Санкт-Петербургский Политехнический Университет Петра Великого
Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

Отчёт по лабораторной работе №6
Курс: Системное программирование
Тема: Инструментальные средства профилирования, трассировки и диагностики в ОС

Выполнил студент группы 13541/3 _____ Д.В. Круминьш
(подпись)

Преподаватель _____ Е.В. Душутина
(подпись)

Санкт-Петербург
2017 г.

Содержание

Постановка задачи

Для операционных систем Windows и Linux необходимо провести обзор инструментов позволяющих:

1. Профилировать программу;
2. Вести трассировку вызовов программы;
3. Запускать программу в режиме отладки.

Подготовка к выполнению работы

2.1 Сведения о системе Windows

Работа производилась на реальной системы, с параметрами представленными ниже.

Элемент	Значение
Имя ОС	Майкрософт Windows 10 Pro (Registered Trademark)
Версия	10.0.14393 Сборка 14393
Дополнительное описание ОС	Недоступно
Изготовитель ОС	Microsoft Corporation
Имя системы	USER-PC
Изготовитель	HP
Модель	OMEN by HP Laptop 15-ce0xx
Тип	Компьютер на базе x64
SKU системы	1ZB00EA#ACB
Процессор	Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz, 2496 МГц, ядер: 4, логических процессоров: 4
Версия BIOS	American Megatrends Inc. F.04, 10.05.2017
Версия SMBIOS	3.0
Версия встроенного контроллера	40.20
Режим BIOS	Устаревший
Изготовитель основной платы	HP
Модель основной платы	Недоступно
Имя основной платы	Основная плата
Роль платформы	Мобильный
Состояние безопасной загрузки	Не поддерживается
Конфигурация PCR7	Привязка невозможна
Папка Windows	C:\Windows
Системная папка	C:\Windows\system32
Устройство загрузки	\Device\HarddiskVolume1
Язык системы	Россия

Аппаратно-зависимый уровень (HAL)	Версия = ”10.0.14393.1378”
Имя пользователя	USER-PC\Tom
Часовой пояс	RTZ 2 (зима)
Установленная оперативная память (RAM)	8,00 ГБ
Полный объем физической памяти	7,87 ГБ
Доступно физической памяти	3,54 ГБ
Всего виртуальной памяти	12,6 ГБ
Доступно виртуальной памяти	6,82 ГБ
Размер файла подкачки	4,75 ГБ
Файл подкачки	C:\pagefile.sys

Таблица 2.1: Информация об используемой системе Windows

2.1.1 Версии используемых программ

Работа происходила с использованием:

- Microsoft Visual Studio Enterprise 2017 (Версия 15.3.0);
- API Monitor v2 Alpha-r13;
- WinDbg Preview 10.0.16465.1002.

2.2 Сведения о системе Linux

Все опыты проводятся на виртуальной системе(64 битной), сведения о которой приведены в листинге 2.1.

```
root@kali:~/Desktop/testFolder# cat /etc/*release*
DISTRIB_ID=Kali
DISTRIB_RELEASE=kali-rolling
DISTRIB_CODENAME=kali-rolling
DISTRIB_DESCRIPTION="Kali GNU/Linux Rolling"
PRETTY_NAME="Kali GNU/Linux Rolling"
NAME="Kali GNU/Linux"
ID=kali
VERSION="2017.2"
VERSION_ID="2017.2"
ID_LIKE=debian
ANSI_COLOR="1;31"
HOME_URL="http://www.kali.org/"
SUPPORT_URL="http://forums.kali.org/"
BUG_REPORT_URL="http://bugs.kali.org/"
```

```
16
17 root@kali:~# uname -r
18 4.12.0-kali1-amd64
19
20 root@kali:~# lscpu
21 Architecture:          x86_64
22 CPU op-mode(s):        32-bit, 64-bit
23 Byte Order:            Little Endian
24 CPU(s):                4
25 On-line CPU(s) list:   0-3
26 Thread(s) per core:    1
27 Core(s) per socket:    2
28 Socket(s):             2
29 NUMA node(s):          1
30 Vendor ID:             GenuineIntel
31 CPU family:            6
32 Model name:            Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz
33 Stepping:              9
34 CPU MHz:               2495.998
35 Hypervisor vendor:     VMware
36 Virtualization type:   full
37 L1d cache:             32K
38 L1i cache:             32K
39 L2 cache:              256K
40 L3 cache:              6144K
41 NUMA node0 CPU(s):     0-3
```

Листинг 2.1: Информация об используемой системе Linux

2.2.1 Версии используемых программ

Работа происходила с использованием:

- Текстового редактора Sublime text(версии 3.0, сборка 3143);
- Компилятора gcc(версия Debian 7.2.0-4);
- Strace(версия 4.15);
- Отладчика GNU Debugger(версия Debian 7.12-6);
- Valgrind(версия 3.13.0).

Linux

3.1 Strace

Strace - это утилита, которая отслеживает системные вызовы, которые представляют собой механизм трансляции, обеспечивающий интерфейс между процессом и операционной системой (ядром). Эти вызовы могут быть перехвачены и прочитаны. Это позволяет лучше понять, что процесс пытается сделать в заданное время.[1]

3.1.1 Рассматриваемая программа

В качестве основной программы для тестирования был выбрана программа по генерации и обработке сигналов(SIGFPE и SIGSEGV) из лабораторной по обработке исключений.

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <signal.h>
5  #include <cstring>
6  #include <iostream>
7
8  using namespace std;
9
10 void GenerateFPException() {
11     int a = 1;
12     int b = a / (a - 1);
13 }
14 void GenerateSEGException() {
15     int* a = NULL;
16     *a = 1;
17 }
18
19 void SIGNAL_handler(int signum, siginfo_t *info, void *context){
20     switch(signum){
21         case SIGFPE:
22             printf("Received signal FP %d\n", signum);
23             break;
24         case SIGSEGV:
25             printf("Received signal SEG %d\n", signum);
26             break;
27     }
28     cout << "Signal number: " << info->si_signo << endl;
29     cout << "Errno: " << info->si_errno << endl;
30     cout << "Signal code: " << info->si_code << endl;
31     cout << "PID: " << info->si_pid << endl;
32     cout << "UID: " << info->si_uid << endl;
33     cout << "Status: " << info->si_status << endl;
```

```
34      cout << "Signal address: " << info->si_addr << endl;
35      cout << "Signal event: " << info->si_band << endl;
36      cout << "File descriptor: " << info->si_fd << endl;
37      exit(EXIT_FAILURE);
38  }
39
40  int main(int argc, char *argv[]){
41      struct sigaction NEW_action;
42
43  /* Инициализация структуры обработчиком */
44      sigemptyset(&NEW_action.sa_mask);
45      NEW_action.sa_flags = SA_SIGINFO;
46      NEW_action.sa_sigaction = &SIGNAL_handler;
47
48      if (argc == 2) {
49          if (strcmp(argv[1], "-FP") == 0 && sigaction(SIGFPE, &NEW_action, NULL)
    ↪  != -1)
50              GenerateFPException();
51          else if (strcmp(argv[1], "-SEG") == 0 && sigaction(SIGSEGV, &NEW_action
    ↪ , NULL) != -1) {
52              GenerateSEGException();
53          }
54      }
55   return 0;
56  }
```

Листинг 3.1: testProgram.cpp

3.1.2 Запуск без ключей

Для запуска утилиты, в консоли необходимо ввести:

strace <ключи strace> **./programName** <ключи programName>

В общем случае ключи strace и программы для трассировки являются опциональными.

```
1  root@kali:~/Desktop/trace# strace ./testProgram.o -FP
2  execve("./testProgram.o", ["./testProgram.o", "-FP"], [/* 46 vars */]) = 0
3  brk(NULL)                               = 0x55bf1c965000
4  access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
5  mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
       ↪ x7fbf8c65f000
6  access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
7  open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
8  fstat(3, {st_mode=S_IFREG|0644, st_size=128097, ...}) = 0
9  mmap(NULL, 128097, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fbf8c63f000
10 close(3)                                = 0
11 access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
12 open("/usr/lib/x86_64-linux-gnu/libstdc++.so.6", O_RDONLY|O_CLOEXEC) = 3
13 read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220\300\10\0\0\0\0\0"..., 832) =
       ↪  832
14 fstat(3, {st_mode=S_IFREG|0644, st_size=1557936, ...}) = 0
15 mmap(NULL, 3665920, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0
       ↪ x7fbf8c0c0000
16 mprotect(0x7fbf8c230000, 2097152, PROT_NONE) = 0
17 mmap(0x7fbf8c430000, 49152, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
       ↪ 3, 0x170000) = 0x7fbf8c430000
18 mmap(0x7fbf8c43c000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,
       ↪ -1, 0) = 0x7fbf8c43c000
19 close(3)                                = 0
20 access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
21 open("/lib/x86_64-linux-gnu/libm.so.6", O_RDONLY|O_CLOEXEC) = 3
22 read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\200V\0\0\0\0\0\0"..., 832) = 832
23 fstat(3, {st_mode=S_IFREG|0644, st_size=1063328, ...}) = 0
```

