid = 3804  **//Вывод трассировки всех нитей**  **(gdb) thread apply all bt**  Thread 1 (LWP 3804):  #0 0x00007fd6d372d740 in \_\_nanosleep\_nocancel ()  at ../sysdeps/unix/syscall-template.S:84  #1 0x00007fd6d372d6aa in \_\_sleep (seconds=0) at ../sysdeps/posix/sleep.c:55  #2 0x0000000000400d8b in Daemon (argc=2, argv=0x7ffe7e2870e8) at main.c:53  #3 0x0000000000400e3a in main (argc=2, argv=0x7ffe7e2870e8) at main.c:79  **// Если бы было несколько нитей, для переключения между ними можно было бы использовать команду "thread <number>"**  **// Переключаемся на фрейм 2, где находится функция мониторинга файла**  **(gdb) frame 2**  #2 0x0000000000400d8b in Daemon (argc=2, argv=0x7ffe7e2870e8) at main.c:53  53 sleep(60);  **//Просмотрим исходный код, который выполнялся при записи дампа**  **(gdb) list +**  48 writeLog("Cycle terminated. Invalid arguments!");  49 return 1;  50 }  51 log = printInfo(fStat);  52 writeLog(log);  53 sleep(60);  54 }  55 return 0;  56 }  57  **//Выведем значение переменной log**  **(gdb) print log**  $1 = 0x602160 <printInfo(stat)::str> "Размер файла: \n\t808 байт\nЧисло жёстких ссылок: \n\t1\ninode файла: \n\t789900\nID устройства, содержащего файл: \n\t2055\nID пользоват", <incomplete sequence \320>...  **//Выведем адрес функции Daemon**  **(gdb) print Daemon**  $2 = {int (int, char \*\*)} 0x400cb9 <Daemon(int, char\*\*)>  (gdb) q  **//Просмотр виртуального адресного пространства демона**  nikita@nikita-K53SM:~/labs/2semester/lab4$ **cat /proc/3804/maps**  **// address perms offset dev inode pathname**  **// dev - device (major:minor)**  00400000-00402000 r-xp 00000000 08:07 793077 /home/nikita/labs/2semester/lab1/daemon/infofile  00601000-00602000 r--p 00001000 08:07 793077 /home/nikita/labs/2semester/lab1/daemon/infofile  00602000-00603000 rw-p 00002000 08:07 793077 /home/nikita/labs/2semester/lab1/daemon/infofile  01e01000-01e22000 rw-p 00000000 00:00 0 [heap]  7fd6d3662000-7fd6d3821000 r-xp 00000000 08:07 793162 /lib/x86\_64-linux-gnu/libc-2.23.so  …  **//Создание и восстановление дампов памяти в диапазоне адресов**  nikita@nikita-K53SM:~/labs/2semester/lab4$ **sudo gdb --pid 3804**  **…**  **(gdb) dump memory infofile-daemon.dump 0x00400000 0x00402000**  **(gdb) dump ihex memory infofile-daemon1.dump 0x00400000 0x00402000**  **(gdb) dump srec memory infofile-daemon2.dump 0x00400000 0x00402000**  **(gdb) dump tekhex memory infofile-daemon3.dump 0x00400000 0x00402000**  **(gdb) restore infofile-daemon.dump binary 0x00400000**  **Restoring binary file infofile-daemon.dump into memory (0x400000 to 0x402000)**  **(gdb) restore infofile-daemon1.dump**  **Restoring section .sec1 (0x400000 to 0x402000)**  **(gdb) restore infofile-daemon2.dump**  **Restoring section .sec1 (0x400000 to 0x402000)**  **(gdb) restore infofile-daemon3.dump**  **Restoring section .newsec (0x400000 to 0x402000)**  **//Детальная трассировка**  **(gdb) bt full**  #0 0x00007fd6d372d740 in \_\_nanosleep\_nocancel () at ../sysdeps/unix/syscall-template.S:84  No locals.  #1 0x00007fd6d372d6aa in \_\_sleep (seconds=0) at ../sysdeps/posix/sleep.c:55  save\_errno = 0  ts = {tv\_sec = 8, tv\_nsec = 87455279}  #2 0x0000000000400d8b in Daemon (argc=2, argv=0x7ffe7e2870e8) at main.c:53  log = 0x602160 <printInfo(stat)::str> "Размер файла: \n\t808 байт\nЧисло жёстких ссылок: \n\t1\ninode файла: \n\t789900\nID устройства, содержащего файл: \n\t2055\nID пользоват", <incomplete sequence \320>...  fStat = {st\_dev = 2055, st\_ino = 789900, st\_nlink = 1, st\_mode = 33204, st\_uid = 1000, st\_gid = 1000,  \_\_pad0 = 0, st\_rdev = 0, st\_size = 808, st\_blksize = 4096, st\_blocks = 8, st\_atim = {tv\_sec = 1493929426,  tv\_nsec = 751658596}, st\_mtim = {tv\_sec = 1493929426, tv\_nsec = 751658596}, st\_ctim = {  tv\_sec = 1493929426, tv\_nsec = 819660101}, \_\_glibc\_reserved = {0, 0, 0}}  #3 0x0000000000400e3a in main (argc=2, argv=0x7ffe7e2870e8) at main.c:79  parpid = 0  sid = 3804  **(gdb) frame 2**  #2 0x0000000000400d8b in Daemon (argc=2, argv=0x7ffe7e2870e8) at main.c:53  53 sleep(60);  **(gdb) print log**  $1 = 0x602160 <printInfo(stat)::str> "Размер файла: \n\t808 байт\nЧисло жёстких ссылок: \n\t1\ninode файла: \n\t789900\nID устройства, содержащего файл: \n\t2055\nID пользоват", <incomplete sequence \320>...  (gdb) detach  Detaching from program: /home/nikita/labs/2semester/lab1/daemon/infofile, process 3804  (gdb) q |

**Выводы**

Были определены адреса точек входа программ, функции main и других объектов программы. Определено, что статические библиотеки располагаются в адресном пространстве процесса, а динамические загружаются во время выполнения программ. Программы-демоны и службы загружаются в память ОС аналогично тому, как загружаются обычные программы.

Профилирование показало, что наиболее ресурсоемкими функциями в программах являются те, которые производят вызовы функций WinAPI или системных вызовов Linux.

**Список использованных источников**

1. <https://www.gnu.org/software/gdb/>
2. <http://valgrind.org/>
3. <https://msdn.microsoft.com/ru-ru/library/c1h23y6c.aspx>