Визитка

ИГОРЬ ШТОМПЕЛЬ, инженер, системный администратор. Сфера профессиональных интересов – свободное программное обеспечение

Основы AMD64

Исследуем с помощью свободного ПО

Техническое развитие не стоит на месте, и вот уже 64-битные платформы становятся повседневностью. Но неизменно одно — желание заглянуть внутрь: посмотреть, как решение устроено, как с ним можно работать. Вот и мы совершим небольшое «погружение» в AMD64 с помощью теории и СПО

Описание архитектуры

Прежде чем перейти к практике, необходимо ознакомиться с основными особенностями архитектуры AMD64. Базовым подробнейшим источником о ней является документация для разработчиков, подготовленная компанией AMD и доступная на официальном сайте [1]. Сначала нам понадобится первое из шести руководств под общим заглавием «AMD64 Architecture Programmer's Manual» – «Volume1: Application Programming» [2].

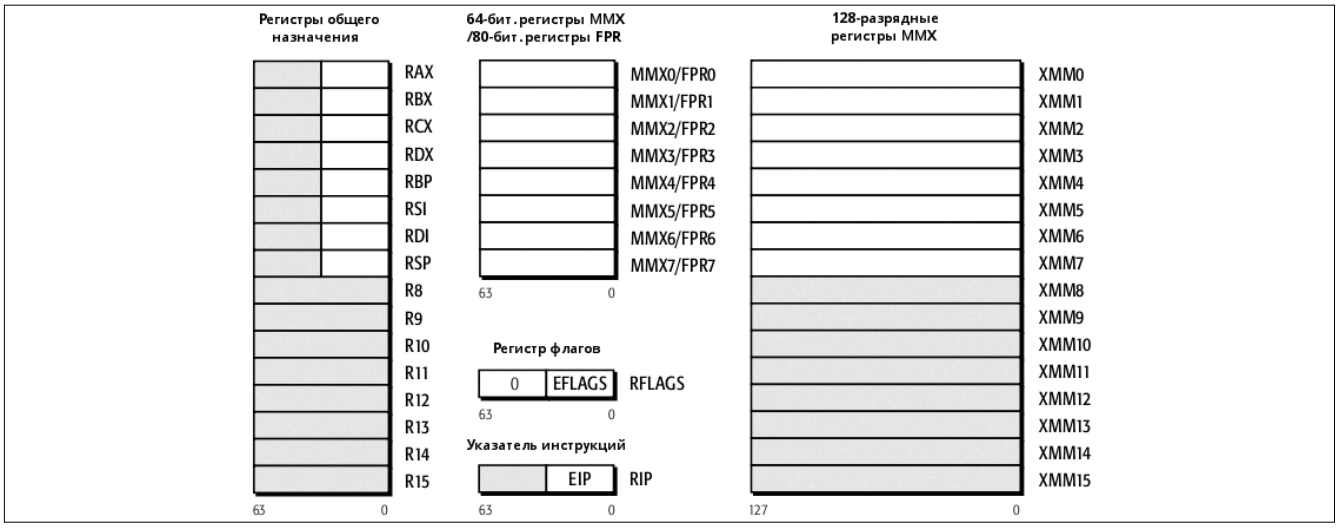
Переходим к рассмотрению новшеств платформы AMD64. Первое из них то, что по сравнению с 32-битной архитектурой появился специальный режим работы – Long Mode (длинный режим). Рассмотрим подробнее режимы работы процессоров архитектуры AMD64, опираясь на таблицу 1.

Итак, два режима работы. Как видно из таблицы 1, именно Наследуемый режим позволяет производить установку 32-битной ОС. В свою очередь в данном режиме недоступны 64-битные регистры, а также новые возможности в области адресации (см. колонки «По умолчанию», «Расширения регистров» и «Типичный» таблицы 1).

Длинный режим имеет два подрежима работы – 64-битный и режим совместимости. С первым мы и будем в основном иметь дело далее (из всех режимов и подрежимов, как видно из таблицы, интересующая нас 64-разрядность обеспечивается именно Long Mode), а второй необходим для обеспечения совместимости с 32-битными приложениями. Обратите внимание, что если Наследуемый режим используется для работы процессора исключительно в 32-битном режиме, то Режим совместимости необходим для переключения между 32 и 64 битами в 64-битном режиме работы.