```
mmap(NULL, 3158248, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0
    ↪ x7fbf8bdbc000
mprotect(0x7fbf8bebf000, 2093056, PROT_NONE) = 0
mmap(0x7fbf8c0be000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
    ↪  0x102000) = 0x7fbf8c0be000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libgcc_s.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\300*\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=92520, ...}) = 0
mmap(NULL, 2188336, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0
    ↪ x7fbf8bba5000
mprotect(0x7fbf8bbbb000, 2093056, PROT_NONE) = 0
mmap(0x7fbf8bdba000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
    ↪  0x15000) = 0x7fbf8bdba000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\4\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1681176, ...}) = 0
mmap(NULL, 3787104, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0
    ↪ x7fbf8b808000
mprotect(0x7fbf8b99b000, 2097152, PROT_NONE) = 0
mmap(0x7fbf8bb9b000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
    ↪ 3, 0x193000) = 0x7fbf8bb9b000
mmap(0x7fbf8bba1000, 14688, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,
    ↪ -1, 0) = 0x7fbf8bba1000
close(3)                                = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
    ↪ x7fbf8c63d000
arch_prctl(ARCH_SET_FS, 0x7fbf8c63e140) = 0
mprotect(0x7fbf8bb9b000, 16384, PROT_READ) = 0
mprotect(0x7fbf8bdba000, 4096, PROT_READ) = 0
mprotect(0x7fbf8c0be000, 4096, PROT_READ) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
    ↪ x7fbf8c63b000
mprotect(0x7fbf8c430000, 40960, PROT_READ) = 0
mprotect(0x55bf1be5d000, 4096, PROT_READ) = 0
mprotect(0x7fbf8c662000, 4096, PROT_READ) = 0
munmap(0x7fbf8c63f000, 128097)          = 0
brk(NULL)                               = 0x55bf1c965000
brk(0x55bf1c997000)                     = 0x55bf1c997000
rt_sigaction(SIGFPE, {sa_handler=0x55bf1bc5cbb0, sa_mask=[], sa_flags=SA_RESTORER|
    ↪ SA_SIGINFO, sa_restorer=0x7fbf8b83b060}, NULL, 8) = 0
--- SIGFPE {si_signo=SIGFPE, si_code=FPE_INTDIV, si_addr=0x55bf1bc5cb8f} ---
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
write(1, "Received signal FP 8\n", 21Received signal FP 8
)  = 21
write(1, "Signal number: 8\n", 17Signal number: 8
)      = 17
write(1, "Errno: 0\n", 9Errno: 0
)              = 9
write(1, "Signal code: 1\n", 15Signal code: 1
)        = 15
write(1, "PID: 465947535\n", 15PID: 465947535
)        = 15
write(1, "UID: 21951\n", 11UID: 21951
)           = 11
write(1, "Status: -1954443208\n", 20Status: -1954443208
)   = 20
write(1, "Signal address: 0x55bf1bc5cb8f\n", 31Signal address: 0x55bf1bc5cb8f
) = 31
write(1, "Signal event: 94279293062031\n", 29Signal event: 94279293062031
) = 29
write(1, "File descriptor: -1954443208\n", 29File descriptor: -1954443208
) = 29
exit_group(1)                           = ?
+++ exited with 1 +++
```

Листинг 3.2: Лог strace

Лог выведенный strace весьма обширный, каждая его строчка это некоторая, вызванная программой системная функция с переданными параметрами. Для того чтобы по-

смотреть лишь список этих функций, необходимо запустить strace с ключом **c**.

Для более детального понимания, разберем одну из строчек вывода.

```
write(1, "Signal number: 8\n", 17Signal number: 8
)       = 17
```

Листинг 3.3: Одна из строчек лога strace

В данной строчке указано, что была вызвана системная функция **write**[2]. Рассмотрим её прототип.

```
ssize_t write(int fd, const void *buf, size_t count);
```

Листинг 3.4: Прототип write

- **fd** - файловый дескриптор, куда необходимо записать информацию;
- **buf** - указатель на текст для вывода;
- **count** - количество байт(символов) для записи.

Целью данной функции является запись **count** байт(символов) из **buf** в файловый дескриптор **fd**.

Также стоит отметить, что размер символа **\n** составляет 1 байт.

3.1.3 Ключ -c

Данный ключ позволяют вывести статистику исполнения программы, в частности:

- Процент времени затраченный на вызов функции;
- Временные затраты исполнения программы;
- Количество вызовов функций;
- Название функции.

```
root@kali:~/Desktop/trace# strace -c ./testProgram.o -FP
Received signal FP 8
Signal number: 8
Errno: 0
Signal code: 1
PID: 495569807
UID: 21865
Status: 479434334
Signal address: 0x55691d89cb8f
Signal event: 93909955496847
File descriptor: 479434334
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 50.77    0.000396          40        10           write
 14.87    0.000116          12        10           mprotect
  8.46    0.000066           5        14           mmap
  6.54    0.000051          10         5           close
  6.54    0.000051          17         3           brk
  2.69    0.000021           4         6           fstat
  2.69    0.000021          21         1           munmap
  1.67    0.000013           3         5           open
```

```
  1.67    0.000013           2         6           6 access
  1.41    0.000011           3         4             read
  1.41    0.000011          11         1             rt_sigaction
  1.28    0.000010          10         1             arch_prctl
  0.00    0.000000           0         1             execve
------ ----------- ----------- --------- --------- ----------------
100.00    0.000780                    67           6 total
root@kali:~/Desktop/trace#
```

Листинг 3.5: Лог со статистикой strace

Как видно из лога статистики, половину времени исполнения занял вывод информации в консоль, что справедливо из-за его обилия.

Далее идут вызовы open, mmap, read - открытие, отображение в память и чтение файла соответственно.

Системный вызов	**Описание**
write	Запись информации в дескриптор файла.
fstat	Получение информации о файле, по его дескриптору.
mprotect	Защита области памяти.
close	Закрытие потока работы с файлом.
access	Проверка прав пользователя на доступ к файлу.
munmap	Удаление интервалов адресов из адресного пространства.
arch_prctl	Установка архитектурно-специфичного состояния.
execve	Запуск программы.

3.1.4 Ключ -e

Данный ключ позволяет фильтровать, какие именно вызовы необходимо зафиксировать. По умолчанию значение стоит как **all**, что означает вывод всех вызовов.

Произведем трассировку вызовов связанных только с сигналом.

```
root@kali:~/Desktop/trace# strace -e trace=\signal ./testProgram.o -FP
rt_sigaction(SIGFPE, {sa_handler=0x562d472c4bb0, sa_mask=[], sa_flags=
    ↪ SA_RESTORER|SA_SIGINFO, sa_restorer=0x7f879d9bf060}, NULL, 8) = 0
--- SIGFPE {si_signo=SIGFPE, si_code=FPE_INTDIV, si_addr=0x562d472c4b8f} ---
Received signal FP 8
Signal number: 8
Errno: 0
Signal code: 1
PID: 1194085263
UID: 22061
Status: -1638087381
Signal address: 0x562d472c4b8f
Signal event: 94752467602319
File descriptor: -1638087381
+++ exited with 1 +++
```

Листинг 3.6: Лог только с сигналом

rt_sigaction[3] в данном случае вызывает функция **sigaction**, которая изменят выполняемое процессором действие при получении определенного сигнала.

Так же произведем трассировку вызовов команды write.

```
root@kali:~/Desktop/trace# strace -e trace=\write ./testProgram.o -FP
```

```
2  --- SIGFPE {si_signo=SIGFPE, si_code=FPE_INTDIV, si_addr=0x5598e76e6b8f} ---
3  write(1, "Received signal FP 8\n", 21Received signal FP 8
4  )  = 21
5  write(1, "Signal number: 8\n", 17Signal number: 8
6  )       = 17
7  write(1, "Errno: 0\n", 9Errno: 0
8  )               = 9
9  write(1, "Signal code: 1\n", 15Signal code: 1
10 )         = 15
11 write(1, "PID: -412193905\n", 16PID: -412193905
12 )        = 16
13 write(1, "UID: 21912\n", 11UID: 21912
14 )            = 11
15 write(1, "Status: 177664056\n", 18Status: 177664056
16 )      = 18
17 write(1, "Signal address: 0x5598e76e6b8f\n", 31Signal address: 0x5598e76e6b8f
18 ) = 31
19 write(1, "Signal event: 94115206163343\n", 29Signal event: 94115206163343
20 ) = 29
21 write(1, "File descriptor: 177664056\n", 27File descriptor: 177664056
22 ) = 27
23 +++ exited with 1 +++
```

Листинг 3.7: Лог только с функцией write

Как и ожидалось, в обоих вариантах лог был отфильтрован.

3.2 Ptrace

Ptrace[4] — системный вызов в некоторых unix-подобных системах (в том числе в Linux, FreeBSD, Max OS X), который позволяет трассировать или отлаживать выбранный процесс. Можно сказать, что ptrace дает полный контроль над процессом: можно изменять ход выполнения программы, смотреть и изменять значения в памяти или состояния регистров.

```
1 #include <sys/ptrace.h>
2 long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

Листинг 3.8: Прототип ptrace

1. **request** — это действие, которое необходимо осуществить, например PTRACE_CONT, PTRACE_PEEKTEXT;
2. **pid** — индентификатор трассируемого процесса;
3. **addr** и **data** зависят от **request**'а.

Начать трассировку можно двумя способами: приаттачиться к уже запущенному процессу (PTRACE_ATTACH), либо запустить его самому с помощью PTRACE_TRACEME.

Для управления трассировкой можно использовать следующие аргументы:

- **PTRACE_SINGLESTEP** — пошаговое выполнение программы, управление будет передаваться после выполнения каждой инструкции; такая трассировка достаточна медленна
- **PTRACE_SYSCALL** — продолжить выполнение программы до входа или выхода из системного вызова

- **PTRACE_CONT** — просто продолжить выполнение программы

3.2.1 Просмотр системных вызовов

Напишем программу для вывода списка системных вызовов, используемых программой (простой аналог утилиты strace).

