Санкт-Петербургский политехнический университет Петра Великого Институт Информационных Технологий и Управления

Кафедра компьютерных систем и программных технологий

Отчёт по практической работе № 1

по предмету «Проектирование ОС и компонентов»

**Загрузка приложений (Windows/Linux)**

Работу выполнил студент гр. 63501/3 Мартынов С. А. Работу принял преподаватель Душутина Е. В.

Санкт-Петербург 2016

**Содержание**

[Постановка задачи 3](#_TOC_250016)

[Введение 4](#_TOC_250015)

1. [Процесс загрузки приложений в Linux 5](#_TOC_250014)
   1. [ELF – формат исполнения и компоновки 5](#_TOC_250013)
   2. [Сегменты и секции 6](#_TOC_250012)
   3. [Структура и назначение полей служебных заголовков 7](#_TOC_250011)
   4. [Процесс загрузки в память 8](#_TOC_250010)
   5. [Резидентное приложение – монитор сетевой активности 9](#_TOC_250009)
   6. [Динамические библиотеки .so 18](#_TOC_250008)
   7. [Резидентное приложение с динамической библиотекой 19](#_TOC_250007)
2. [Процесс загрузки приложений Windows 21](#_TOC_250006)
   1. [Выполнение ЕХЕ-модуля 21](#_TOC_250005)
   2. [Динамически подключаемые библиотеки 22](#_TOC_250004)
   3. [Реализация резидентного приложения 24](#_TOC_250003)
   4. [Анализ исполнения приложения 27](#_TOC_250002)

[Заключение 31](#_TOC_250001)

[Список литературы 32](#_TOC_250000)

# Постановка задачи

В рамках данной работы необходимо написать полезную программу для ОС семейства Linux и Windows. Программа должна быть выполнена в качестве резидентного (не де- мона) приложения. Далее переписать ту же программу с использованием динамически загружаемой библиотеки.

Таким образом, в результате работы должно получиться четыре программы:

* Резидентное приложение для Windows собранное единым модулем
* Резидентное приложение для Windows с динамической библиотекой (.dll)
* Резидентное приложение для Linux собранное единым модулем
* Резидентное приложение для Linux с динамической библиотекой (.so)

В процессе работы требуется изучить принцип загрузки приложений в различных опе- рационных системах и описать особенности приложений использующих динамические библиотеку.

# Введение

В связи с тем, что сегодня уровень сложности программного обеспечения очень высок, разработка приложений с использованием только какого-либо языка программирования (например, языка C) значительно затрудняется. Программист должен затратить массу времени на решение стандартных задач по созданию многооконного интерфейса. Реализа- ция технологии связывания и встраивания объектов потребует от программиста еще более сложной работы.

Чтобы облегчить работу программиста, следует пере использовать ранее написанный код. С одной стороны это позволяет не решать одну задачу дважды, с другой – появляются дополнительные требования по оформлению существующих решений и их распространению. Долгое время в Unix (а потом Linux) среде распространение велось с исходных кодах. При этом предполагалось, что пользователь достаточно грамотен для работы с таким источником.

С распространением персональных компьютеров и приложений для них, получили популяр- ность динамические библиотеки, которые позволяли, в частости, обновлять приложения без их пере сборки конечным пользователем, использовать различные языки для решения разных задач и даже упрощение локализации.

В данной работе рассматривается процесс загрузки приложений на операционных системах семейства Windows и Linux и порядок работы с динамическими библиотеками.

# Процесс загрузки приложений в Linux

## ELF – формат исполнения и компоновки

Изначально UNIX (и производные от нее операционные системы) поддерживали множество исполняемых форматов, но теперь стандартом де-факто для LINUX и BSD стал ELF. Стандарт для формата ELF изначально был разработан и опубликован компанией USL как часть двоичного интерфейса приложений операционной системы UNIX System V. Затем он был выбран комитетом TIS и развит в качестве переносимого формата для различных операционных систем, работающих на 32-разрядной аппаратной архитектуре Intel x86. ELF быстро набрал популярность и, после того как компания HP расширила формат и опубликовала стандарт ELF-64, распространился и на 64-разрядных платформах. Иногда еще встречается древний a.out, но это достаточно особые случаи, требующие совместимости с железом.

Аббревиатура ELF расшифровывается как Execution and Linkable Format (формат исполне- ния и компоновки). Он во многом напоминает win32 PE. В начале ELF-файла расположен служебный заголовок (ELF-header), описывающий основные характеристики файла — тип (исполнения или линковки), архитектура ЦП, виртуальный адрес точки входа, размеры и смещения остальных заголовков. . .

За ELF-header’ом следует таблица сегментов (program header table), перечисляющая имею- щиеся сегменты и их атрибуты. В формате линковки она необязательно. Линкеру сегменты не важны и он работает исключительно на уровне секций. Напротив, системный загрузчик, загружающий исполняемый ELF-файл в память, игнорирует секции, и оперирует целыми сегментами[1].

Стандарт формата ELF различает несколько типов файлов:

* + - Перемещаемый файл – хранит инструкции и данные, которые могут быть связаны с другими объектными файлами. Результатом такой связи может быть разделяемый объектный файл или исполняемый файл. К этому типу относятся объектные файлы статических библиотек.
    - Разделяемый объектный файл – также содержит инструкции и данные и может быть связан с другими перемещаемыми файлами и разделяемыми объектными файлами, в результате чего будет создан новый объектный файл, либо при запуске программы на выполнение операционная система может динамически связать его с исполняемым файлом программы, в результате чего будет создан исполняемый образ программы. В последнем случае речь идет о разделяемых библиотеках.
    - Исполняемый файл – содержит полное описание, позволяющее системе создать образ процесса. В том числе: инструкции, данные, описание необходимых разделяемых объектных файлов и необходимую символьную и отладочную информацию.

## Сегменты и секции

Сегмент – это непрерывная область адресного пространства со своими атрибутами доступа. В частности, сегмент кода имеет атрибут исполнения, а сегмент данных – атрибуты чтения и записи. Стоит отметить, что ELF-сегменты это не сегменты x86 процессора! В защищенном режиме 386+ никаких "сегментов"уже нет, а есть только селекторы и все сегменты ELF-файла загружается в единый 4 Гбайтовый x86-сегмент! В зависимости от типа сегмента, величина выравнивания в памяти может варьировать от 4h до 1000h байт (размер страницы на x86). В самом ELF-файле хранятся в невыровненном виде, плотно прижатые друг к другу.

Ближайший аналог ELF-сегментов – PE-секции, но в PE-файлах, секция – это наименьшая структурная единица, а в ELF-файлах сегмент может быть разбит на один или несколько фрагментов – секций. В частности, типичный кодовый сегмент состоит из

* + - секций .init – процедуры инициализации,
    - секции .plt – секция связок,
    - секции .text – основой код программы,
    - секции .finit – процедуры финализации.

Секции нужны линкеру для комбинирования, чтобы он мог отобрать секции с похожими атрибутами и оптимальным образом растасовать их по сегментам при сборке файла, то есть "скомбинировать"[2].

Несмотря на то, что системный загрузчик игнорирует таблицу секций, линкер все-таки помещает ее копию в исполняемый файл. Это приводит к не значительному расходу места, зато эта информация полезна для отладчиков и дизассемблеров. По не совсем понятным причинам gdb и многие другие программы отказываются загружать в файл с поврежденной или отсутствующей таблицей секций, чем часто пользуются для защиты программ от постороннего вмешательства. Структура файла представлена на рисунке 1.