Кроме того, вы не обнаружите в длинном режиме поддержки режима Virtual-8086, а также реального режима. Он, если можно так выразиться, полностью сконцентрирован на выполнении 64-битных приложений, а также при необходимости обеспечивает совместимость и позволяет

Рисунок 1. Регистры платформы AMD64



запускать 32-битные. А раз так, для его задействования необходимо устанавливать исключительно 64-битную операционную систему (благо это давно уже не проблема как для пользователей Linux, так и для пользователей других популярных операционных систем) или в крайнем случае использовать эмуляцию (например, через QEMU).

Поскольку в работе процессора большое значение имеют регистры, то нам необходимо познакомиться с последними поближе. Далее мы будем исследовать регистры процессоров AMD64 общего назначения, которые для наглядности представили вместе с их разрядами в таблице 2. В подготовке таблицы мы опирались на первую часть официального руководства от AMD.

Другим новшеством стали изменения в регистрах процессоров платформы (см. рис. 1):

- шестнадцать регистров общего назначения (General Purpose Registers, GPRs);
- восемь 64-битных регистров MMX и данных с плавающей точкой (Floating-Point Registers, FPR);
- шестнадцать 128-битных регистров XMM;
- регистр флагов EFLAGS и указатель инструкций EIP.

Даже поверхностный взгляд на рисунок позволяет понять, что RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP – это расширенные до 64 бит регистры общего назначения EAX, EBX, ECX и т.д. А вот регистры с R8 по R15 – это новые восемь регистров общего назначения, доступные в режиме x86-64. Таким образом, все регистры общего назначения являются 64-битными.

Расширение регистров затронуло и XMM (они предназначены для передачи параметров типа float и double) – разработчики добавили восемь дополнительных регистров XMM8-XMM15, которые доступны только в 64-битном режиме.

Обратите внимание, что для доступа к 64-битным регистрам общего назначения используется префикс REX (R). Его назначение подробно описано в третьей части документации от AMD – «General-Purpose and System Instructions» (см. раздел 1.2.7 «REX Prefixes» на с.11). По сути, префиксы – дополнительные четыре бита к формату инструкций, которые обеспечивают доступ к 64-битным регистрам с помощью специальных кодов (см. таблицу 2) [3].

Рассмотрим регистры общего назначения подробнее (см. таблицу 3). Первые четыре регистра используются в качестве переменных процессора при выполнении команд (например, хранения данных или аргументов), а также для целей адресации. Обратите внимание, что названия первых четырех регистров обусловлены назначением, которое они выполняют. Итак:

RAX (Accumulator) – аккумулятор, который ориентирован на хранение результата действий, полученного в ходе операции над двумя операндами;

RBX (Base) – регистр базы, который необходим для хранения адреса (базового) операнда, который находится в памяти;

RCX (Counter) – счетчик, который содержит количество повторений циклов, строковых операций, а также сдвигов;

Таблица 1. Режимы работы процессоров AMD64

Рабочий режим		Требование к операционной системе	Требование к перекомпиляции приложений	По умолчанию		Расширения регистров	Типичный
				Размер адреса (биты)	Размер операнда (биты)		Размер регистра общего назначения
Длинный режим (Long Mode)	64-битный режим	64-битная ОС	Да	64	32	Да	64
	Режим совместимости		Нет	32		Нет	32
				16	16		16
Наследуемый режим (Legacy Mode)	Защищенный режим	Наследуемая 32-битная ОС	Нет	32	32	Нет	32
	Режим Virtual-8086			16	16		
	Реальный режим	Наследуемая 16-битная ОС		16	16		16

Таблица 2. Префиксы REX

Префиксы	Мнемоники и биты			Нех-Коды
	Мнемоники	Позиция бита	Описание	
REX	REX.W	3	Если значение 0, то регистры интерпретируются, как 32-битные. Если 1, то как 64-битные – 1 бит	40-4F
	REX.R	2	Расширение для поля инструкции reg, входящего в ModRM, позволяющее получить доступ к 64-битным регистрам , – 1 бит	
	REX.X	1	Расширение для поля инструкции index, входящего в SIB, позволяющее получить доступ к 64-битным регистрам, – 1 бит	
	REX.B	0	Расширение для полей инструкции r/m (ModRM), base (SIB), опкода, позволяющее получить доступ к 64-битным регистрам, – 1 бит	