Для начала необходимо сделать fork — родительский процесс будет отлаживать дочерний:

```
1  int main(int argc, char *argv[]) {
2    pid_t pid = fork();
3    if (pid)
4      parent(pid);
5    else
6      child();
7    return 0;
8  }
```

В дочернем процессе начинаем трассировку с PTRACE_TRACEME и запускаем нужную программу:

```
1  void child() {
2    ptrace(PTRACE_TRACEME, 0, 0, 0);
3    execl("/bin/echo", "/bin/echo", "Hello, world!", NULL);
4    perror("execl");
5  }
```

При выполнении **execl**[5] трассируемый процесс остановится, передав свое новое состояние родительскому. Поэтому родительский процесс сначала должен подождать запуска программы с помощью **waitpid**[6] (можно просто **wait**, так как дочерний процесс всего один):

```
1    int status;
2    waitpid(pid, &status, 0);
```

Чтобы как-то различать системные вызовы и другие остановки (например SIGTRAP), предусмотрен специальный параметр PTRACE_O_TRACESYSGOOD — при остановке на системном вызове родительский процесс получит в статусе SIGTRAP | 0x80:

```
1  ptrace(PTRACE_SETOPTIONS, pid, 0, PTRACE_O_TRACESYSGOOD);
```

Теперь можно в цикле выполнять PTRACE_SYSCALL до выхода из программы, и смотреть значение регистра rax для определения номера системного вызова. Для этого используем PTRACE_GETREGS. Следует отметить, что регистр rax в момент остановки заменен, и поэтому необходимо использовать сохраненный state.orig_rax:

```
1    while (!WIFEXITED(status)) {
2
3      struct user_regs_struct state;
4
5      ptrace(PTRACE_SYSCALL, pid, 0, 0);
6      waitpid(pid, &status, 0);
7
8      if (WIFSTOPPED(status) && WSTOPSIG(status) & 0x80) {
9        ptrace(PTRACE_GETREGS, pid, 0, &state);
10       printf("SYSCALL %d at %08lx\n", state.orig_rax, state.rip);
11     }
12   }
```

WIFSTOPPED(status) возвращает истинное значение, если дочерний процесс, из-за которого функция вернула управление, в настоящий момент остановлен.

WSTOPSIG(status) возвращает номер сигнала, из-за которого дочерний процесс был остановлен.

Итоговый код выглядит следующим образом:

```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/user.h>
#include <sys/reg.h>
#include <stdio.h>

void parent(pid_t pid ){
    int status;
    waitpid(pid, &status, 0);//ожидание запуска программы

    ptrace(PTRACE_SETOPTIONS, pid, 0, PTRACE_O_TRACESYSGOOD);//сообщать о
    ↪ системных вызовах

    // пока дочерний процесс не закончил работу
    while (!WIFEXITED(status)) {
        struct user_regs_struct state;
        // в цикле выполняем этот вызов для просмотра всех системных вызовов
        ptrace(PTRACE_SYSCALL, pid, 0, 0);
        waitpid(pid, &status, 0);

        if (WIFSTOPPED(status) && WSTOPSIG(status) & 0x80) {
            //получаем информацию о вызове
            ptrace(PTRACE_GETREGS, pid, 0, &state);
            printf("SYSCALL %d at %08lx\n", state.orig_rax, state.rip);
    }
  }
}

void child() {
  ptrace(PTRACE_TRACEME, 0, 0, 0);//запускаем трассировку
  execl("/bin/echo", "/bin/echo", "Hello, world!", NULL);//запускаем выбранную
   ↪ программу
  perror("execl");
}

int main(int argc, char *argv[]) {
  pid_t pid = fork();
  if (pid)
    parent(pid);
  else
    child();
  return 0;
}
```

Листинг 3.9: testProgram2.cpp

```
root@kali:~/Desktop/trace# g++ testProgram2.cpp –o testProgram2.o
root@kali:~/Desktop/trace# ./testProgram2.o
SYSCALL 12 at 7f775dae18d9
SYSCALL 12 at 7f775dae18d9
SYSCALL 21 at 7f775dae2607
SYSCALL 21 at 7f775dae2607
```

```
7  ...
8  SYSCALL 9 at 7f775d8103fa
9  SYSCALL 9 at 7f775d8103fa
10 SYSCALL 3 at 7f775d75678d
11 SYSCALL 3 at 7f775d75678d
12 SYSCALL 2 at 7f775d7566ec
13 SYSCALL 2 at 7f775d7566ec
14 SYSCALL 5 at 7f775d807092
15 SYSCALL 5 at 7f775d807092
16 SYSCALL 9 at 7f775d8103fa
17 SYSCALL 9 at 7f775d8103fa
18 SYSCALL 3 at 7f775d75678d
19 SYSCALL 3 at 7f775d75678d
20 SYSCALL 5 at 7f775d807092
21 SYSCALL 5 at 7f775d807092
22 SYSCALL 1 at 7f775d807720
23 Hello, world!
24 SYSCALL 1 at 7f775d807720
25 SYSCALL 3 at 7f775d79da4b
26 SYSCALL 3 at 7f775d79da4b
27 SYSCALL 3 at 7f775d79da4b
28 SYSCALL 3 at 7f775d79da4b
29 SYSCALL 231 at 7f775d7e4608
```

Листинг 3.10: Запуск testProgram2.cpp

Как видно из лога, после системного вызова 1, был напечатан **Hello, world!**

3.2.2 Перехват системного вызова

Попробуем теперь перехватить вызов, и изменить выводимый текст. Системный вызов write[2] выглядит так:

```
1 ssize_t write(int fd, const void *buf, size_t count);
```

Листинг 3.11: Прототип write

- **fd** - файловый дескриптор(номер), куда необходимо записать информацию;
- **buf** - указатель на текст для вывода;
- **count** - количество байт(символов) для записи.

Для подмены текста используем PTRACE_POKETEXT:

```
1 if (state.orig_rax == 1) {
2     char * text = (char *)state.rsi;
3     ptrace(PTRACE_POKETEXT, pid, (void*)(text+7), 0x696e6544); //Deni
4     ptrace(PTRACE_POKETEXT, pid, (void*)(text+11), 0x000a2173); //s!\n
5 }
```

Текст переписывается в виде байт символов **0x696e6544**[7], что переводится как **Deni**, то есть
0x44 = D
0x65 = e
0x6e = n
0x69 = i
Итоговый измененный код выглядит следующим образом:

```c
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/user.h>
#include <sys/reg.h>
#include <stdio.h>

void parent(pid_t pid ){
    int status;
    waitpid(pid, &status, 0);//ожидание запуска программы

    ptrace(PTRACE_SETOPTIONS, pid, 0, PTRACE_O_TRACESYSGOOD);//сообзать о
    ↪ системных вызовах

    // пока дочерний процесс не закончил работу
    while (!WIFEXITED(status)) {
        struct user_regs_struct state;
        // в цикле выполняем этот вызов для просмотра всех системных вызовов
        ptrace(PTRACE_SYSCALL, pid, 0, 0);
        waitpid(pid, &status, 0);

        if (WIFSTOPPED(status) && WSTOPSIG(status) & 0x80) {

            //получаем информацию о вызове
            ptrace(PTRACE_GETREGS, pid, 0, &state);

            if (state.orig_rax == 1) {
                char * text = (char *)state.rsi;
                ptrace(PTRACE_POKETEXT, pid, (void*)(text+7), 0x696e6544); //
    ↪ Deni
                ptrace(PTRACE_POKETEXT, pid, (void*)(text+11), 0x000a2173); //s
    ↪ !\n
            }

            printf("SYSCALL %d at %08lx\n", state.orig_rax, state.rip);
    }
  }
}

void child() {
  ptrace(PTRACE_TRACEME, 0, 0, 0);//запускаем трассировку
  execl("/bin/echo", "/bin/echo", "Hello, world!", NULL);//запускаем выбранную
    ↪ программу
  perror("execl");
}

int main(int argc, char *argv[]) {
  pid_t pid = fork();
  if (pid)
    parent(pid);
  else
    child();
  return 0;
}
```

Листинг 3.12: testProgram2.cpp

Запустим программу.

```
root@kali:~/Desktop/trace# g++ testProgram2.cpp –o testProgram2.o
```

```
root@kali:~/Desktop/trace# ./testProgram2.o
SYSCALL 12 at 7fc75725a8d9
SYSCALL 12 at 7fc75725a8d9
SYSCALL 21 at 7fc75725b607
SYSCALL 21 at 7fc75725b607
...
SYSCALL 3 at 7fc756ecf78d
SYSCALL 2 at 7fc756ecf6ec
SYSCALL 2 at 7fc756ecf6ec
SYSCALL 5 at 7fc756f80092
SYSCALL 5 at 7fc756f80092
SYSCALL 9 at 7fc756f893fa
SYSCALL 9 at 7fc756f893fa
SYSCALL 3 at 7fc756ecf78d
SYSCALL 3 at 7fc756ecf78d
SYSCALL 5 at 7fc756f80092
SYSCALL 5 at 7fc756f80092
SYSCALL 1 at 7fc756f80720
Hello, Denis!
SYSCALL 1 at 7fc756f80720
SYSCALL 3 at 7fc756f16a4b
SYSCALL 3 at 7fc756f16a4b
SYSCALL 3 at 7fc756f16a4b
SYSCALL 3 at 7fc756f16a4b
SYSCALL 231 at 7fc756f5d608
```

Листинг 3.13: Запуск testProgram2.cpp

Таким образом, был перехвачен системный вызов write в программе /bin/echo для вывода своего текста.

3.3 Syslog

Syslog[8] — стандарт отправки сообщений о происходящих в системе событиях (логов), использующийся в компьютерных сетях, работающих по протоколу IP.

Протокол syslog прост: отправитель посылает короткое текстовое сообщение, размером меньше 1024 байт получателю сообщения. Получатель при этом носит имя «syslogd», «syslog daemon», либо же, «syslog server». Сообщения могут отправляться как по UDP, так и по TCP. Как правило, такое сообщение отсылается в открытом виде.

Syslog используется для удобства администрирования и обеспечения информационной безопасности. Он реализован под множество платформ и используется в множестве устройств. Поэтому, использование syslog позволяет обеспечить сбор информации с разных мест и хранение её в едином репозитории.

В UNIX-системах для использования syslog необходимо подключить файл syslog.h. Пример использования:

```
#include <syslog.h>

openlog("имя программы", LOG_PID, LOG_USER);
syslog(LOG_NOTICE, "Can not open file "%s" for writing.",filename);
closelog();
```

Листинг 3.14: Пример использования syslog

openlog() устанавливает связь с программой, ведущей системный журнал. Первый аргумент добавляется к каждому сообщению и обычно представляет собой название

программы. Второй аргумент – option указывает флаг управляющий работой openlog() и соответствующих вызовов syslog(). В данном примере у option установлен LOG_PID. Это значит, что к каждому сообщению будет добавляться pid нашей программы.

syslog() создает сообщение и передает его демону. Первым аргументом syslog() получает уровень критичности сообщения. Всего существует 8 уровней: самый нижний LOG_DEBUG, самый верхний LOG_EMERG. Остальные аргументы передаются так же, как в printf.

closelog() закрывает описатель, используемый для записи данных в журнал.