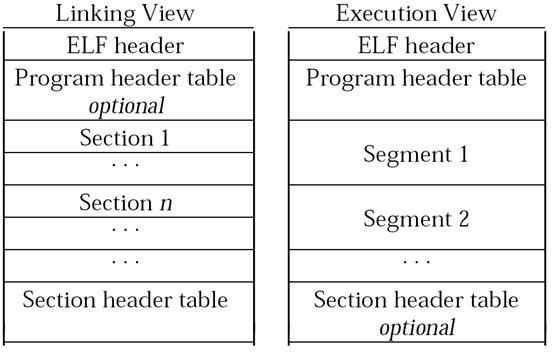


Рис. 1: Структура ELF-формат с точки зрения линкера (слева) и системного загрузчика операционной системы (справа)

## Структура и назначение полей служебных заголовков

Заголовок файла (ELF Header) имеет фиксированное расположение в начале файла и содержит общее описание структуры файла и его основные характеристики, такие как: тип, версия формата, архитектура процессора, виртуальный адрес точки входа, размеры и смещения остальных частей файла.

* + - *e\_ident[]* – Массив байт, каждый из которых определяет общую характеристику файла. Первые четыре байта в массиве определяют сигнатуру файла и всегда должны содержать 0x7f 0x45 0x4c 0x46 соответственно.
    - *e\_type* – Тип файла.
    - *e\_machine* – Архитектура аппаратной платформы, для которой файл создан.
    - *e\_version* – Номер версии формата.
    - *e\_entry* – Точка входа.
    - *e\_phoff* – Расположение таблицы заголовков программы.
    - *e\_shoff* – Расположение таблицы заголовков разделов.
    - *e\_flags* – Связанные с файлом флаги, зависящие от процессора.
    - *e\_ehsize* – Размер[5] заголовка файла.
    - *e\_phentsize* – Размер каждого заголовка программы.
    - *e\_phnum* – Число заголовков программы.
    - *e\_shentsize* – Размер каждого заголовка разделов.
    - *e\_shnum* – Число заголовков разделов.
    - *e\_shstrndx* – Индекс записи в таблице разделов, указывающей на таблицу названий разделов.

## Процесс загрузки в память

По умолчанию ELF-заголовок проецируется по адресу 8048000h, который прописан в его заголовке. Это и есть базовый адрес загрузки. На стадии линковки он может быть свободно изменен на другой, но большинство программистов оставляют его "как есть". Все сегменты проецируются в память в соответствии с виртуальными адресами, прописанными в таблице сегментов, причем, виртуальная проекция образа всегда непрерывна, и между сегментами не должно быть незаполненных "дыр".

Начиная с адреса 40000000h располагаются совместно используемые библиотеки ld-linix.so, libm.so, libc.so и другие, которые связывают операционную систему с прикладной програм- мой. Ближайший аналог из мира Windows – KERENL32.DLL, реализующая win32 API, что расшифровывается как Application Programming Interface, но при желании программа может вызывать функции операционной системы и напрямую. В NT за это отвечает прерывание INT 2Eh, в LINUX – как правило INT 80h (на самом деле к текущему моменту в этом вопросе была проделана некоторая оптимизация, о которой будет сказано позже, при рассмотрении вывода утилиты ldd)[3].

Для вызова функций типа открытия файла мы можем обратиться либо к библиотеке libc, либо непосредственно к самой операционной системе. Первый вариант – самый громоздкий, самый переносимый, и наименее приметный. Последний – прост в реализации, но испытывает проблемы совместимости с различными версиями LINUX’а.

Последний гигабайт адресного пространства (от адреса C0000000h и выше) занимают код и данные операционной системе, к которым можно обращаться только посредством прерывания INT 80h или через разделяемые библиотеки.

Стек находится в нижних адресах. Он начинается с базового адреса загрузки и "растет вверх"по направлению к нулевым адресам. В большинстве Линукс-систем стек исполняем

(то есть сюда можно скопировать машинный код и передать на него управления), одна- ко, некоторые администраторы устанавливают заплатки, отнимающие у стека атрибут исполнимости. Карта памяти представлена на рисунке 2.

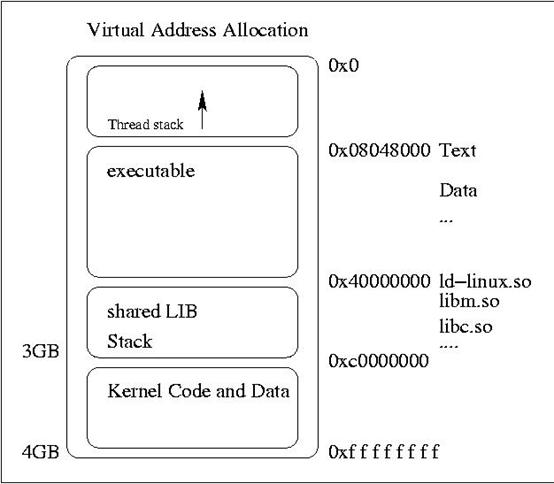


Рис. 2: Карта памяти загруженного образа исполняемого файла

## Резидентное приложение – монитор сетевой активности

В качестве полезного приложение было решено создать простую утилиту, которая отобра- жает количество полученных и отправленных пакетов по указанному сетевому интерфейсу. Процесс организации интерфейса с пользователем интереса не представляет, но работа с системой построена по средствам извлечения информации из файла /proc/net/dev и представлена в листинге 1.

Листинг 1: Функция получения информации о трафике по сетевому интерфейсу (src/ELF/lin/parse.cpp)

bool parse ( char *\** ifname , long long *\** rx\_bytes , long long *\** rx\_packets , long long *\** tx\_bytes , long long *\** tx\_packets ) {

7

8

|  |  |  |  |
| --- | --- | --- | --- |
| 9  10  11  12  13  14  15  16  17  18  19 |  | std : : s t r i n g i n t e r f a c e ( i fname ) ; i n t e r f a c e . append ( " : " ) ;  std : : s t r i n g buf f ;  std : : i f s tre a m n e ts ta t ( "/ proc / net/ dev" ) ;  while ( std : : g e t l i n e ( ne ts tat , buf f ) ) {  s i z e \_ t s h i f t = buf f . f ind\_ f i rst\_ not\_ of ( ’ ’ i f ( buf f . compare ( s h i f t , i n t e r f a c e . l e ng th (  {  std : : regex rx (R" ( [ ^ [ : alpha : ] ] [ [ : d i g i t  ;  std : : s re g e x \_ i te ra to r pos ( buf f . cbegin (  ; | |
| 20 |  |  | *\** rx\_bytes = std : : s t o l l ( pos*−*>s t r ( ) ) ; |
| 21 |  |  | ++pos ; |
| 22 |  |  | *\** rx\_packets = std : : s t o l l ( pos*−*>s t r ( ) ) ; |
| 23 |  |  | std : : advance ( pos , 7 ) ; |
| 24 |  |  | *\** tx\_bytes = std : : s t o l l ( pos*−*>s t r ( ) ) ; |
| 25 |  |  | ++pos ; |
| 26 |  |  | *\** tx\_packets = std : : s t o l l ( pos*−*>s t r ( ) ) ; |
| 27 |  |  |  |
| 28 |  |  | re turn true ; |
| 29 |  | } |  |
| 30 |  | } |  |
| 31 |  | re turn | f a l s e ; |
| 32 | } | | |

После компиляции можно собрать информацию об объектном файле. Полная демонстрация возможностей objdump займёт довольно много места, но основные возможности представ- лены в следующем листинге 2. В 1-й строке запрашивается информация о хедерах файла, их именах и расположении.