RDX (Data) – регистр данных, позволяющий хранить часть результата, не уместившуюся в аккумулятор;
RSI (Source Index) – индекс источника, т. е. указатель на адрес источника, откуда необходимо прочитать данные (текущий адрес);
RDI (Detination Index) – индекс приемника, т. е. указатель на приемник, в который необходимо записать данные (текущий адрес);
RBP (Base Pointer) – регистр базы, указатель, необходимый для произвольного доступа к данным стека;
RSP (Stack Pointer) – указатель вершины стека;
R8-R15 – согласно упоминавшейся документации AMD (ч. 1, с. 32 – далее будем ссылаться на документациию AMD следующим образом: ДА) явного назначения у этих регистров нет. За исключением R11, который может содержать значения SYSCALL (инструкция, представляющая «быстрый» системный вызов – например, доступ к выполнению процедур ядра (CPL0, current privilege level) для программы, выполняющейся на пользовательском уровне, – CPL3 (см. ДА, ч. 1, с. 67)/SYSRET (при использовании этой инструкции контроль от операционной системы передается программе, которая сгенерировала системный вызов).

Таблица 3. Регистры общего назначения в 64-разрядном режиме

Регистры	Регистры и разряды			
RAX		EAX		
	63-32	31-16	AX 15-0	
			AH 15-8	AL 7-0
RBX		EBX		
	63-32	31-16	BX 15-0	
			BH 15-8	BL 7-0
RCX		ECX		
	63-32	31-16	CX 15-0	
			CH 15-8	CL 7-0
RDX		EDX		
	63-32	31-16	DX 15-0	
			DH 15-8	DL 7-0
RSI		ESI		
	63-32	31-16	SI 15-0	
				SIL 7-0
RDI		EDI		
	63-32	31-16	DI 15-0	
				DIL 7-0
RBP		EBP		
	63-32	31-16	BP 15-0	
				BPL 7-0
RSP		ESP		
	63-32	31-16	SP 15-0	
				SPL 7-0
R8-R15		R8D-R15D		
	63-32	31-16	R8W-R15W	
				R8B-R15B

Кроме регистров базового назначения, мы будем затрагивать и работу со следующими регистрами:
RIP – указатель инструкции, который содержит адрес следующей команды;
RFLAGS – регистр битовых флагов, которые «отслеживают» состояние процессора (например, разрешены ли прерывания, был ли перенос и т.д.).

Добавим к сказанному, что RSI и RDI задействуются при строковых операциях, а RBP и RSP – при взаимодействии со стеком. В свою очередь, разрядность всех описанных регистров была представлена на рис. 1.

Для получения более подробной информации о регистрах процессоров можно обратиться к работам С.В. Зубкова, В.И. Юрова и Д. Эриксона [4].

Предложенные книги отличаются подходами. Первые две сразу погружают читателя в большой массив подробностей и деталей. Третья книга, второе издание которой хотелось бы отметить особо, предлагает другой подход. Дается небольшое количество информации, а затем на практических примерах объясняются особенности работы. Поскольку в своем названии книга Эриксона имеет слова «хакинг», «эксплойт», хотелось бы надеяться, что информация из нее не будет использоваться для осуществления злоумышленных действий, а послужит источником необходимых сведений для более глубоко понимания безопасности Linux-систем, так нужной в том числе и системным администраторам.

Необходимое программное обеспечение

После небольшого теоретического введения перейдем к практической части. Для начала подготовим инструменты, которые нам понадобятся.

Все действия мы будем осуществлять в операционной системе Linux. В моем случае это Linux Mint 10 (основан на Ubuntu 10.10) с кодовым именем Julia. Дистрибутив не лишен «неприятных сюрпризов»: например, уникальная разработка проекта – «Менеджер программ» с рейтингом программ, основанным на отзывах пользователей, – после большого количества установок/удалений программ «отказался» производить инсталляцию новых приложений. Но это не представляет особой проблемы, так как есть Synaptic, да и apt-get не будем забывать. Справедливости ради необходимо сказать, что при повторной переустановке проблема не повторилась, и система функционирует хорошо. В целом Linux Mint произвел хорошее впечатление, и, думаю, для рабочих столов это вполне подходящее решение.

Помимо операционной системы, нам понадобится компилятор GCC (GNU Compuler Collection – сразу после установки операционной системы. В моем случае по умолчанию доступна версия 4.4.4), а также отладчик GDB (доступна версия 7.2) [5].

GCC – набор компиляторов (для языков C/C++, Java, Objective C, Fortran, Ada и других), входящий в состав GNU toolchain (набор инструментов для компиляции и генерации исполняемого кода). Причем поддержка языков постоянно расширяется. Так, например, вначале было объявлено о включении в GCC-фронтенда «gccgo» для языка программирования Go [6]. Для более подробного изучения работы со свободным набором компиляторов можно ознакомиться

с официальной документацией, доступной на сайте GCC, а также с книгой А. Гриффитса [7].