Напишем программу, которая запишет в системный журнал некоторые записи.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <syslog.h>
4
5  int main(){
6
7      openlog("test",LOG_PID,LOG_USER);
8      for (int i=0;i<5;i++){
9          syslog(LOG_INFO,"myTestMessage number %d ",i);
10     };
11     closelog();
12     return 0;
13 };
```

Листинг 3.15: testProgram3.cpp

Скомпилируем и запустим.

```
1 root@kali:~/Desktop/trace# g++ testProgram3.cpp –o testProgram3.o
2 root@kali:~/Desktop/trace# ./testProgram3.o
3 root@kali:~/Desktop/trace# less /var/log/messages | grep myTest
4 Dec 10 12:30:18 kali test[10461]: myTestMessage number 0
5 Dec 10 12:30:18 kali test[10461]: myTestMessage number 1
6 Dec 10 12:30:18 kali test[10461]: myTestMessage number 2
7 Dec 10 12:30:18 kali test[10461]: myTestMessage number 3
8 Dec 10 12:30:18 kali test[10461]: myTestMessage number 4
```

Листинг 3.16: Запуск testProgram3.cpp

Как и ожидалось, после запуска программы, в системный журнал **/var/log/messages** были добавлены записи программы, с указанием ее PID.

3.4 Backtrace

Backtrace[9] – это системный вызов, который позволяет получать трассировку вызовов в программе. Для этого напишем программу, в которой используется несколько пустых вложенных вызовов.

```
1 #include <execinfo.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 /* Получаем информацию о стеке вызовов функций и печатаем ее */
6 void print_trace (void) {
7         void *array[10];
8         size_t size;
9         char **strings;
```

```
10          size_t i;
11          size = backtrace (array, 10);
12          strings = backtrace_symbols (array, size);
13          printf ("Catched %zd stack items.\n", size);
14          for (i = 0; i < size; i++)
15                  printf ("%s\n", strings[i]);
16          free (strings);
17  }
18
19  // функции, для увеличения глубины стека
20  void subFunction_3 (void) {
21          print_trace ();
22  }
23
24  void subFunction_2 (void) {
25          subFunction_3 ();
26  }
27
28  void subFunction_1 (void) {
29          subFunction_2 ();
30  }
31
32  int main (void) {
33          subFunction_1 ();
34          return 0;
35  }
```

Листинг 3.17: testProgram4.cpp

При компиляции был добавлен ключ **rdynamic**, для того чтобы linker включал всю информацию о символах и как итог в выводе присутствовало название функции.

```
1  root@kali:~/Desktop/trace# g++ testProgram4.cpp –o testProgram4.o –rdynamic
2  root@kali:~/Desktop/trace# ./testProgram4.o
3  Catched 7 stack items.
4  ./testProgram4.o(_Z11print_tracev+0x19) [0x55fb33b3ba43]
5  ./testProgram4.o(_Z13subFunction_3v+0x9) [0x55fb33b3bac8]
6  ./testProgram4.o(_Z13subFunction_2v+0x9) [0x55fb33b3bad4]
7  ./testProgram4.o(_Z13subFunction_1v+0x9) [0x55fb33b3bae0]
8  ./testProgram4.o(main+0x9) [0x55fb33b3baec]
9  /lib/x86_64–linux–gnu/libc.so.6(__libc_start_main+0xf1) [0x7f20818c42e1]
10 ./testProgram4.o(_start+0x2a) [0x55fb33b3b94a]
```

Листинг 3.18: Компилируем и запускаем testProgram4.cpp

Как и ожидалось, программа зафиксировала выполненный стек вызовов.

3.5 GDB

GNU Debugger (GDB)[10] представляет собой переносимый отладчик, предназначенный для отладки программ на различных языках программирования во многих UNIX-подобных системах. GDB располагает большим количеством различных средств для слежения и контроля за выполнением программ, представленных в виде отдельных команд. Далее приведены используемые в нем функции и их описание:

1. run – запускает программу на выполнение;
2. break (b) – устанавливает точку останова; параметром может быть номер строки или название функции;

3. rbreak - устанавливает точку останова на функциях, название которых соответствует заданному регулярному выражению;
4. info breakpoints – выводит список всех имеющихся точек останова;
5. clear – удаляет все точки останова на текущем уровне стека;
6. delete – удаляет точку останова или контрольное выражение;
7. list – выводит исходный код; в качестве параметра передаются название файла исходного кода, затем, через двоеточие, номер начальной и конечной строки;
8. backtrace (bt) – выводит стек вызовов функций в текущий момент выполнения программы;
9. finish – выполняет программу до выхода из текущей функции; отображает возвращаемое значение,если такое имеется;
10. continue (c) – продолжает выполнение программы от текущей точки до конца;
11. next – пошаговое выполнение программы, но, в отличие от команды step, не выполняет пошагово вызываемые функции;
12. step (s) – пошаговое выполнение программы.

Для экспериментов возьмем программу из лабораторной по исключениям, в которой при обработке одного из исключений вызывается еще одно исключение.

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <signal.h>
5  #include <cstring>
6
7  void GenerateFPException() {
8          int a = 1;
9          int b = a / (a - 1);
10 }
11 void GenerateSEGException() {
12         int* a = NULL;
13         *a = 1;
14 }
15
16 void FP_handler(int signum){
17     printf("Received signal FP %d\n", signum);
18     GenerateSEGException();
19     printf("Exit in FP signal\n");
20     exit(EXIT_FAILURE);
21 }
22
23 void SEG_handler(int signum){
24     printf("Received signal SEG %d\n", signum);
25     printf("Exit in SEG signal\n");
26     exit(EXIT_FAILURE);
27 }
28
29
30 int main(int argc, char *argv[]){
31         if (signal(SIGFPE, FP_handler) == SIG_ERR)
32                 printf("Cant set SIGFP handler\n");
```

```
33      if (signal(SIGSEGV, SEG_handler) == SIG_ERR)
34          printf("Cant set SIGSEGV handler\n");
35
36          if (argc == 2) {
37                  if (strcmp(argv[1], "-FP") == 0)
38                          GenerateFPException();
39                  else if (strcmp(argv[1], "-SEG") == 0) {
40                          GenerateSEGException();
41                  }
42          }
43   return 0;
44  }
```

Листинг 3.19: testProgram5.cpp

Скомпилируем и запустим программу.

```
1  root@kali:~/Desktop/trace# g++ testProgram5.cpp -o testProgram5.o -g
2  root@kali:~/Desktop/trace# ./testProgram5.o -FP
3  Received signal FP 8
4  Received signal SEG 11
5  Exit in SEG signal
```

Листинг 3.20: Запуск testProgram5.cpp

Было сгенерировано исключение FP(ошибка деления), для которого был вызван соответствующий обработчик, в котором было вызвано исключение SEG(ошибка доступа к памяти), для которого был вызван также обработчик, который и закончил работу программы. Рассмотрим какую информацию можно получить используя gdb.

```
1  root@kali:~/Desktop/trace# gdb testProgram5.o
2  (gdb) run -FP
3  Starting program: /root/Desktop/trace/testProgram5.o -FP
4
5  Program received signal SIGFPE, Arithmetic exception.
6  0x000055555555479f in GenerateFPException () at testProgram5.cpp:9
7  9           int b = a / (a - 1);
8  (gdb) bt
9  #0  0x000055555555479f in GenerateFPException () at testProgram5.cpp:9
10 #1  0x00005555555548bb in main (argc=2, argv=0x7fffffffe1f8) at testProgram5.
       ↪ cpp:38
11 (gdb) c
12 Continuing.
13 Received signal FP 8
14
15 Program received signal SIGSEGV, Segmentation fault.
16 0x00005555555547b7 in GenerateSEGException () at testProgram5.cpp:13
17 13          *a = 1;
18 (gdb) bt
19 #0  0x00005555555547b7 in GenerateSEGException () at testProgram5.cpp:13
20 #1  0x00005555555547e6 in FP_handler (signum=8) at testProgram5.cpp:18
21 #2  <signal handler called>
22 #3  0x000055555555479f in GenerateFPException () at testProgram5.cpp:9
23 #4  0x00005555555548bb in main (argc=2, argv=0x7fffffffe1f8) at testProgram5.
       ↪ cpp:38
24 (gdb) s
25 SEG_handler (signum=0) at testProgram5.cpp:23
26 23  void SEG_handler(int signum){
27 (gdb) bt
28 #0  SEG_handler (signum=0) at testProgram5.cpp:23
29 #1  <signal handler called>
```

```
#2  0x00005555555547b7 in GenerateSEGException () at testProgram5.cpp:13
#3  0x00005555555547e6 in FP_handler (signum=8) at testProgram5.cpp:18
#4  <signal handler called>
#5  0x000055555555479f in GenerateFPException () at testProgram5.cpp:9
#6  0x00005555555548bb in main (argc=2, argv=0x7fffffffe1f8) at testProgram5.
    ↪ cpp:38
(gdb) c
Continuing.
Received signal SEG 11
Exit in SEG signal
[Inferior 1 (process 10787) exited with code 01]
```

Листинг 3.21: Отладка testProgram5.cpp

В ходе отладки было выявлено, что после генерации сигнала об ошибке в ходе арифметической операции был вызван соответствующий обработчик, который сгенерировал сигнал SIGSEGV. Далее, системой без какого-либо ожидания завершения предыдущего обработчика было передано управление обработчику сигнала SIGSEGV, который завершил работу программы.