) ;

) , i n t e r f a c e ) == 0 )

: ] ] + [ ^ [ : alpha : ] ] ) " )

) , buf f . cend ( ) , rx )

Листинг 2: Демонстрация работы программы objdump

user@ host$ objdump *−−*h eaders . / netmonitor

. / netmonitor :

f i l e format e l f 6 4 *−*x86 *−*64

1

2

3

4

5 S e c t i o ns :

6 Idx Name S i z e VMA LMA F i l e o f f Algn

7 0 . i n te rp 0000001 c 0000000000400238 0000000000400238

|  |  |  |  |
| --- | --- | --- | --- |
| 8 |  | 00000238 2*\*\**0  CONTENTS, | |
| 9  10  11 | 1  2 | . note . ABI*−*ta g 00000020  00000254 2*\*\**2  CONTENTS,  . note . gnu . build *−*id 0000 00000274 2*\*\**2 | |
| 12 |  |  | CONTENTS, |
| 13 | 3 | . gnu . hash | 000000 b4 |
|  |  | 00000298 | 2*\*\**3 |
| 14 |  |  | CONTENTS, |
| 15 | 4 | . dynsym | 00000918 |
|  |  | 00000350 | 2*\*\**3 |
| 16 |  |  | CONTENTS, |
| 17 | 5 | . dynstr | 00000 e70 |
|  |  | 00000 c68 | 2*\*\**0 |
| 18  19 | CONTENTS,  6 . gnu . v e r s i o n 000000 c2 | | |
|  |  | 00001 ad8 2*\*\**1 | |
| 20 |  | CONTENTS, | |
| 21 | 7 | . gnu . version\_ r 000000 f 0 | |
|  | 00001 ba0 2*\*\**3 | | |
| 22 |  |  | CONTENTS, |
| 23 | 8 | . r e l a . dyn | 00000180 |
|  |  | 00001 c90 | 2*\*\**3 |
| 24 |  |  | CONTENTS, |
| 25 | 9 | . r e l a . p l t | 000006 f 0 |
|  | 00001 e10 2*\*\**3 | | |
| 26 |  |  | CONTENTS, |
| 27 | 10 | . i n i t | 0000001 a |
|  | 00002500 2*\*\**2 | | |
| 28 |  |  | CONTENTS, |
| 29 | 11 | . p l t | 000004 b0 |

ALLOC, LOAD, READONLY, DATA

0000000000400254 0000000000400254

ALLOC, LOAD, READONLY, DATA

0024 0000000000400274 0000000000400274

ALLOC, LOAD, READONLY, DATA

0000000000400298 0000000000400298

ALLOC, LOAD, READONLY, DATA

0000000000400350 0000000000400350

ALLOC, LOAD, READONLY, DATA

0000000000400 c68 0000000000400 c68

ALLOC, LOAD, READONLY, DATA

0000000000401 ad8 0000000000401 ad8

ALLOC, LOAD, READONLY, DATA

0000000000401 ba0 0000000000401 ba0

ALLOC, LOAD, READONLY, DATA

0000000000401 c90 0000000000401 c90

ALLOC, LOAD, READONLY, DATA

0000000000401 e10 0000000000401 e10

ALLOC, LOAD, READONLY, DATA

0000000000402500 0000000000402500

ALLOC, LOAD, READONLY, CODE

0000000000402520 0000000000402520

|  |  |  |  |
| --- | --- | --- | --- |
| 30  31  32  33  34 | 00002520 2*\*\**4  CONTENTS, ALLOC, LOAD, READONLY, CODE  12 . p l t . got 00000008 00000000004029 d0 00000000004029 d0  000029 d0 2*\*\**3  CONTENTS, ALLOC, LOAD, READONLY, CODE  13 . te xt 00011 b42 00000000004029 e0 00000000004029 e0  000029 e0 2*\*\**4  CONTENTS, ALLOC, LOAD, READONLY, CODE | | |
| 35 | 14 | . f i n i 00000009 | 0000000000414524 0000000000414524 |
| 36 |  | 00014524 2*\*\**2  CONTENTS, | ALLOC, LOAD, READONLY, CODE |
| 37 | 15 | . rodata 00000 d33 | 0000000000414540 0000000000414540 |
| 38 |  | 00014540 2*\*\**5  CONTENTS, | ALLOC, LOAD, READONLY, DATA |
| 39 | 16 | . eh\_frame\_hdr 0000058 c | 0000000000415274 0000000000415274 |
| 40 |  | 00015274 2*\*\**2  CONTENTS, | ALLOC, LOAD, READONLY, DATA |
| 41 | 17 | . eh\_frame 00002734 | 0000000000415800 0000000000415800 |
| 42 |  | 00015800 2*\*\**3  CONTENTS, | ALLOC, LOAD, READONLY, DATA |
| 43 | 18 | . gcc\_except\_table 00000 | 875 0000000000417 f 34 0000000000417 f 34 |
| 44 |  | 00017 f 34 2*\*\**2  CONTENTS, | ALLOC, LOAD, READONLY, DATA |
| 45 | 19 | . i ni t\_ array 00000010 | 0000000000618 de8 0000000000618 de8 |
| 46 |  | 00018 de8 2*\*\**3  CONTENTS, | ALLOC, LOAD, DATA |
| 47 | 20 | . f i ni \_ array 00000008 | 0000000000618 df 8 0000000000618 df 8 |
| 48 |  | 00018 df 8 2*\*\**3  CONTENTS, | ALLOC, LOAD, DATA |
| 49 | 21 | . j c r 00000008 | 0000000000618 e00 0000000000618 e00 |
| 50 |  | 00018 e00 2*\*\**3  CONTENTS, | ALLOC, LOAD, DATA |
| 51 | 22 | . dynamic 000001 f 0 | 0000000000618 e08 0000000000618 e08 |
| 52 |  | 00018 e08 2*\*\**3  CONTENTS, | ALLOC, LOAD, DATA |
| 53 | 23 | . got 00000008 | 0000000000618 f f 8 0000000000618 f f 8 |
|  |  | 00018 f f 8 2*\*\**3 |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 54 |  | CONTENTS, | ALLOC, LOAD, DATA |  |
| 55 | 24 | . got . p l t 00000268 | 0000000000619000 0000000000619000 |
| 56 |  | 00019000 2*\*\**3  CONTENTS, | ALLOC, LOAD, DATA |
| 57 | 25 | . data 00000420 | 0000000000619280 0000000000619280 |
| 58 |  | 00019280 2*\*\**5  CONTENTS, | ALLOC, LOAD, DATA |
| 59 | 26 | . bss 00000648 | 00000000006196 a0 00000000006196 a0 |
| 60 |  | 000196 a0 2*\*\**5  ALLOC |  |
| 61 | 27 | . comment 0000002 f | 0000000000000000 0000000000000000 |
| 62 |  | 000196 a0 2*\*\**0  CONTENTS, | READONLY |
| 63 | 28 | . debug\_aranges 00000 ab0 | 0000000000000000 0000000000000000 |
| 64 |  | 000196 c f 2*\*\**0  CONTENTS, | READONLY, DEBUGGING |
| 65 | 29 | . debug\_info 00082 dbe | 0000000000000000 0000000000000000 | 0001 |
| 66 |  | a 17 f 2*\*\**0  CONTENTS, | READONLY, DEBUGGING |  |
| 67 | 30 | . debug\_abbrev 000019 f 8 | 0000000000000000 0000000000000000 | 0009 |
| 68 |  | c f 3 d 2*\*\**0  CONTENTS, | READONLY, DEBUGGING |  |
| 69 | 31 | . debug\_line 00008 cb8 | 0000000000000000 0000000000000000 | 0009 |
| 70 |  | e 935 2*\*\**0  CONTENTS, | READONLY, DEBUGGING |  |
| 71 | 32 | . debug\_str 0007 f 0 c 7 | 0000000000000000 0000000000000000 | 000 |
| 72 |  | a75ed 2*\*\**0  CONTENTS, | READONLY, DEBUGGING |  |
| 73 | 33 | . debug\_loc 0004 c 7 c 9 | 0000000000000000 0000000000000000 |  |
| 74 |  | 001266 b4 2*\*\**0  CONTENTS, | READONLY, DEBUGGING |  |
| 75 | 34 | . debug\_ranges 00012670 | 0000000000000000 0000000000000000 |  |
| 76 |  | 00172 e7d 2*\*\**0  CONTENTS, | READONLY, DEBUGGING |  |

Другой удобной программой для вывода информации о ELF файле является readelf (вывод программы приведён в сокращенном виде, листинг 3).