Помимо компилятора, как мы сказали выше, нам понадобится отладчик. Для наших целей вполне подойдет GNU Debbuger. Он позволяет производить отладку программ, написанных на таких языках программирования, как C/C++, Fortran, FreePascal, Ada, FreeBASIC. GDB доступен для большого количества платформ, в том числе x86, x86-64 (что нам в данном случае и надо), IA-64, ARM, Alpha. Как и GCC, GDB является свободным программным обеспечением.

Отладчик не имеет графического интерфейса. Сторонние разработчики создали ряд интерфейсов, призванных повысить удобство работы: DDD, cgdb и другие. Мы же будем использовать стандартный для GDB интерфейс – консоли.

Более подробно с возможностями отладчика можно познакомиться в официальной документации, доступной также в неофициальном переводе на русский язык [8].

Кроме того, после установки Linux Mint (как, впрочем, и во многих дистрибутивах) доступен набор инструментов GNU Binary Utilities (Binutils). В первую очередь нам будут интересны objdump и readelf. Кроме того, сторонние разработчики создали для objdump графическую оболочку, получившую название Dissy [9].

Установить последнюю можно так:

```
$ sudo apt-get install dissy
```

Другой отладчик, о котором хотелось бы сказать, – это Evan's Debugger (EDB). Если GCC – отладчик исходного кода (т.е. подразумевается, что этот самый исходный код у вас есть), то EDB – отладчик/дизассемблер, который ориентирован на использование в качестве инструмента реверсивной инженерии (с графическим интерфейсом на Qt4.5). Название программа получила по имени разработчика – Эвана Терена (Evan Teran). Любопытно то, как он описывает мотивы, побудившие его начать разработку отладчика. Работая в Windows, Теран использовал в качестве отладчика OllyDbg, но в Linux он не смог найти эквивалента последнему. Поэтому задался целью создать отладчик с подобной функциональностью, но только для Linux. Кстати, ранее он работал в рамках проекта KDE над повышением скорости KCalc, а в текущее время сосредоточился на своих проектах.

Среди основных возможностей программы:

- поддержка платформ x86 и x86-64;
- интуитивно понятный интерфейс пользователя;
- поддержка стандартных операций отладки (установка точек останова, выполнение программы и т. д.);
- поддержка вкладок для окна дампа;
- поддержка плагинов (например, доступны плагины для просмотра переменных среды, управления точками останова, поиск «бинарных строк» – поиск строк в дампе, проверки обновлений и другие), кстати, ядро отладчика реализовано в виде плагина.

Кстати, другие инициативы Эвана Терена не менее интересны. Так, он работает над библиотекой edisassm, так как на определенном этапе возможностей libdisasm в качестве дизассемблерного движка для EDB ему оказалось недостаточно. По мнению разработчика, libdisasm – это «застойный» проект, который ориентирован только на i386. Поэтому Теран начал проект своей библиотеки edisassm (объектно-ориентированное API для дизассемблирования), которая позволит декодировать все инструкции i386/EMT64 (в том числе MMX и SSE – SSE, SSE2, SSE3, SSSE3). Начиная с версии 1.5.0 библиотека поддерживает работу с 64-битным кодом.

Другими интересными проектами Терана являются шестнадцатиричный редактор QHexView (на его основе организован вывод hex-дампа в окне EDB), незавершенная операционная система evanOS, Pretendo (NES-эмулятор), библиотека libunif, RPG Engine и другие [10].

Произведем установку EDB. Для этого скачаем с домашней страницы проекта (ссылка на нее приведена выше) архив с исходным кодом программы – debugger-0.9.16.tgz. Далее распакуйте его в папку, в которой вы хотите осуществить установку программы:

```
$ tar xfz debugger-0.9.16.tgz
```

Архив будет распакован в каталог debugger. Но перед установкой программы необходимо разрешить зависимости. На официальной странице проекта указана необходимость установки Qt4.5 и boost 1.35. В зависимости от версии Ubuntu или основанных на нем операционных систем может понадобиться установка следующих зависимостей (проверьте, что у вас уже установлено, и измените команду):