По ходу отладки программы, была вызвана функция **bt** которая позволила показать стек вызовов, в котором был зафиксирован стек вызовов функций-генератором исключений и их обработчиков.

```
root@kali:~/Desktop/trace# gdb testProgram5.o
(gdb) break 31
Breakpoint 1 at 0x842: file testProgram5.cpp, line 31.
(gdb) break 36
Breakpoint 2 at 0x892: file testProgram5.cpp, line 36.
(gdb) break 37
Breakpoint 3 at 0x898: file testProgram5.cpp, line 37.
(gdb) run –FP
Starting program: /root/Desktop/trace/testProgram5.o –FP

Breakpoint 1, main (argc=2, argv=0x7fffffffe1f8) at testProgram5.cpp:31
31          if (signal(SIGFPE, FP_handler) == SIG_ERR)
(gdb) c
Continuing.

Breakpoint 2, main (argc=2, argv=0x7fffffffe1f8) at testProgram5.cpp:36
36          if (argc == 2) {
(gdb) c
Continuing.

Breakpoint 3, main (argc=2, argv=0x7fffffffe1f8) at testProgram5.cpp:37
37                  if (strcmp(argv[1], "–FP") == 0)
(gdb) finish
"finish" not meaningful in the outermost frame.
(gdb) q
A debugging session is active.

    Inferior 1 [process 10833] will be killed.

Quit anyway? (y or n) y
```

Листинг 3.22: Отладка testProgram5.cpp

Были поставлены точки останова в определенных строках(31, 36, 37), отладчик корректно остановился в этих строках с выводом исходного кода. Далее с помощью функции **q(quit)** был произведен выход из отладки, перед которым gdb предупредил что текущий процесс отладки будет удален.

```
root@kali:~/Desktop/trace# gdb testProgram5.o
(gdb) break 36
Breakpoint 1 at 0x892: file testProgram5.cpp, line 36.
(gdb) run –FP
Starting program: /root/Desktop/trace/testProgram5.o –FP

Breakpoint 1, main (argc=2, argv=0x7fffffffe1f8) at testProgram5.cpp:36
36          if (argc == 2) {
(gdb) print argc
$1 = 2
(gdb) print argv
$2 = (char **) 0x7fffffffe1f8
(gdb) print argv[0]
$3 = 0x7fffffffe4ff "/root/Desktop/trace/testProgram5.o"
(gdb) print argv[1]
$4 = 0x7fffffffe522 "–FP"
(gdb) set argv[1]="–SEG"
(gdb) print argv[1]
$5 = 0x555555767c20 "–SEG"
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x00005555555547b7 in GenerateSEGException () at testProgram5.cpp:13
13          *a = 1;
(gdb) c
Continuing.
Received signal SEG 11
Exit in SEG signal
[Inferior 1 (process 10852) exited with code 01]
(gdb) q
```

Листинг 3.23: Отладка testProgram5.cpp

В логе отладки выше, была установлена точка остановки на строчке где сравнивается количество переменных переданных в функцию main. С помощью команды **print** было показано содержимое массива argv. Далее командой **set** была изменена переменная argv[1], которая отвечала за работу программы. И как итог, программа была запущена с флагом **-FP**, но в процессе отладки он был изменен на флаг **-SEG**, что изменило дальнейшие действия программы.

```
root@kali:~/Desktop/trace# gdb testProgram5.o
(gdb) list 10
5   #include <cstring>
6
7   void GenerateFPException() {
8           int a = 1;
9           int b = a / (a – 1);
10  }
11  void GenerateSEGException() {
12          int* a = NULL;
13          *a = 1;
14  }
(gdb) list main
25      printf("Exit in SEG signal\n");
26      exit(EXIT_FAILURE);
27  }
28
29
30  int main(int argc, char *argv[]){
```

```
20 31          if (signal(SIGFPE, FP_handler) == SIG_ERR)
21 32                  printf("Cant set SIGFP handler\n");
22 33      if (signal(SIGSEGV, SEG_handler) == SIG_ERR)
23 34          printf("Cant set SIGSEGV handler\n");
24
25 (gdb) q
```

Листинг 3.24: Отладка testProgram5.cpp

Также в отладке имеется возможность посмотреть исходный код определенных строк или функций. Для этого используется функция **list <param>**, где param номер строчки или функции, которая будет центрирована при выводе.

3.6 Valgrind

Valgrind[11] является многоцелевым инструментом профилирования кода и отладки памяти. Программа способна контролировать использование памяти, например, вызовы malloc и free (или new и delete в C++). Если используется неинициализированная память, запись за пределы концов массива, или не освобождение указателей, Valgrind может это обнаружить.

В состав пакета Valgrind входит множество инструментов (некоторые дополнительные инструменты не входят в его состав). Инструмент по умолчанию (и наиболее используемый) — **Memcheck**.

Проблемы, которые может обнаружить Memcheck, включают в себя:

- попытки использования не инициализированной памяти,
- чтение/запись в память после её освобождения,
- чтение/запись за границами выделенного блока,
- утечки памяти.

Для теста возьмем программу из листинга 3.20 и запустим valgrind с инструментов memcheck.

```
1  root@kali:~/Desktop/trace# valgrind --tool=memcheck --leak-check=yes ./
       ↪ testProgram5.o -FP
2  ==12177== Memcheck, a memory error detector
3  ==12177== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
4  ==12177== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
5  ==12177== Command: ./testProgram5.o -FP
6  ==12177==
7  Received signal FP 8
8  ==12177== Invalid write of size 4
9  ==12177==    at 0x1087B7: GenerateSEGException() (testProgram5.cpp:13)
10 ==12177==    by 0x1087E5: FP_handler(int) (testProgram5.cpp:18)
11 ==12177==    by 0x570505F: ??? (in /lib/x86_64-linux-gnu/libc-2.24.so)
12 ==12177==    by 0x10879A: GenerateFPException() (testProgram5.cpp:9)
13 ==12177==    by 0x1088BA: main (testProgram5.cpp:38)
14 ==12177==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
15 ==12177==
16 Received signal SEG 11
17 Exit in SEG signal
18 ==12177==
19 ==12177== HEAP SUMMARY:
20 ==12177==     in use at exit: 0 bytes in 0 blocks
```

```
==12177==   total heap usage: 2 allocs, 2 frees, 73,728 bytes allocated
==12177==
==12177== All heap blocks were freed -- no leaks are possible
==12177==
==12177== For counts of detected and suppressed errors, rerun with: -v
==12177== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Листинг 3.25: Проверка testProgram5.cpp

Каждая строчка лога начинается с числа, оно означает ID процесса во время выполнения программы. По логу можно заметить что неверное обращение к памяти было успешно зафиксировано, а также был выведен стек команд. Ошибку деления на ноль - утилита не обнаружила.

3.6.1 Неинициализированная память

Для теста была написана программа, в которой используется неинициализированная память.

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

int main(void){
    char *p;
    char c = *p;
    printf("\n [%c]\n",c);
    return 0;
}
```

Листинг 3.26: testProgram6.cpp

```
root@kali:~/Desktop/trace# g++ testProgram6.cpp -o testProgram6.o -g
root@kali:~/Desktop/trace# valgrind --tool=memcheck --leak-check=yes ./
    ↪ testProgram6.o
==12245== Memcheck, a memory error detector
==12245== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==12245== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==12245== Command: ./testProgram6.o
==12245==
==12245== Use of uninitialised value of size 8
==12245==    at 0x108676: main (testProgram6.cpp:6)
==12245==
==12245== Invalid read of size 1
==12245==    at 0x108676: main (testProgram6.cpp:6)
==12245==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==12245==
==12245==
==12245== Process terminating with default action of signal 11 (SIGSEGV)
==12245==  Access not within mapped region at address 0x0
==12245==    at 0x108676: main (testProgram6.cpp:6)
==12245==  If you believe this happened as a result of a stack
==12245==  overflow in your program's main thread (unlikely but
==12245==  possible), you can try to increase the size of the
==12245==  main thread stack using the --main-stacksize= flag.
==12245==  The main thread stack size used in this run was 8388608.
==12245==
==12245== HEAP SUMMARY:
==12245==     in use at exit: 0 bytes in 0 blocks
==12245==   total heap usage: 1 allocs, 1 frees, 72,704 bytes allocated
```

```
==12245==
==12245== All heap blocks were freed -- no leaks are possible
==12245==
==12245== For counts of detected and suppressed errors, rerun with: -v
==12245== Use --track-origins=yes to see where uninitialised values come from
==12245== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
Segmentation fault
```

Листинг 3.27: Проверка testProgram6.cpp

Утилита успешно сообщила о том что используется неинициализированная переменная. Также в логи зафиксировано то что программа закончила свою работу с сигналом **11 SIGSEGV** - ошибкой доступа к памяти.

3.6.2 Утечка памяти

Напишем программу, в которой происходит утечка памяти.

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

int main(void){
    char *p = malloc(1);
    *p = 'a';
    char c = *p;
    printf("\n [%c]\n",c);
    return 0;
}
```

Листинг 3.28: testProgram7.cpp

```
root@kali:~/Desktop/trace# valgrind --tool=memcheck --leak-check=yes ./
    ↪ testProgram7.o
==12335== Memcheck, a memory error detector
==12335== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==12335== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==12335== Command: ./testProgram7.o
==12335==

 [a]
==12335==
==12335== HEAP SUMMARY:
==12335==     in use at exit: 1 bytes in 1 blocks
==12335==   total heap usage: 3 allocs, 2 frees, 73,729 bytes allocated
==12335==
==12335== 1 bytes in 1 blocks are definitely lost in loss record 1 of 1
==12335==    at 0x4C2BBEF: malloc (vg_replace_malloc.c:299)
==12335==    by 0x1086CB: main (testProgram7.cpp:5)
==12335==
==12335== LEAK SUMMARY:
==12335==    definitely lost: 1 bytes in 1 blocks
==12335==    indirectly lost: 0 bytes in 0 blocks
==12335==      possibly lost: 0 bytes in 0 blocks
==12335==    still reachable: 0 bytes in 0 blocks
==12335==         suppressed: 0 bytes in 0 blocks
==12335==
==12335== For counts of detected and suppressed errors, rerun with: -v
==12335== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Листинг 3.29: Проверка testProgram7.cpp

Утилита зафиксировало что был выделен 1 байт памяти, но не был освобожден. Если добавить оператор **free** перед завершение программы, то утилита не зафиксирует некорректной работы с памятью.