Листинг 3: Демонстрация работы программы readelf

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 89  90 | ELF Header :  Magic : 7 f 45 4 c 46 02 01 01 03 00 00 00 00 00 00 00 00 | | | | | | |
| 91 | Class : | | |  | ELF64 | | |
| 92 | Data : | | |  | 2 ’ s complement , l i t t l e endian | | |
| 93 | Version : | | |  | 1 ( c urre nt ) | | |
| 94 | OS/ABI : | | |  | UNIX *−* GNU | | |
| 95 | ABI Version : | | |  | 0 | | |
| 96 | Type : | | |  | EXEC ( Executable f i l e ) | | |
| 97 | Machine : | | |  | Advanced Micro Devices X86*−*64 | | |
| 98 | Version : | | |  | 0 x1 | | |
| 99 | Entry point address : | | |  | 0 x402 c 20 | | |
| 100 | S tart o f program headers : | | |  | 64 ( bytes i nto f i l e ) | | |
| 101 | S tart o f s e c t i o n headers : | | |  | 1635848 ( bytes i nto f i l e ) | | |
| 102 | Flags : | | |  | 0 x0 | | |
| 103 | S i z e o f t h i s header : | | |  | 64 ( bytes ) | | |
| 104 | S i z e o f program headers : | | |  | 56 ( bytes ) | | |
| 105 | Number o f program headers : | | |  | 9 | | |
| 106 | S i z e o f s e c t i o n headers : | | |  | 64 ( bytes ) | | |
| 107 | Number o f s e c t i o n headers : | | |  | 39 | | |
| 108 | S e c t i o n header s t r i n g tab l e | | | index : | 36 | | |
| 109 |  | | |  |  | | |
| 110 |  | | |  |  | | |
| 111 | Program Headers : | | |  |  | | |
| 112 | Type O f f s e t | | |  | VirtAddr PhysAddr | | |
| 113 |  | | F i l e S i z | | MemSiz | Flags Align | |
| 114 | PHDR | | 0 x0000000000000040 | | 0 x0000000000400040 | 0 | |
|  | x0000000000400040 | | | | | | |
| 115 |  | | 0 x 00000000000001 f 8 | | 0 x 00000000000001 f 8 | R E | 8 |
| 116 | INTERP | | 0 x0000000000000238 | | 0 x0000000000400238 | 0 |  |
|  | x0000000000400238 | | | | | | |
| 117 |  | 0 x 000000000000001 c 0 x 000000000000001 c R 1 | | | | | |
| 118 |  | [ Requesting program i n t e r p r e t e r : / l i b 6 4 / ld*−*linux *−*x86 *−*64. so . 2 ] | | | | | |
| 119 | LOAD | 0 x0000000000000000 0 x0000000000400000 0 | | | | | |
|  | x0000000000400000 | | | | | | |
| 120 | 0 x00000000000187 a9 0 x00000000000187 a9 R E | | | | | | |
|  | 200000 | | | | | | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 121 | LOAD 0 x0000000000018 de 8 0 x0000000000618 de 8 0  x0000000000618 de 8 | | | | | | | |
| 122 |  | | | 0 x00000000000008 b8 | 0 x 0000000000000 f 00 | RW | | |
|  |  | | | 200000 |  |  | | |
| 123 | DYNAMIC | | | 0 x 0000000000018 e 08 | 0 x 0000000000618 e 08 | 0 | | |
|  | x 0000000000618 e 08 | | | | | | | |
| 124 |  | | | 0 x 00000000000001 f 0 | 0 x 00000000000001 f 0 | RW | 8 | |
| 125 | NOTE | | | 0 x0000000000000254 | 0 x0000000000400254 | 0 |  | |
|  | x0000000000400254 | | | | | | | |
| 126 |  | | | 0 x0000000000000044 | 0 x0000000000000044 | R | 4 | |
| 127 | GNU\_EH\_FRAME | | | 0 x0000000000015274 | 0 x0000000000415274 | 0 |  | |
|  | x0000000000415274 | | | | | | | |
| 128 |  | | | 0 x 000000000000058 c | 0 x 000000000000058 c | R | 4 | |
| 129 | GNU\_STACK | | | 0 x0000000000000000 | 0 x0000000000000000 | 0 |  | |
|  | x0000000000000000 | | | | | | | |
| 130 |  | | | 0 x0000000000000000 | 0 x0000000000000000 | RW | 10 | |
| 131 | GNU\_RELRO | | | 0 x0000000000018 de 8 | 0 x0000000000618 de 8 | 0 |  | |
| 132  133  134 | x0000000000618 de 8  0 x0000000000000218 0 x0000000000000218 R 1  S e c t i o n to Segment mapping : | | | | | | | |
| 135 | Segment S e c t i o ns . . . | | | | | | |  |
| 136 | 00 |  | | | | | |  |
| 137 | 01 | . i n te rp | | | | | |  |
| 138 | 02 | . i n te rp . note . ABI*−*ta g . note . gnu . build *−*id . gnu . hash | | | | | | . dynsym |
|  | . dynstr . gnu . v e r s i o n . gnu . version\_ r . r e l a . dyn . r e l a . p l t . i n i t  . p l t . p l t . got . te xt . f i n i . rodata . eh\_frame\_hdr . eh\_frame . gcc\_except\_table | | | | | | | |
| 139 | 03 |  | . i ni t\_ array . f i ni \_ array . j c r . dynamic . got . got . p l t . data | | | | | |
|  |  | . bss |  | | | | | |
| 140 | 04 |  | . dynamic | | | | | |
| 141 | 05 |  | . note . ABI*−*ta g . note . gnu . build *−*id | | | | | |
| 142 | 06 |  | . eh\_frame\_hdr | | | | | |
| 143 | 07 |  |  | | | | | |
| 144 | 08 |  | . i ni t\_ array . f i ni \_ array . j c r . dynamic . got | | | | | |

Информацию о символах можем получить при помощи утилиты ss (нас интересует функция

parse, строка 56, листинг 4).

|  |  |  |  |
| --- | --- | --- | --- |
| 40 | 0000000000414520 | T | \_\_libc\_csu\_fini |
| 41 | 00000000004144 b0 | T | \_\_libc\_csu\_init |
| 42 |  | U | \_\_libc\_start\_main@@GLIBC\_2 . 2 . 5 |
| 43 | 00000000004029 e0 | T | main |
| 44 |  | U | memcmp@@GLIBC\_2 . 2 . 5 |
| 45 |  | U | memcpy@@GLIBC\_2. 1 4 |
| 46 |  | U | memmove@@GLIBC\_2 . 2 . 5 |
| 47 |  | w | \_\_pthread\_key\_create |
| 48 | 0000000000402 c90 | t | register\_ tm\_ c lones |
| 49 |  | U | \_\_stack\_chk\_fail@@GLIBC\_2 . 4 |
| 50 | 0000000000402 c20 | T | \_ start |
| 51 |  | U | strchr@@GLIBC\_2 . 2 . 5 |
| 52 |  | U | strlen@@GLIBC\_2 . 2 . 5 |
| 53 |  | U | strtoll@@GLIBC\_2 . 2 . 5 |
| 54 | 00000000006196 a0 | D | TMC\_END |
| 55 |  | U | \_Unwind\_Resume@@GCC\_3 . 0 |
| 56 | 0000000000403140 | T | \_Z5parsePcPxS0\_S0\_S0\_ |
| 57 |  | U | \_ZdlPv@@GLIBCXX\_3 . 4 |
| 58 |  | U | \_ZdlPvm@@CXXABI\_1 . 3 . 9 |
| 59  60  61  62  63  64 | 0000000000619 cd8 u  \_ZGVZNKSt8\_\_d  0000000000619 cc 8 u  \_ZGVZNKSt8\_\_d  0000000000619 cb8 u  \_ZGVZNKSt8\_\_d  0000000000619 ca8 u  \_ZGVZNKSt8\_\_d  00000000004048 e0 W  \_ZN9 gnu\_cxx6 00000000004048 e0 W  \_ZN9 gnu\_cxx6 | | etail11\_AnyMatcherINSt7\_\_cxx1112  etail11\_AnyMatcherINSt7\_\_cxx1112  etail11\_AnyMatcherINSt7\_\_cxx1112  etail11\_AnyMatcherINSt7\_\_cxx1112  stoaIxxcIiEEET0\_PFT\_PKT1\_PPS3\_D stoaIxxcJiEEET0\_PFT\_PKT1\_PPS3\_ |