Таблица 4. Сводная информация об используемых инструментах

Название	Версия	Описание	Лицензия	Официальный сайт	Примечание
GNU Compiler Collection (GCC)	4.4.4	Набор компиляторов	С версии 4.2.2 – GNU GPLv3	http://gcc.gnu.org	
GNU Debbuger (GDB)	7.2	Отладчик	GNU GPLv3 (согласно анонсу в разделе новостей с официального сайта – с версии 7.0.1)	http://www.gnu.org/software/gdb	
GNU Binary Utilities	2.20	Набор инструментов для исследования объектного кода	GNU GPLv3	http://www.gnu.org/software/binutils	readelf, objdump
Evan's Debugger (EDB)	0.9.16	Отладчик	GNU GPLv2	http://www.codef00.com/projects.php	
pcalc	2	Калькулятор для программистов	GNU GPLv2	http://sourceforge.net/projects/pcalc	
Dissy	v9	Графическая оболочка для objdump	GNU GPLv2	http://code.google.com/p/dissy	

```
$ sudo apt-get install qt4-qmake libqt4-dev libboost-dev g++
```

Теперь перейдем в каталог debugger и дадим команду:

```
$ qmake
```

А после ее успешного выполнения:

```
$ sudo make install
```

После этого программа будет установлена, и для ее запуска достаточно дать команду:

```
$ edb
```

Кстати, для удобства запуска программы можно отредактировать главное меню, добавив в него пункт с EDB.

Для быстрого перевода чисел из одной системы счисления в другую удобно пользоваться калькулятором pcalc [11]. Установим его.

В качестве одной из зависимостей программа использует генератор лексических анализаторов Flex, который и необходимо установить (по умолчанию он не доступен после установки ни в Ubuntu 10.10, ни в Linux Mint 10):

```
$ sudo apt-get install flex
```

Перейдем непосредственно к установке pcalc. Для начала загрузим архив pcalc-2.tar.lzma со страницы проекта на Sourceforge.net, а затем скопируем этот файл в каталог, в котором будет осуществляться установка. Далее распакуем архив:

```
$ tar xJf pcalc-2.tar.lzma
```

Программа разархивируется в текущий каталог, создав каталог pcalc-2. Перейдем в него и дадим команду:

```
$ make
```

а затем:

```
$ sudo make install
```

Все, программа успешно установлена.

Для удобства мы составили сводную таблицу (см. таблицу 4), посвященную инструментам, которые будем использовать.

Рисунок 2. Вывод заголовка ELF-файла с помощью readelf

```
bgt@freedesk ~/Develop $ readelf -h first
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400410
  Start of program headers:          64 (bytes into file)
  Start of section headers:          4432 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 27
```

Исследуем

Для начала напишем небольшую программу на языке C. Например, такую:

```
#include <stdio.h>

int main ()
{
    int a, b, c;
    for (a=0; a<=3; a++)
    {
        printf ("a равно %d\n", a);
    }
    for (b=0; b<=3; b++)
    {
        printf ("b равно %d\n", b);
    }
    c = a + b;
    printf ("Сумма a и b равна %d\n", c);
    return 0;
}
```

В этой программе мы вначале даем указание компилятору включить информацию о стандартной библиотеке ввода-вывода (заголовочные файлы хранятся в каталоге /usr/include – в некоторых системах путь может быть другим). Это заголовочный файл стандартного ввода-вывода – stdio.h. Как известно, сам язык C не содержит каких-либо средств ввода-вывода, поэтому очень зависим от своей стандартной библиотеки, которая проходит процесс стандартизации вместе с языком.

Далее мы объявляем три целочисленных переменных – a, b и c. Затем организуем два цикла, каждый из которых выполняется до тех пор, пока переменная, использованная в цикле, меньше или равна 3. А затем присваиваем сумму переменных a и b переменной c и осуществляем вывод ее значения с помощью функции printf.

Оператор return 0; – это код возврата программы.

Теперь скомпилируем нашу программу для AMD x86-64. Дело в том, что нам понадобится воспользоваться специальной опцией компилятора GCC, которая входит в «набор» опций, определенных для платформ x86 и x86-64 (поскольку AMD64 может работать в режиме совместимости с x86, потребовалась новая функциональность компилятора, учитывающая возможности 64-битной платформы, в том числе в плане совместимости).

Кстати, подробнее с данными опциями можно ознакомиться в официальной документации компилятора или в книге А. Гриффитса [12].