Windows

4.1 Visual Studio

Отладчик **Visual Studio** позволяет наблюдать за поведением программы во время выполнения и искать таким образом логические ошибки. Отладчик работает со всеми языками программирования Visual Studio и библиотеками. С помощью отладчика можно прерывать выполнение для пошагового выполнения кода и оценки переменных в приложении.[12]

С его помощью возможно:

- Использовать точки останова;
- Просматривать информацию о производительности системы, стек вызовов, значения переменных;
- Пошаговое выполнение кода.

Напишем простую программу для отладки, и поставим точку останова в строчке 5.

```
1  #include <iostream>
2  using namespace std;
3
4  void multiplyFunction(int a, int b) {
5      cout << "a*b=" << a*b << endl;
6  }
7
8  int main()
9  {
10     int a = 5;
11     int b = 10;
12     cout << "a=" << a << ", b=" << b << endl;
13     multiplyFunction(a, b);
14     return 0;
15 }
```

Листинг 4.1: test1.cpp

Программа остановилась на указанной точке останова.

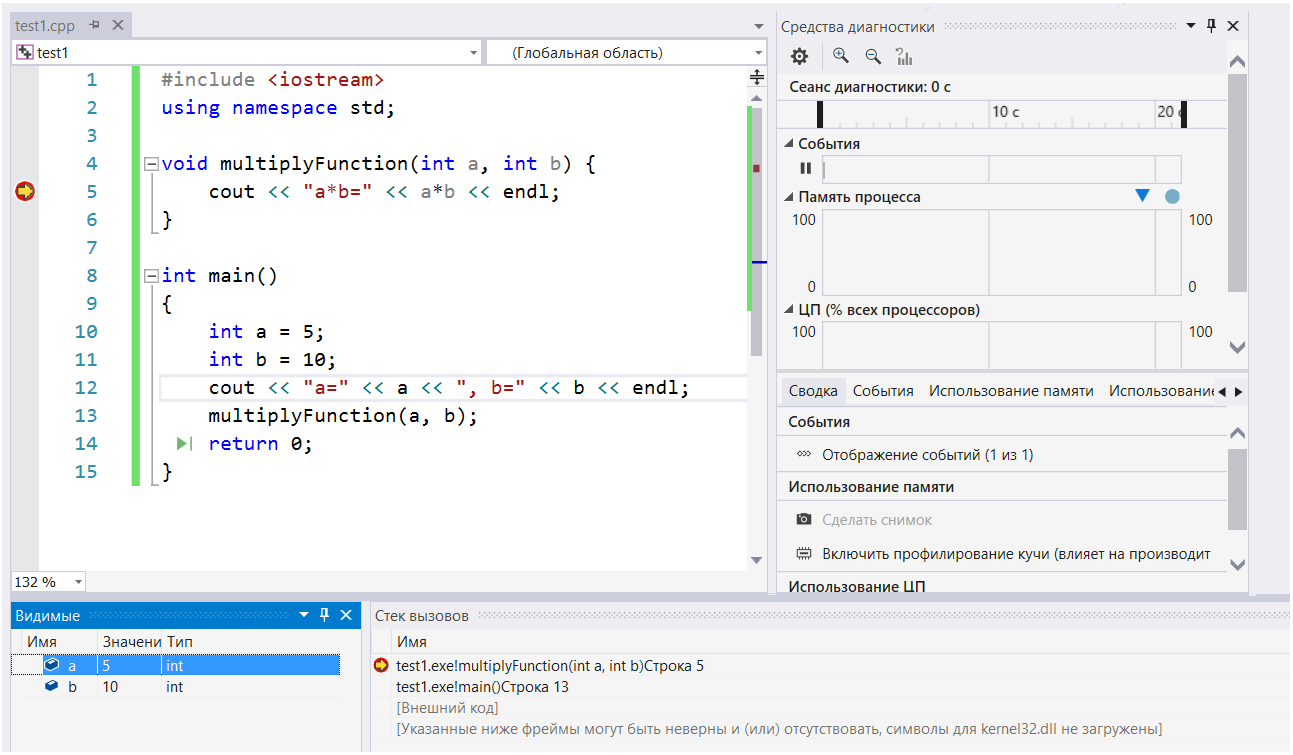

Рис. 4.1: Меню отладки Visual Studio

Изменим значение переменной **a** с 5 на 8.

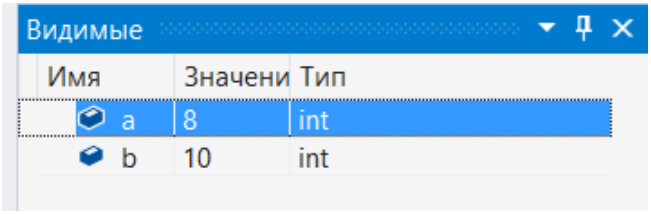

Рис. 4.2: Список переменных

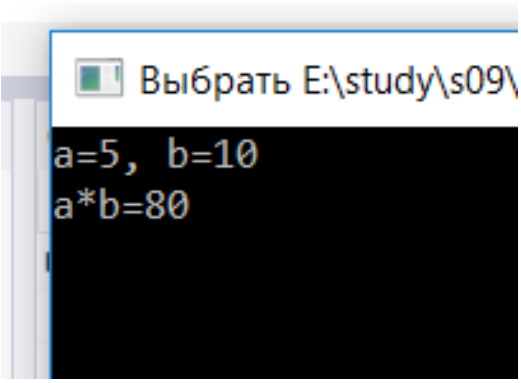

Рис. 4.3: Результат

Как итог в процессе отладке одна из переменных была изменена, что повлияло на итоговый результат выполнения программы.

4.2 API Monitor

Одним из инструментов для мониторинга вызовов к api windows является - **API Monitor**. Утилита бесплатная. Поддерживаются 32-бит и 64-бит системы. Позволяет установить мониторинг заданных API функций и интерфейсов. Показывает состояние параметров функции до и после вызова.

Напишем небольшую программу которая записывает в файл некоторые текстовые данные.

```
1  #include <iostream>
2  #include <fstream>
3  using namespace std;
4
5  int main() {
6      ofstream myfile;
7      myfile.open("example.txt");
8      myfile << "Writing this to a file.\n";
9      myfile.close();
10     return 0;
11 }
```

Листинг 4.2: test1.cpp

Данная программа была скомпилирована и запущена в API Monitor. В основном меню крайне много полезной информации.

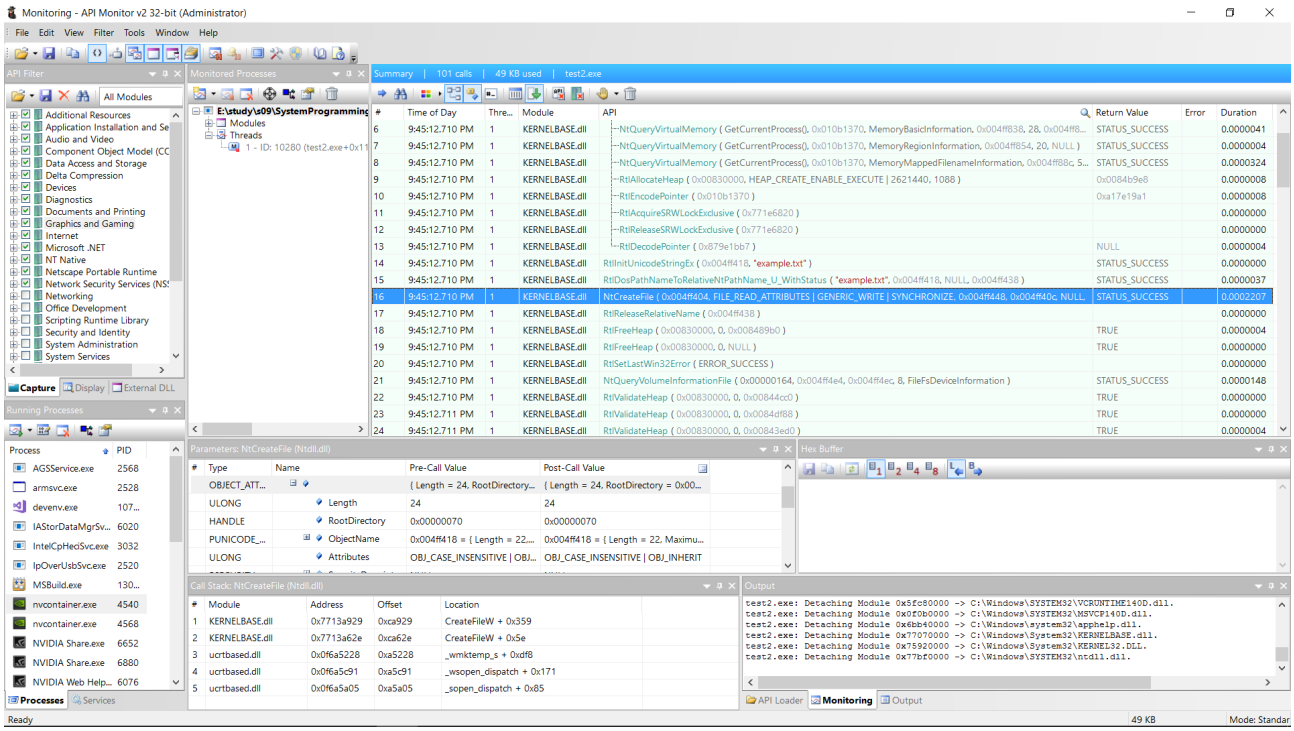

Рис. 4.4: Основное окно API Monitor

В списке вызовов системных функций, была найдена функция **NtCreateFile**[13], которая и необходима для создания файла. Её прототип представлен далее.

```
1  NTSTATUS NtCreateFile(
2    _Out_    PHANDLE            FileHandle,
3    _In_     ACCESS_MASK        DesiredAccess,
4    _In_     POBJECT_ATTRIBUTES ObjectAttributes,
```

```
5    _Out_     PIO_STATUS_BLOCK     IoStatusBlock,
6    _In_opt_  PLARGE_INTEGER       AllocationSize,
7    _In_      ULONG                FileAttributes,
8    _In_      ULONG                ShareAccess,
9    _In_      ULONG                CreateDisposition,
10   _In_      ULONG                CreateOptions,
11   _In_      PVOID                EaBuffer,
12   _In_      ULONG                EaLength
13 );
```