Листинг 4: Демонстрация работы программы nm

regex\_traitsIcEELb0ELb0ELb0

regex\_traitsIcEELb0ELb0ELb1

regex\_traitsIcEELb0ELb1ELb0

regex\_traitsIcEELb0ELb1ELb1

pT2\_EPKcS5\_PmS9\_

DpT2\_EPKcS5\_PmS9\_

65

U \_ZNKSt5ctypeIcE13\_M\_widen\_initEv@@GLIBCXX\_3 . 4 . 1 1

0000000000403 cd0 W \_ZNKSt5ctypeIcE8do\_widenEc

0000000000403 ce 0 W \_ZNKSt5ctypeIcE9do\_narrowEcc

U \_ZNKSt6locale2id5\_M\_idEv@@GLIBCXX\_3 . 4

66

67

68

Зависимость от библиотек показывает утилита ldd (листинг 5). Тут стоит обратить внима- ние на виртуальную библиотеку в строке 1. В те времена, когда процессоры с архитектурой x86 только появились, взаимодействие пользовательских приложений со службами опера- ционной системы осуществлялось с помощью прерываний. По мере создания более мощных процессоров эта схема взаимодействия становилась узким местом системы. Во всех про- цессорах, начиная с Pentium II, Intel реализовала механизм быстрых системных вызовов (Fast System Call), в котором вместо прерываний используются инструкции SYSENTER и SYSEXIT, ускоряющие выполнение системных вызовов.

Библиотека linux-vdso.so.1 является виртуальной библиотекой, или виртуальным динамиче- ски разделяемым объектом (VDSO), который размещается только в адресном пространстве отдельной программы. В более ранних системах эта библиотека называлась linux-gate.so.1. Эта виртуальная библиотека содержит всю необходимую логику, обеспечивающую для пользовательских приложений наиболее быстрый доступ к системным функциям в зави- симости от архитектуры процессора – либо через прерывания, либо (для большинства современных процессоров) через механизм быстрых системных вызовов.

Листинг 5: Демонстрация работы программы ldd

l inux *−*vdso . so . 1 => ( 0 x 00007 f fd 1 a 560000 )

l i b s t d c ++.so . 6 => / usr / l i b /x86\_64*−*linux *−*gnu/ l i b s t d c ++.so . 6 ( 0 x 00007 f bf af 1 a 0000 )

l i bg c c \_ s . so . 1 => / l i b /x86\_64*−*linux *−*gnu/ l i bg c c \_ s . so . 1 ( 0 x 0 0 0 0 7 f bf ae f 8 a 0 0 0 )

l i b c . so . 6 => / l i b /x86\_64*−*linux *−*gnu/ l i b c . so . 6 ( 0 x 00007 f bf aebc 0000 ) l ibm . so . 6 => / l i b /x86\_64*−*linux *−*gnu/ l ibm . so . 6 ( 0 x 00007 fbfae 8 b 7000 )

/ l i b 6 4 / ld*−*linux *−*x86 *−*64. so . 2 ( 0 x 0000562 b 8 f 3 ca 000 )

1

2

3

4

5

6

Адрес загрузки программы не меняется, однако адреса подключения динамических биб- лиотек и область размещения стека изменяются при повторном запуске. Отсюда можно сделать вывод о том, что представленные адреса виртуальные.

Теперь программу можно запустить для проверки сетевого интерфейса каждые 2 секунды. Работа программы показана на рисунке 3.

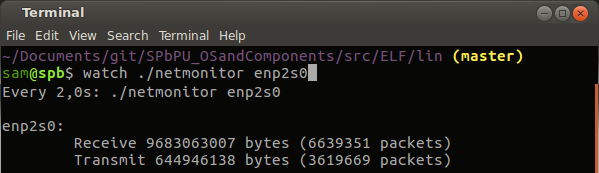


Рис. 3: Исполнение программы

## Динамические библиотеки .so

Библиотека - это набор скомпонованных особым образом объектных файлов. Библиотеки подключаются к основной программе во время линковки. По способу компоновки биб- лиотеки подразделяют на архивы (статические библиотеки, static libraries) и совместно используемые (динамические библиотеки, shared libraries). В Linux, кроме того, есть меха- низмы динамической подгрузки библиотек. Суть динамической подгрузки состоит в том, что запущенная программа может по собственному усмотрению подключить к себе какую- либо библиотеку. Благодаря этой возможности создаются программы с подключаемыми плагинами, такие как XMMS.

Статическая библиотека - это просто архив объектных файлов, который подключается к программе во время линковки. Эффект такой же, как при компиляции файлов отдельно.

В отличие от статических библиотек, код совместно используемых (динамических) биб- лиотек не включается в бинарник. Вместо этого в бинарник включается только ссылка на библиотеку.

Рассмотрим преимущества и недостатки статических и совместно используемых библиотек. Статические библиотеки делают программу более автономной: программа, скомпонован- ная со статической библиотекой может запускаться на любом компьютере, не требуя наличия этой библиотеки (она уже "внутри"бинарника). Программа, скомпонованная с динамической библиотекой, требует наличия этой библиотеки на том компьютере, где она запускается, поскольку в бинарнике не код, а ссылка на код библиотеки. Не смотря на такую зависимость, динамические библиотеки обладают двумя существенными преиму- ществами. Во-первых, бинарник, скомпонованный с совместно используемой библиотекой меньше размером, чем такой же бинарник, с подключенной к нему статической библиотекой (статически скомпонованный бинарник). Во-вторых, любая модернизация динамической

библиотеки, отражается на всех программах, использующих ее. Таким образом, если неко- торую библиотеку foo используют 10 программ, то исправление какой-нибудь ошибки в foo или любое другое улучшение библиотеки автоматически улучшает все программы, которые используют эту библиотеку. Именно поэтому динамические библиотеки называют совмест- но используемыми. Чтобы применить изменения, внесенные в статическую библиотеку, нужно пересобрать все 10 программ.

В Linux статические библиотеки обычно имеют расширение .a (Archive), а совместно используемые библиотеки имеют расширение .so (Shared Object). Хранятся библиотеки, как правило, в каталогах /lib и /usr/lib. В случае иного расположения (относится только к совместно используемым библиотекам), приходится явно указать путь, чтобы программа запустилась[2].

## Резидентное приложение с динамической библиотекой

В динамическую библиотеку вынесена функция, отвечающая за взаимодействие с системой. При компиляции, отдельно собирается библиотека, и отдельно исполняемый файл.

user@host$ g++ -o libparse.so -shared -fPIC -std=c++14 parse.cpp user@host$ g++ main.cpp -L. -lparse -o netmonitor

Теперь простой запуск приложения приведёт к ошибке, т.к. система ожидает наличия файла библиотеки в строго определённом месте.

user@host$ ./netmonitor

./netmonitor: error while loading shared libraries: libparse.so: cannot open shared user@host$

Отсутствие библиотеки можно легко обнаружить при запуске утилиты ldd (строка 2, листинг 6).