Итак, для компиляции дадим следующую команду:

```
$ gcc -m64 -o first first.c
```

Разберем по порядку. Опция -m64 специфицирует длину данных типа int равной 32 битам, а данных типа long и указателей – 64 битам в 64-разрядной среде AMD64. Если эта опция будет использована на 32-битной платформе, то длина в 32 бита будет не только у данных типа int, но и у данных типа long с указателями.

Опция -o позволяет специфицировать имя выходного файла в формате:

```
-o "имя выходного файла"
```

Кстати, имеется возможность получить ассемблерный код на основе файла с исходным кодом на языке C. Для этого можно воспользоваться опцией -S и дать следующую ко-

манду (в каталоге, в котором выполняется команда, будет создан файл first.s):

```
gcc -S first.c
```

Также можно использовать опцию -O0 (если не указывать сразу после опции уровень, как мы в данном случае, – 0, то будет задан уровень по умолчанию – 1), которая указывает компилятору отказаться от применения оптимизации, т.е. полученный в ходе компиляции объектный файл будет полностью соответствовать структуре файла с исходным кодом. Приведем пример использования опции:

```
gcc -O0 -o first first.c
```

Теперь в каталоге, в котором происходила компиляция (в моей системе – это каталог Develop в домашнем каталоге) должен появиться исполняемый файл с именем first. Запустим его на выполнение. В выводе вы должны увидеть следующее:

```
a равно 0
a равно 1
a равно 2
a равно 3
b равно 0
b равно 1
b равно 2
b равно 3
Сумма a и b равна 8
```

Программа работает так, как мы и предполагали.

Теперь выясним, работает ли наша программа в 64-битном режиме. Сделать это можно несколькими способами. Например, с помощью утилиты readelf, которая входит в состав набора GNU Binary Utilities (GNU Binutils) [13]. Программа readelf предназначена для получения информации об объектном файле в формате ELF (ELF – формат исполняемых файлов в Linux и многих других разновидностях UNIX).

Итак, воспользуемся readelf и дадим следующую команду (см. рис. 2):

```
$ readelf -h first
```

В данном случае мы специфицируем (опция -h, она же --file-header) вывод информации только о заголовке нашего ELF-файла, для вывода всей информации, которую может предоставить утилита, воспользуйтесь опцией -a.

В данном выводе нам будут интересны два поля – Class и Machine. Как видно на рис. 2, значение первого поля – ELF64, а второго – Advanced Micro Devices X86-64. Таким образом, наш ELF-файл first относится к классу 64-битных ELF-файлов, скомпилированных для платформы AMD64.

Другим простейшим способом является использование утилиты file (предоставляет информацию о типе файла) в связке с утилитой nm (вывод символьной информации из объектного файла). Приведем пример использования (см. рис. 3):

```
$ file first && nm first
```

Еще один инструмент, позволяющий нам получить информацию об ELF-файле, – objdump, который входит в состав GNU Binutils (так же, как и readelf). Утилита ориентирована на вывод информации об объектных файлах. Кстати, читатель уже, наверное, заметил разницу между двумя утилитами – objdump и readelf. Если второй инструмент имеет «узкую специализацию» – работа с ELF-форматом, то objdump поддерживает работу с большим количеством форматов файлов на основе библиотеки BFD (Binary File Descriptor) [14].

Итак, воспользуемся утилитой objdump для получения информации о файле first:

```
$ objdump -f ./first
```

Рисунок 3. Получение информации об ELF-файле с помощью MC

```
Файл: first            Строка 1 Позиция 0 1277 байт
/home/bgt/Develop/first: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.15, not stripped
0000000000600e50 d _DYNAMIC
0000000000600fe8 d _GLOBAL_OFFSET_TABLE_
0000000000400618 R _IO_stdin_used
                 w _Jv_RegisterClasses
0000000000600e30 d __CTOR_END__
0000000000600e28 d __CTOR_LIST__
0000000000600e40 D __DTOR_END__
0000000000600e38 d __DTOR_LIST__
00000000004006d0 r __FRAME_END__
0000000000600e48 d __JCR_END__
0000000000600e48 d __JCR_LIST__
0000000000601020 A __bss_start
0000000000601010 D __data_start
00000000004005d0 t __do_global_ctors_aux
0000000000400460 t __do_global_dtors_aux
0000000000601018 D __dso_handle
                 w __gmon_start__
0000000000600e24 d __init_array_end
0000000000600e24 d __init_array_start
0000000000400530 T __libc_csu_fini
0000000000400540 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
0000000000601020 A _edata
0000000000601030 A _end
0000000000400608 T _fini
00000000004003c8 T _init
0000000000400410 T _start
000000000040043c t call_gmon_start
0000000000601020 b completed.7424
0000000000601010 W data_start
0000000000601028 b dtor_idx.7426
00000000004004d0 t frame_dummy
00000000004004f4 T main
                 U system@@GLIBC_2.2.5
```

Мы использовали опцию -f, которая позволяет осуществить вывод информации о заголовке объектного файла. В выводе программы мы видим, что мы работаем с 64-битным исполняемым файлом ELF для архитектуры i386 и x86-64 (об особенностях режима компиляции с опцией -m64 посредством GCC мы говорили выше) (см. рис. 4).