Листинг 4.3: Прототип NtCreateFile

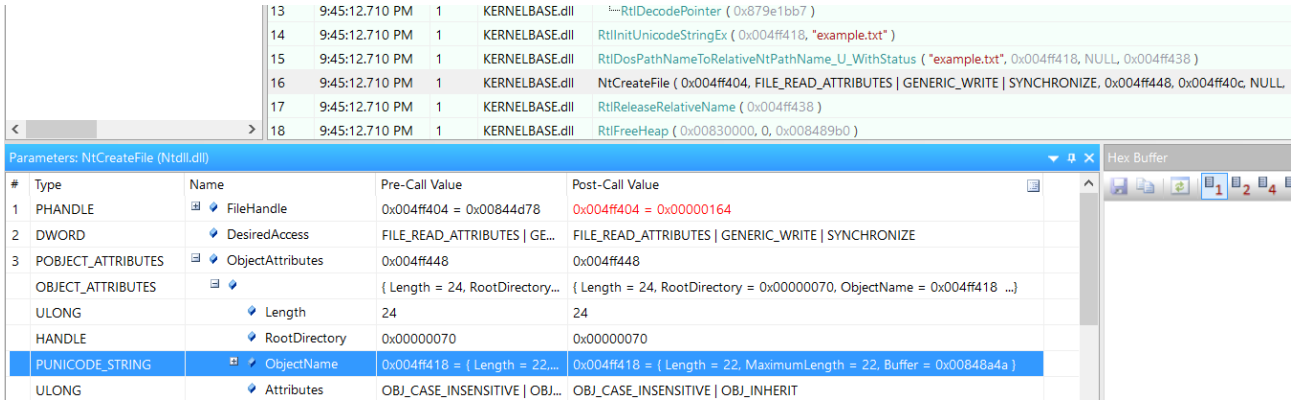

Рис. 4.5: Вызов NtCreateFile

В скриншоте также представлены параметры переданные в функцию. Раскрыв более подробно третий параметр **POBJECT_ATTRIBUTES** в нем была найдена переменная с названием **PUNICODE_STRING** значение которой равнялось **0x004ff418**.

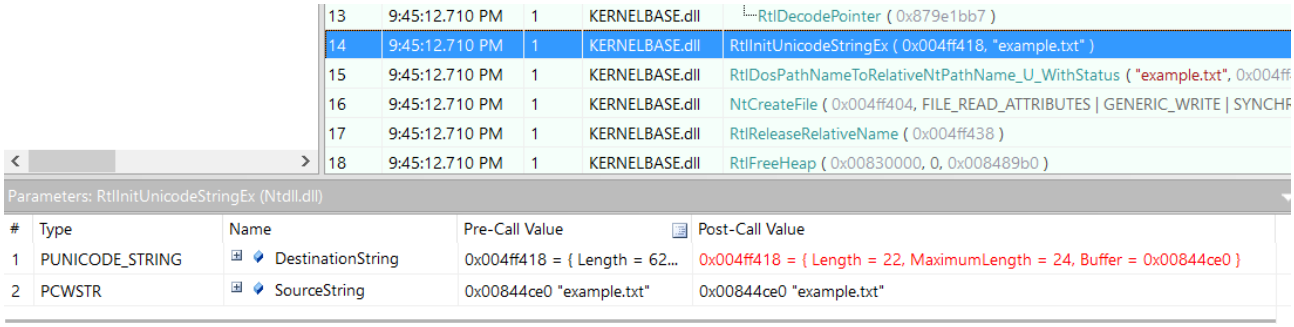

Рис. 4.6: Адрес 0x004ff418

Если еще раз взглянуть на список вызовов, то можно заметить вызов **RtlInitUnicodeStringEx**[14], который инициализирует в адрес 0x004ff418 текст(**example.txt** с адреса 0x00844ce0.

```
1  NTSTATUS RtlInitUnicodeStringEx
2  (
3    PUNICODE_STRING target,
4    PCWSTR          source
5  )
```

Листинг 4.4: Прототип RtlInitUnicodeStringEx

4.3 Windbg

WinDbg — позволяет отлаживать 32/64 битные приложения пользовательского уровня, драйвера, может быть использован для анализа аварийных дампов памяти, поддерживает автоматическую загрузку отладочных символов, имеется встроенный скриптовый язык для автоматизации процесса отладки, а самое главное распространяется корпорацией Microsoft совершенно бесплатно.

Откроем отладчик на примере программы из листинга 4.2.

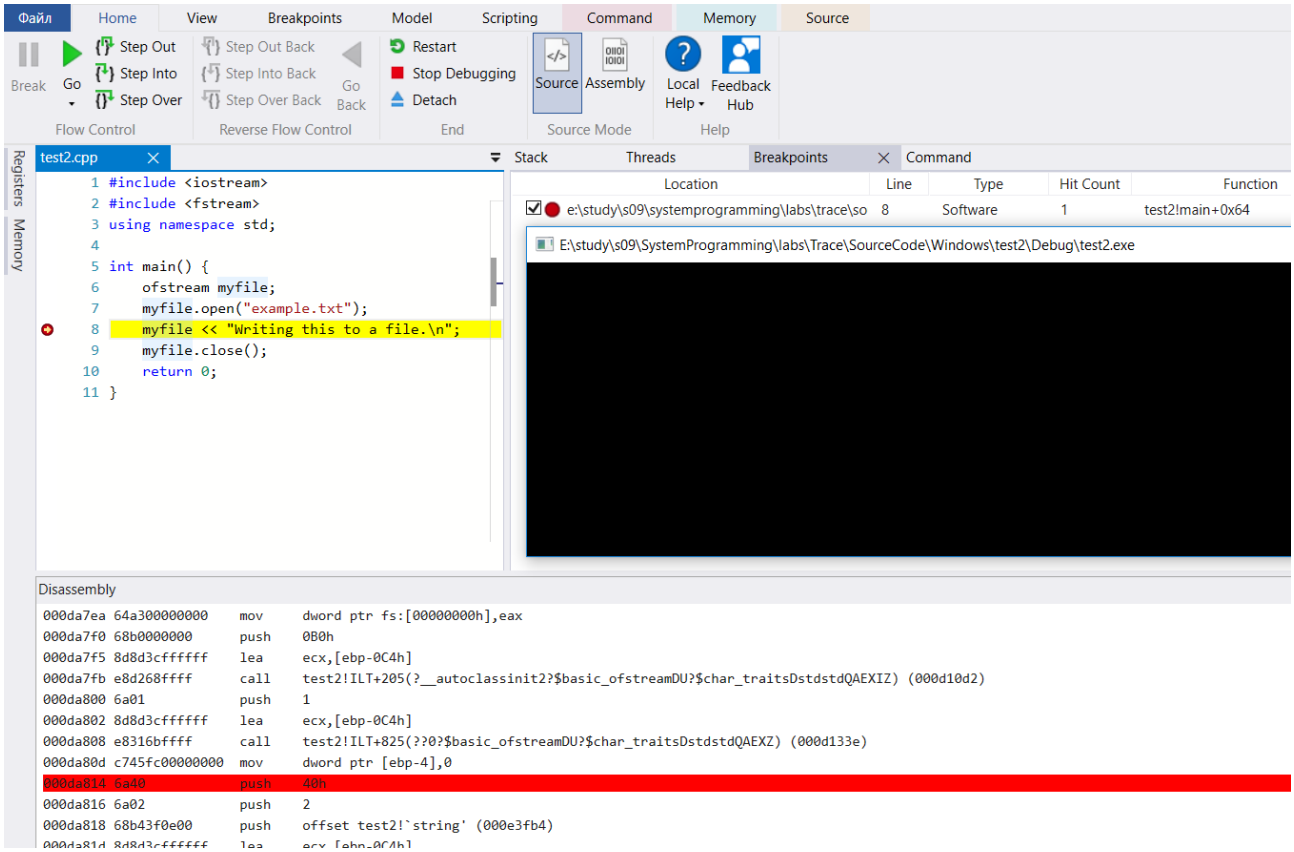

Рис. 4.7: Основное окно WinDbg

Основным преимуществом данного отладчика является работа на уровне ассемблера.

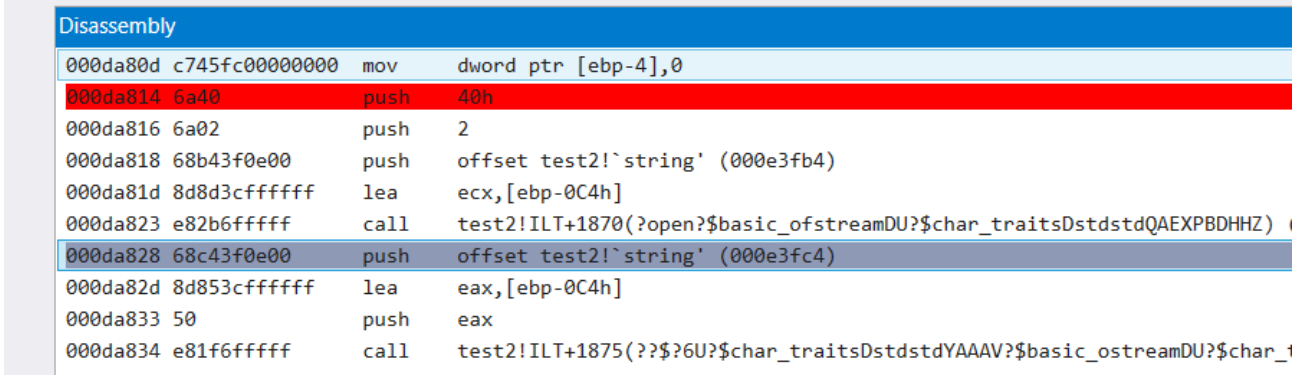

Рис. 4.8: Ассемблерные команды

В списке ассемблерных команд была замечена команда **push offset**, которая помещает в стек адрес строки символов, в данном случае адрес - **0x000e3fc4**. Перейдем по

данному адресу.