Листинг 6: Демонстрация работы программы ldd для приложения, использующего дина- мическую библиотеку

l inux *−*vdso . so . 1 => ( 0 x 00007 f f c 495 ca 000 ) l i b p a r s e . so => not found

l i b s t d c ++.so . 6 => / usr / l i b /x86\_64*−*linux *−*gnu/ l i b s t d c ++.so . 6 ( 0 x 00007 f ee 33 a 20000 )

l i b c . so . 6 => / l i b /x86\_64*−*linux *−*gnu/ l i b c . so . 6 ( 0 x 00007 f ee 33656000 ) l ibm . so . 6 => / l i b /x86\_64*−*linux *−*gnu/ l ibm . so . 6 ( 0 x 00007 f ee 3334 d 000 )

1

2

3

4

5

6

/ l i b 6 4 / ld*−*linux *−*x86 *−*64. so . 2 ( 0 x 0000561 a 68 f 06000 )

l i bg c c \_ s . so . 1 => / l i b /x86\_64*−*linux *−*gnu/ l i bg c c \_ s . so . 1 ( 0 x 00007 f ee 33137000 )

7

Эту проблему можно обойти если явным образом перед запуском программы передать путь к библиотеке через параметры

user@host$ LD\_LIBRARY\_PATH=. ./netmonitor enp2s0 enp2s0:

Receive 9686554269 bytes (6645253 packets)

Transmit 645757402 bytes (3625776 packets) user@host$

Из результатов анализа распределения памяти можно сделать вывод о том, что при запуске программы ей выделяется свободное место в памяти, в следствии чего адреса, по которым располагаются точки входа или подключения меняются при повторных запусках, однако, состав, порядок и смещения загружаемых модулей относительно начального адреса в выделенной памяти не изменяется.

# Процесс загрузки приложений Windows

## Выполнение ЕХЕ-модуля

При запуске ЕХЕ-файла (сокр. англ. executable — исполнимый) загрузчик операционной системы создает для его процесса виртуальное адресное пространство и проецирует на него исполняемый модуль. Далее загрузчик анализирует раздел импорта и пытается спроецировать все необходимые DLL на адресное пространство процесса.

Поскольку в разделе импорта указано только имя DLL (без пути), загрузчику приходит- ся самому искать ее на дисковых устройствах в компьютере пользователя. Поиск DLL осуществляется в следующей последовательности.

* + - Каталог, содержащий ЕХЕ-файл.
    - Текущий каталог процесса.
    - Системный каталог Windows
    - Основной каталог Windows
    - Каталоги, указанные в переменной окружения PATH.

Проецируя DLL-модули на адресное пространство, загрузчик проверяет в каждом из них раздел импорта. Если у DLL есть раздел импорта (что обычно бывает), загрузчик проецирует следующий DLL-модуль. При этом загрузчик ведет учет загружаемых DLL и проецирует их только один раз, даже если загрузки этих DLL требуют и другие модули.

Если найти файл DLL не удается, загрузчик выводит сообщение об ошибке.

Найдя и спроецировав на адресное пространство процесса все необходимые DLL-модули, загрузчик настраивает ссылки на импортируемые идентификаторы. Для этого он вновь просматривает разделы импорта в каждом модуле, проверяя наличие указанного иденти- фикатора в соответствующей DLL.

Если идентификатор не найден, то это заканчивается выводом сообщения об ошибке, если же идентификатор найден, загрузчик отыскивает его RVA и прибавляет к виртуальному адресу, по которому данная DLL размещена в адресном пространстве процесса, а затем сохраняет полученный виртуальный адрес в разделе импорта EXE-модуля. И с этого момента ссылка в коде на импортируемый идентификатор приводит к выборке его адреса из раздела импорта вызывающего модуля, открывая таким образом доступ к импортируемой переменной, функции или функции-члену C++ класса. После этого динамические связи установлены, первичный поток процесса начал выполняться, и приложение начинает

работать[4].

Загрузка всех DLL и настройка ссылок занимает какое-то время, но, поскольку такие операции выполняются лишь при запуске процесса, на производительности приложения это не сказывается Тем не менее, для многих программ подобная задержка при инициализации неприемлема. Чтобы сократить время загрузки приложения, можно модифицировать базовые адреса EXE- и DLL-модулей и провести их (модулей) связывание.

## Динамически подключаемые библиотеки

Динамические библиотеки (dynamic-link libraries, DLL) – краеугольный камень операцион- ной системы Windows, начиная с самой первой ec версии. В DLL содержатся все функции Windows API. Три самые важные DLL:

* + - Kernel32.dll – управление памятью, процессами и потоками
    - User32.dll – поддержка пользовательского интерфейса, в том числе функции, связан- ные с созданием окон и передачей сообщений
    - GDI32.dll – графика и вывод текста.

В Windows есть и другие DLL, функции которых предназначены для более специализи- рованных задач. Например, в AdvAPI32.dll содержатся функции для защиты объектов, работы с реестром и регистрации событий, в ComDlg32.dll – стандартные диалоговые окна (вроде File Open и File Save), a ComCrl32.dll поддерживает стандартные элементы управления.

Вот основные причины, по которым инструмент DLL получил такую популярность:

* + - **Расширение функциональности приложения**. DLL можно загружать в адресное пространство процесса динамически, что позволяет приложению, определив, какие действия от него требуются, подгружать нужный код. Поэтому одна компания, создав какое-то приложение, может предусмотреть расширение его функциональности за счет DLL от других компаний.
    - **Возможность использования разных языков программирования**. У разра- ботчика есть выбор, на каком языке писать ту или иную часть приложения. Поль- зовательский интерфейс приложения можно создавать на Microsoft Visual Basic, а прикладную реализовать на С++. Программа на Visual Basic может загружать DLL, написанные на С++, Коболе, Фортране и др.
    - **Более простое управление проектом**. Если в процессе разработки программного

продукта отдельные его модули создаются разными группами, то при использовании DLL таким проектом управлять гораздо проще. Однако конечная версия приложения должна включать как можно меньше файлов.

* + - **Экономия памяти**. Если одну и ту же DLL использует несколько приложений, в оперативной памяти может храниться только один ее экземпляр, доступный этим приложениям. Пример – DLL-версия библиотеки С/С++. Эту библиотеку используют многие приложения. Если всех их скомпоновать со статически подключаемой версией этой библиотеки, то код таких функций, как sprintf, strcpy, malloc и др., будет многократно дублироваться в памяти. Но если они компонуются с DLL-версией библиотеки С/С++, в памяти будет присутствовать лишь одна копия кода этих функций, что позволит гораздо эффективнее использовать оперативную память.
    - **Разделение ресурсов**. DLL могут содержать такие ресурсы, как шаблоны диалого- вых окон, строки, значки и битовые карты (растровые изображения). Эти ресурсы доступны любым программам.
    - **Упрощение локализации**. DLL нередко применяются для локализации приложе- ний. Например, приложение, содержащее только код без всяких компонентов поль- зовательского интерфейса, может загружать DLL с компонентами локализованного интерфейса
    - Решение проблем, связанных с **особенностями различных платформ**. В разных версиях Windows содержатся разные наборы функций. Зачастую разработчикам нужны новые функции, существующие в той версии системы, которой они пользуются. Если Windows пользователя не поддерживает эти функции, ему не удастся запустить такое приложение: загрузчик попросту откажется его запускать. Но если эти функции будут находиться в отдельной DLL, можно будет загрузить программу даже в более ранних версиях Windows.
    - **Реализация специфических возможностей**. Определенная функциональность в Windows доступна только при использовании DLL Например, отдельные виды ловушек (устанавливаемых вызовом SetWindowsHookEx и SetWinEventHook) можно задействовать при том условии, что функция уведомления ловушки размещена в DLL. Кроме того, расширение функциональности оболочки Windows возможно лишь за счет создания СОМ-объектов, существование которых допустимо только в DLL. Это же относится и к загружаемым Web-браузером ActiveX-элементам, позволяющим создавать Web-страницы с более богатой функциональностью.