Помимо рассмотренных инструментов, получить информацию о режиме работы программы позволяет отладчик. В Linux основным отладчиком для программ, которые доступны вместе с исходными текстами, является GDB.

Выясним с помощью отладчика, в каком режиме работает программа first. Например, так: установим точку останова на функцию main, затем запустим программу на выполнение, а далее просмотрим состояние регистров процессора. Для этого запустим gdb в каталоге с этой программой:

```
$ gdb -q first
```

Будет запущен сеанс GDB (см. рис. 5) в так называемом Тихом (Quiet – см. пункт 2.1.2 официального руководства по GNU Debbuger или его русского перевода, упоминавшийся выше) режиме (не будут отображаться вводное сообщение и информация об авторских правах – см. рис. 6) и станет доступна его консоль.

Далее установим точку останова на функцию main:

```
(gdb) break main
```

Затем запустим программу:

```
(gdb) run
```

Осталось получить информацию о регистрах:

```
(gdb) info registers
```

Кстати, для большого количества команд в GDB доступны их сокращенные аналоги. Например, последнюю команду GDB можно было отдать следующим образом:

```
(gdb) i r
```

Итак, в выводе GDB (см. рис. 7) видно, что задействованы 64-битные регистры (RAX, RBX, RDX, RCX и т.д.). А если вспомним о регистрах процессоров платформы AMD64, то, как мы говорили ранее (расширенные регистры доступны исключительно в 64-битном подрежиме Длинного режима работы процессора), станет ясно: программа работает в 64-битном режиме.

Прежде чем мы продолжим, мне хотелось бы заострить внимание на одном вопросе. Компилятор GCC позволяет включать в исполняемый файл дополнительную отладочную информацию (например, исходный код). Чтобы задействовать эту возможность компилятора, надо использовать опцию -g. Применительно к нашему примеру это будет выглядеть следующим образом:

```
$ gcc -m64 -g -O0 -o first first.c
```

Теперь запустим отладчик:

```
$ gdb -q first
```

В консоли отладчика дадим команду:

```
(gdb) list
```

В выводе GDB мы увидим исходный код исследуемого файла (см. рис. 8).

Исследовать ELF-файл на предмет поставленной задачи можно не только в консольных инструментах, но и в отладчике с графическим интерфейсом – EDB.

Запустим программу:

```
$ edb
```

Далее откроем файл first: меню File → Open, а затем открыть файл стандартным образом, в появившемся окне диалога открытия файла. Откроется окно программы, которое условно можно разделить на пять частей:

- вверху – меню и панель инструментов;
- ниже слева – область вывода дизассемблированного кода;

Рисунок 4. Вывод заголовка объектного файла с помощью objdump

```
bgt@freedesk ~/Develop $ objdump -f ./first

./first:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0000000000400410
```

Рисунок 5. Старт GDB в тихом режиме

```
bgt@freedesk ~/Develop $ gdb -q first
Reading symbols from /home/bgt/Develop/first...(no debugging symbols found)...done
(gdb)
```

Рисунок 6. Старт GDB

```
bgt@freedesk ~/Develop $ gdb first
GNU gdb (GDB) 7.2-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/bgt/Develop/first...(no debugging symbols found)...done.
(gdb)
```