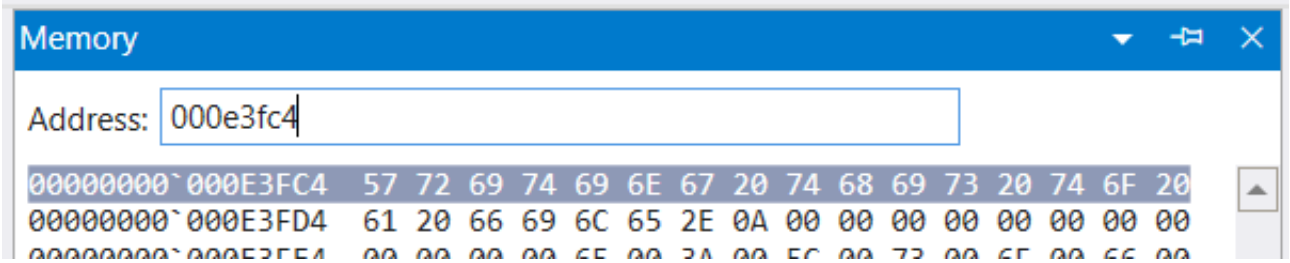

Рис. 4.9: Адрес 0x000e3fc4

Используя таблицу hex кодов[7] символов преобразуем их в читаемый формат.

57	72	69	74	69	6e	67	20	74	68	69	73	20	74	6f	20
W	r	i	t	i	n	g	(sp)	t	h	i	s	(sp)	t	o	(sp)

Таблица 4.1: Расшифровка значений по адресу 0x000e3fc4

62	20	66	69	6c	65	2e	0a
a	(sp)	f	i	l	e	.	(nl)

Таблица 4.2: Расшифровка значений по адресу 0x000e3fd4

Таким образом, с помощью возможностей WinDbg удалось посмотреть сегмент памяти в котором хранились символы для записи в файл.

4.4 Журнал событий

Журнал событий Windows - это средство, позволяющее программам и самой системе Windows регистрировать и хранить уведомления в одном месте. В журнале регистрируются все ошибки, информационные сообщения и предупреждения программ.[**?**]

События из журнала могут быть выгружены в виде файла. Для этого доступны следующие форматы:

- EVTX (Windows Event Log) – бинарный файл специфичной структуры;
- XML – форматированный текст;
- TXT – текстовый формат, значения полей разделены символом табуляции;
- CSV – текстовый формат, значения полей разделены запятой.

Для просмотра журнала событий необходимо:

1. нажать кнопку **Пуск**;
2. выбрать **Панель управления**;
3. выбрать **Администрирование**;
4. выбрать **Просмотр событий**;

5. выбрать интересующий журнал, в данной работе будет использоваться журнал **Приложение**, категории **Журналы Windows**;

6. для просмотра события, кликнуть по нему.

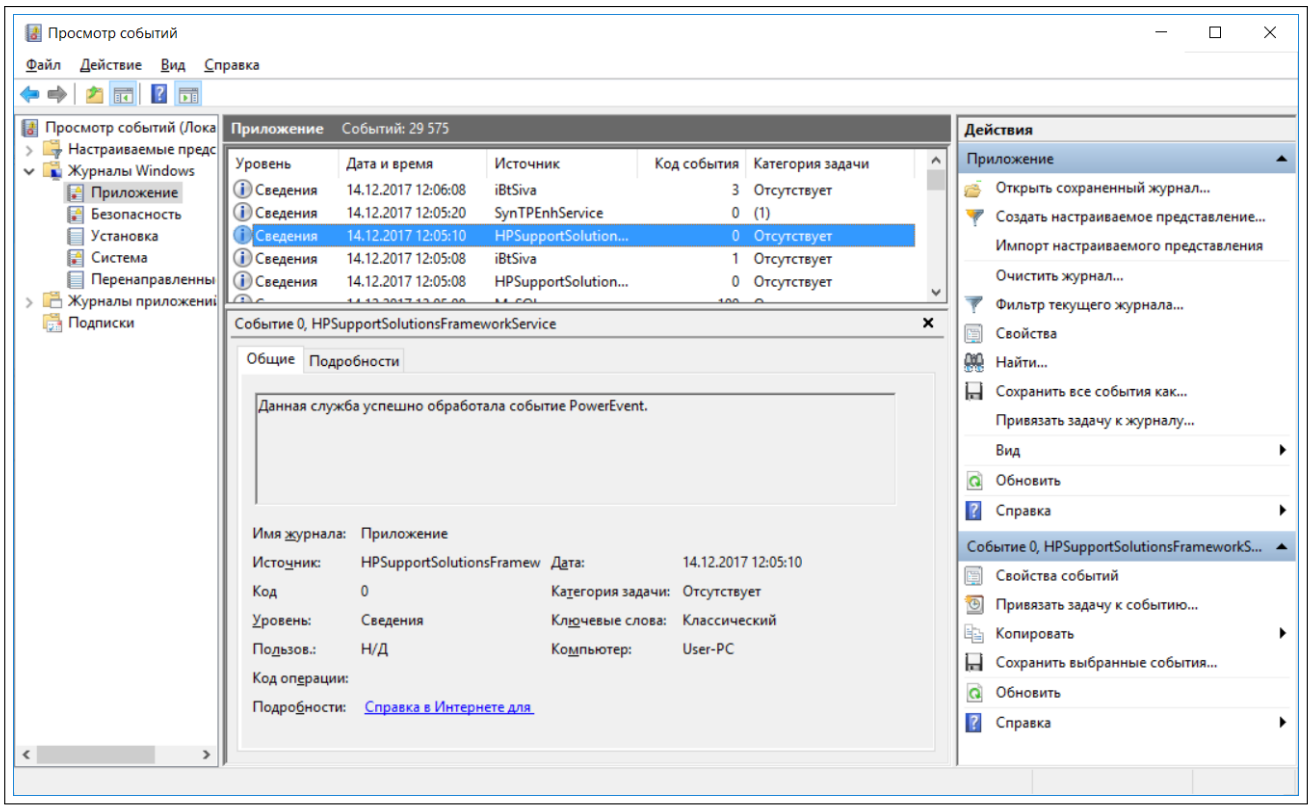

Рис. 4.10: Окно журнала событий

Вывод

В данной работе я познакомился с многими инструментами для профилирования и диагностики программ в Windows и Linux.

В Windows особый интерес представляет программа **API Monitor**, она имеет понятный интерфейс учитывая большой объем выводимой информации. Также она позволяет более подробно рассмотреть параметры передаваемые в функцию, что крайне удобно.

WinDbg также заслуживает внимания, особенно из-за того что это относительно новая программа поставляемая от Microsoft, конечно она и раньше существовала, но по сравнению с текущей это совершенно другой продукт.

В Linux, на мой взгляд большее количество подобных утилит. В том числе есть специфичные инструменты, например **Valgrind memcheck** который направлен на выявление ошибок при работе с памятью.

Литература

[1] Утилиты системного администратора Linux: Strace [Электронный ресурс]. — URL: http://rus-linux.net/MyLDP/consol/strace.html (дата обращения: 2017-12-10).

[2] Write - write to a file descriptor [Электронный ресурс]. — URL: https://linux.die.net/man/2/write (дата обращения: 2017-12-10).

[3] Sigaction - examine and change a signal action [Электронный ресурс]. — URL: https://linux.die.net/man/2/rt_sigaction (дата обращения: 2017-12-10).

[4] ptrace - process trace [Электронный ресурс]. — URL: http://man7.org/linux/man-pages/man2/ptrace.2.html (дата обращения: 2017-12-10).

[5] execl execute a file [Электронный ресурс]. — URL: https://linux.die.net/man/3/execl (дата обращения: 2017-12-10).

[6] wait for process to change state [Электронный ресурс]. — URL: https://linux.die.net/man/2/waitpid (дата обращения: 2017-12-10).

[7] ascii table [Электронный ресурс]. — URL: http://www.bluesock.org/~willg/dev/ascii.html (дата обращения: 2017-12-10).

[8] Протокол syslog [Электронный ресурс]. — URL: http://worm.org.ua/2010/11/rsyslog-setup-opensuse/ (дата обращения: 2017-12-10).

[9] support for application self-debugging [Электронный ресурс]. — URL: http://man7.org/linux/man-pages/man3/backtrace.3.html (дата обращения: 2017-12-10).

[10] Отладка с помощью GDB [Электронный ресурс]. — URL: http://linux.yaroslavl.ru/docs/altlinux/doc-gnu/gdb/gdb.html (дата обращения: 2017-12-10).

[11] Использование Valgrind для поиска утечек и недопустимого использования памяти [Электронный ресурс]. — URL: http://cppstudio.com/post/4348/ (дата обращения: 2017-12-10).

[12] Отладка в Visual Studio [Электронный ресурс]. — URL: https://msdn.microsoft.com/ru-ru/library/k0k771bt.aspx (дата обращения: 2017-12-13).

[13] NtCreateFile function [Электронный ресурс]. — URL: https://msdn.microsoft.com/en-us/library/bb432380(v=vs.85).aspx (дата обращения: 2017-12-13).

[14] RtlInitUnicodeStringEx function [Электронный ресурс]. — URL: https://source.winehq.org/WineAPI/RtlInitUnicodeStringEx.html (дата обращения: 2017-12-13).

[15] Перехват системных вызовов с помощью ptrace [Электронный ресурс]. — URL: https://special.habrahabr.ru/kyocera/p/111266/ (дата обращения: 2017-12-10).

[16] Сообщения программе, ведущей системный журнал [Электронный ресурс]. — URL: https://www.opennet.ru/man.shtml?topic=openlog&category=3&russian=0 (дата обращения: 2017-12-10).