Зачастую создать DLL проще, чем приложение, потому что она является лишь набором автономных функций, пригодных для использования любой программой, причем в DLL

обычно нет кода, предназначенного для обработки циклов выборки сообщений или создания окон. DLL представляет собой набор модулей исходного кода, в каждом из которых содер- жится определенное число функций, вызываемых приложением (исполняемым файлом) или другими DLL. Файлы с исходным кодом компилируются и компонуются так же, как и при создании ЕХЕ-файла. Но, создавая DLL, нужно указывать компоновщику ключ /DLL. Тогда компоновщик записывает в конечный файл информацию, по которой загрузчик операционной системы определяет, что данный файл – DLL, а не приложение

Чтобы приложение (или другая DLL) могло вызывать функции, содержащиеся в DLL, образ ее файла нужно сначала спроецировать на адресное пространство вызывающего процесса. Это достигается либо за счёт неявного связывания при загрузке, либо за счет явного – в период выполнения.

Как только DLL спроецирована на адресное пространство вызывающего процесса, ее функции доступны всем потокам этого процесса. Фактически библиотеки при этом теряют почти всю индивидуальность: для потоков код и данные DLL – просто дополнительные код и данные, оказавшиеся в адресном пространстве процесса. Когда поток вызывает из DLL какую-то функцию, та считывает свои параметры из стека потока и размещает в этом стеке собственные локальные переменные. Кроме того, любые созданные кодом DLL объекты принадлежат вызывающему потоку или процессу – DLL ничем пе владеет.

Например, если DLL-функция вызывает VirtualAlloc, резервируется регион в адресном пространстве того процесса, которому принадлежит поток, обратившийся к DLL-функции. Если DLL будет выгружена из адресного пространства процесса, зарезервированный регион не освободится, так как система не фиксирует того, что регион зарезервирован DLL-функций. Считается, что он принадлежит процессу и поэтому освободится, только если поток этого процесса вызовет VirtualFree или завершится сам процесс.

## Реализация резидентного приложения

Как и в случае с Linux, приложение отображает количество переданной и полученной информации сетевым интерфейсом. Тут интересно, что используемую в процессе функцию мы получаем непосредственно во время работы (стр. 20 - 22, листинг 7).

Листинг 7: Утилита сбора информации о трафике на сетевых интерфейсах (src/ELF/win/main.cpp)

#i nc l ude <iostream>

#i nc l ude <windows . h>

#i nc l ude <i ph l p api . h>

1

2

3

4

5 bool Get I f Table (PMIB\_IFTABLE *\**m\_pTable) ;

6

7 i n t main ( i n t argc , char *\** argv [ ] ) {

8 PMIB\_IFTABLE m\_pTable = NULL;

9

10 i f ( Get I f Table(&m\_pTable) == f a l s e ) {

11 re turn 1 ;

12 }

13

14 // Обход списка с е те вых интерфейс ов

15 f o r (UINT i = 0 ; i < m\_pTable*−*>dwNumEntries ; i ++) {

16 MIB\_IFROW Row = m\_pTable*−*>t ab l e [ i ] ;

17 char sz Descr [ MAXLEN\_IFDESCR ] ;

18 memcpy( sz Descr , Row. bDescr , Row. dwDescrLen ) ;

19 sz Descr [ Row. dwDescrLen ] = 0 ;

20

21 // Выв од с обранной информации

22 std : : cout << sz Descr << " : " << std : : endl ;

23 std : : cout << "\ t Received : " << Row. dw InOctets

24 << " , Sent : " << Row. dwOutOctets << std : : endl ;

25 std : : cout << std : : endl ;

26 }

27

28 // Завершение работы

29 d e l e te ( m\_pTable) ;

30 char a = getchar ( ) ;

31 re turn 0 ;

32 }

33

34 bool Get I f Table (PMIB\_IFTABLE *\**m\_pTable) {

35 // Тип указателя на функцию Get I f Table

36 typedef DWORD( \_ s tdc a l l *\** TGetIf Table ) (

37 MIB\_IFTABLE *\** p If Table , // Буфер таблицы интерфейс ов

38 ULONG *\** pdwSize , // Размер буфера

39 BOOL bOrder ) ; // Сортировать таблицу?

40

|  |  |  |
| --- | --- | --- |
| 41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61 |  | // Пыта емся HINSTANCE i p  i p hl pap i = L i f ( ! i ph l pap std : : c e r r  re turn f a l  }  // Получаем TGetIf Table p GetIf Table  // Получили DWORD m\_dw p GetIf Table (  *\**m\_pTable =  i f ( p GetIf Ta  {  std : : c e r r d e l e te *\**m re turn f a l |
| 62 |  | } |
| 63 |  |  |
| 64 |  | re turn true ; |
| 65 | } |  |

Результат выполнения программы представлен на рисунке 4.

подгрузить i p hl p api . d l l

h l papi ;

oad Library (L" i p hl p ap i . d l l " ) ; i ) {

<< " i phl papi . d l l not supported " << std : : endl ;

s e ;

адре с функции

p GetIf Table ;

= ( TGetIf Table ) GetProcAddress ( i phl papi , " Get I f Table " ) ;

тре буемый размер буфера

Adapters = 0 ;

*\** m\_pTable , &m\_dwAdapters , TRUE) ;

new MIB\_IFTABLE[ m\_dwAdapters ] ;

ble ( *\** m\_pTable , &m\_dwAdapters , TRUE) != ERROR\_SUCCESS)

<< " Error while Get I f Table " << std : : endl ;

\_pTable ; s e ;

Заполнение таблицы с информацией об интерфейсах (стр. 37, листинг 7) вынесена в отдельную функцию (стр.34, листинг 7). В приложении производится большое количество обращений к различным DLL. Для упрощения дальнейшего анализа вынесем функцию в отдельную динамическую библиотеку (myDllLib) и продолжим анализ.

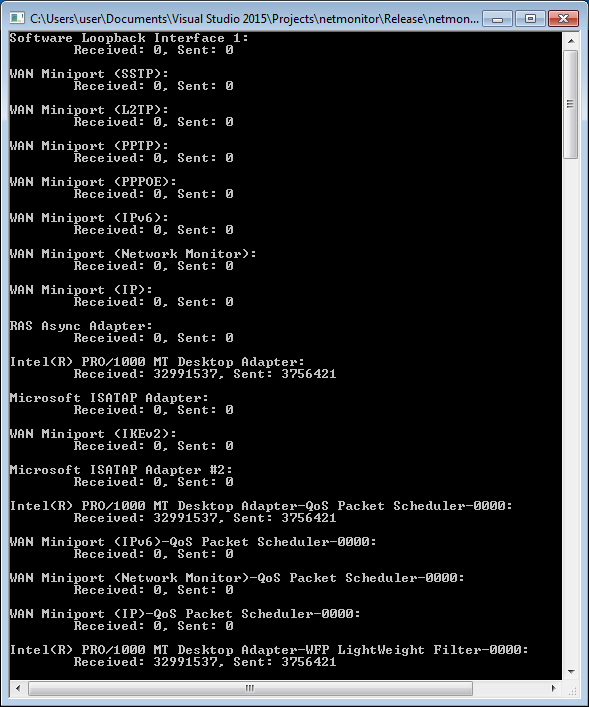


Рис. 4: Исполнение программы в среде Windows

## Анализ исполнения приложения

Утилита API Monitor может декодировать параметры функций и возвращаемые значе- ния, чтобы представить их в понятном формате, а также отобразить точки обращения разработанной динамической myDllLib.dll

Отладчик OllyDbg позволяет осуществить разбор пользовательского режима (ring 3). При запуске программы, изначальной точкой входа является функция \_tmainCRTStartup. Ниже приведена команда вызова данной функции:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| # 14013 | Time of Day  12:11:01.054 | PM | Thread | 1 | Module API myDllLib.dll | Return Val DecodePointe |
| 14014 | 12:11:01.054 | PM |  | 1 | myDllLib.dll | DecodePointe |
| 14015 | 12:11:01.054 | PM |  | 1 | myDllLib.dll | EncodePointe |
| 14016 | 12:11:01.054 | PM |  | 1 | myDllLib.dll | DecodePointe |
| 14017 | 12:11:01.054 | PM |  | 1 | myDllLib.dll | EncodePointe |
| # 14020 | Time of Day  12:11:01.054 | PM | Thread | 1 | Module API myDllLib.dll | Return Val DecodePointe |
| 14021 | 12:11:01.054 | PM |  | 1 | myDllLib.dll | DecodePointe |
| 14022 | 12:11:01.054 | PM |  | 1 | myDllLib.dll | EncodePointe |
| 14023 | 12:11:01.054 | PM |  | 1 | myDllLib.dll | DecodePointe |
| 14024 | 12:11:01.054 | PM |  | 1 | myDllLib.dll | EncodePointe |
| 14025 | 12:11:01.054 | PM |  | 1 | myDllLib.dll | DecodePointe |
| 14026 | 12:11:01.054 | PM |  | 1 | myDllLib.dll | DecodePointe |

CPU Disasm Address Hex dump Command Comments

01271BA3 |. E8 ADF4FFFF CALL 01271055; [ security\_init\_cookie 01271BA8 |. E8 73FCFFFF CALL tmainCRTStartup; [ tmainCRTStartup

Сама функция \_tmainCRTStartup находится по адресу 012F1820:

CPU Disasm

Address

012F1820

Hex dump Command

Comments

/$

55 PUSH EBP; INT myDllTest. tmainCRTStartup(void)

Адрес начала подключения динамической библиотеки находится по адресу 012F1820:

CPU Disasm

Address

012F1820

012F1821

012F1823

012F1825

012F182A

Hex

/$

|.

|.

|.

|.

dump

55

8BEC

Command Comments

PUSH EBP; INT myDllTest. tmainCRTStartup(void) MOV EBP,ESP

6A

68

68

FE

D06E2F01 7D102F01

PUSH

PUSH PUSH

-2

OFFSET 012F6ED0 012F107D

|  |  |  |  |
| --- | --- | --- | --- |
| 012F182F | |. | 64:A1 0000000 | MOV EAX,DWORD PTR FS:[0] |
| 012F1835 | |. | 50 | PUSH EAX |
| 012F1836 | |. | 83C4 E4 | ADD ESP,-1C |
| 012F1839 | |. | 53 | PUSH EBX |
| 012F183A | |. | 56 | PUSH ESI |
| 012F183B | |. | 57 | PUSH EDI |
| 012F183C | |. | A1 24802F01 | MOV EAX,DWORD PTR DS:[ security\_cookie] |
| 012F1841 | |. | 3145 F8 | XOR DWORD PTR SS:[EBP-8],EAX |
| 012F1844 | |. | 33C5 | XOR EAX,EBP |
| 012F1846 | |. | 50 | PUSH EAX |
| 012F1847 | |. | 8D45 F0 | LEA EAX,[EBP-10] |
| 012F184A | |. | 64:A3 0000000 | MOV DWORD PTR FS:[0],EAX |
| 012F1850 | |. | 8965 E8 | MOV DWORD PTR SS:[EBP-18],ESP |
| 012F1853 | |. | C745 FC 00000 | MOV DWORD PTR SS:[EBP-4],0 |
| 012F185A | |. | C745 E4 00000 | MOV DWORD PTR SS:[EBP-1C],0 |

Адреса точки входа и подключения библиотек не изменились.

Process Monitor является усовершенствованным инструментом отслеживания для Windows, который в режиме реального времени отображает активность файловой системы, реестра, а также процессов и потоков. Проверим адреса подключения динамической библиотеки при помощи утилиты Process Monitor.

Description:

Company: Name: Version: Path:

myDllTest.exe

C:\Projects\myDllLib\Debug\myDllTest.exe

Command PID:

Line:

3380

myDllTest.exe

Parent PID: 2436

Session ID: 1

User: sba-PC\sba

Auth ID: 00000000:0001b9ac

Architecture: 32-bit

Virtualized: False

Integrity: Обязательная метка\Средний обязательный уровень Started: 22.03.2016 2:02:47

Ended: 22.03.2016 2:04:46

Modules:

myDllTest.exe 0x250000 0x1c000 C:\Projects\myDllLib\Debu 22.03.2016 0:45:34

MSVCR120D.dll 0x72e10000 0x1bf000 C:\Windows\SysWOW64\MS Microsoft Corporation 12.00.21005.1 built by: REL 05.10.2013 5: myDllLib.dll 0x73be0000 0x1b000 C:\Projects\myDllLib\Deb 22.03.2016 0:45:33

wow64win.dll 0x74b40000 0x5c000 C:\Windows\SYSTEM32\wow6 Microsoft Corporation 6.1.7601.19018 (win7sp1\_gdr.150928-1507) wow64.dll 0x74ba0000 0x3f000 C:\Windows\SYSTEM32\wow64.d Microsoft Corporation 6.1.7601.19018

(win7sp1\_gdr.150928-1507) 29.09.2015 6:12:08

wow64cpu.dll 0x74c10000 0x8000 C:\Windows\SYSTEM32\wow64 Corporation 6.1.7601.19018 (win7sp1\_gdr.150928-1507)

29.09.2015 6:12:09

KERNELBASE.dll 0x75220000 0x47000 C:\Windows\syswow64\KE Corporation 6.1.7601.18015 (win7sp1\_gdr.121129-1432)

29.09.2015 6:00:36

kernel32.dll 0x75350000 0x110000 C:\Windows\syswow64\ker Corporation 6.1.7601.18015 (win7sp1\_gdr.121129-1432)

29.09.2015 6:00:35

ntdll.dll 0x77100000 0x1a9000 C:\Windows\SYSTEM32\ntdll.

Microsoft Corporation 6.1.7600.16385

(win7\_rtm.090713-1255) 29.09.2015 6:07:47

ntdll.dll 0x772e0000 0x180000 C:\Windows\SysWOW64\ntdll.

Microsoft Corporation 6.1.7600.16385

(win7\_rtm.090713-1255) 29.09.2015 5:57:52

Как видно из вывода, тестовая динамическая библиотека myDllLib.dll была подключена по адресу 0x73be0000 и имеет размер 0x1b000.

Выделение памяти в ОС Windows и Linux схожи: и в том и в другом случае программе выделяется произвольная доступная область памяти, вследствие чего адреса подключения модулей различаются, однако порядок подключения модулей, их смещения относительно выделенной области памяти, а так же виртуальный адрес точки входа остается неизменным.

# Заключение

В данной работе были рассмотрены механизмы загрузки исполняемых приложений в операционных системах семейства Windows и Linux.

Современные подходы к разработке больших приложений предполагают использование динамических библиотек, обладающих своими особенностями.

Достоинства:

* + - экономия памяти за счёт использования одной библиотеки несколькими процессами;
    - разработка различных модулей на различных языках;
    - возможность исправления ошибок (достаточно заменить файл библиотеки и переза- пустить работающие программы).

Недостатки:

* + - возможность нарушения API (при внесении изменений в библиотеку существующие программы могут перестать работать);
    - конфликт версий динамических библиотек (разные программы могут ожидать разные версии библиотек);
    - доступность одинаковых функций по одинаковым адресам в разных процессах (упро- щает эксплуатацию уязвимостей).

Главной особенностью динамических библиотек является ускорение процесса разработки и предоставление хорошо протестированных решений, что является важнейшими задачами в индустрии.

# Список литературы

[1] Лав Р. Linux. Системное программирование. 2-е изд. – СПб.: Питер, 2014 – 448 стр. [2] Рэймонд Э.С.. Искусство программирования для Unix. – М.: Вильямс, 2005 – 544 стр. [3] Касперски К. Секреты поваров компьютерной кухни или ПК: решение проблем – M.:

BHV, 2003 – 560 стр.

[4] Харт Дж. Системное программирование в среде Windows. – M.: 2005