Рисунок 7. Вывод состояния регистров процессора с помощью GDB

```
bgt@freedesk ~/Develop $ gdb -q first
Reading symbols from /home/bgt/Develop/first...(no debugging symbols found)...done.
(gdb) break main
Breakpoint 1 at 0x4004f8
(gdb) run
Starting program: /home/bgt/Develop/first

Breakpoint 1, 0x00000000004004f8 in main ()
(gdb) info registers
rax            0x7ffff7dd8ec8   140737351880392
rbx            0x0      0
rcx            0x0      0
rdx            0x7fffffffe888   140737488349320
rsi            0x7fffffffe878   140737488349304
rdi            0x1      1
rbp            0x7fffffffe790   0x7fffffffe790
rsp            0x7fffffffe790   0x7fffffffe790
r8             0x7ffff7dd7300   140737351873280
r9             0x7ffff7deb4f0   140737351955696
r10            0x7fffffffe5e0   140737488348640
r11            0x7ffff7a77c90   140737348336784
r12            0x400410 4195344
r13            0x7fffffffe870   140737488349296
r14            0x0      0
r15            0x0      0
rip            0x4004f8 0x4004f8 <main+4>
eflags         0x246    [ PF ZF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb)
```

- справа – область, в которой отображается состояние регистров;
- слева внизу – область дампа;
- справа от нее – область стека.

У интерфейса есть ряд интересных возможностей. Так, изменять размер области можно путем перетаскивания за «ушки» – для примера одно из них выделено красным кружком. Синим кружком выделена кнопка, которая позволяет превратить область в окно (см. рис. 10 и 11).

Итак, обратившись к области просмотра регистров, мы видим, что задействованы 64-битные регистры, а обратившись к окну дампа и просмотрев текст, мы увидим, что работаем с 64-битным ELF-файлом.

Работа с регистрами

Выше мы уже сказали, как получить информацию о состоянии регистров. Но если нам необходимо получить информацию не о 64-битном регистре RAX, а, например, о 32-битном EAX? В GDB это просто.

Запустим GDB следующим образом:

```
$ gdb -q first
```

В консоли отладчика установим точку останова на функцию main:

```
(gdb) break main
```

А затем запустим программу:

```
(gdb) run
```

Теперь можно просмотреть состояние регистра eax:

```
(gdb) info register eax
```

или сокращенный вариант:

```
(gdb) i r eax
```

Таким образом можно получить доступ ко всем регистрам общего назначения. Для примера, мы показали на рис. 9 просмотр всех регистров RAX (от RAX до AL).

Но в целом это можно сделать и «на глаз». Так, например, в области просмотра состояние регистров отладчика EDB после запуска на выполнение программы first (Run → Debug или <F9>) получим значения регистра RAX. В моем случае – 00007ff1189e2ec8.

Итак, значением AL будут младшие восемь бит – c8, а значением AH последующие восемь бит – 2e. Поскольку AX включает в себя в качестве младшего регистра AL, а в качестве старшего – AH, то его значением будет 2ec8, что составляет 16 бит. Значение 32-битного регистра EAX – 189e2ec8, т.е. «плюс» еще 16 бит после AX. Условно говоря, RAX – это EAX + 32 бита (00007ff1189e2ec8).

Кстати, для быстрого перевода чисел между двоичной, десятичной и шестнадцатиричной системами можно воспользоваться pcalc (используем значение EAX, о котором мы говорили выше):

```
$ pcalc 0x2ec8
```

В выводе получим:

```
11976           0x2ec8          0y10111011001000
```

Таким образом, шестнадцатиричное число 0x2ec8 в десятичной системе равно 11976, а в двоичной – 0y10111011001000.

Вопросы дизассемблирования

Основной свободный дизассемблер – это objdump. Как мы говорили выше, он имеет графическую оболочку – Dissy. Приведем примеры его использования.

Дадим команду для вывода информации в объектном файле (только секций, которые содержат информацию):

```
$ objdump -d ./first
```

Вывод будет содержать информацию в следующем формате: адреса памяти, машинные команды процессора, команды ассемблера. Например (синтаксис AT&T):

```
4003c8:    48 83 ec 08    sub    $0x8,%rsp
```

Если добавить опцию -M:

```
$ objdump -M intel -d ./first
```

то вывод, приведенный выше, будет иметь вид:

```
4003c8:    48 83 ec 08    sub    rsp,0x8
```

Графическая оболочка Dissy делает работу с objdump наглядной. Так, откроем в ней объектный файл first (File → Open). В верхней части окна станут доступны секции программы (с адресом, размером и названием). При щелчке по любой из секций в нижней части окна становится доступной подробная информация о ней (в формате, указанном выше) (см. рис. 11).

В целом Dissy предоставляет удобную навигацию по секциям объектного файла.

Однако рассмотренные инструменты не всегда могут нам помочь. Так, например, программа tiny-crackme (ELF-файл) имеет поврежденный заголовок [15]. При попытке исследовать последний с помощью objdump:

Рисунок 8. Вывод исходного кода программы в GDB

```
bgt@freedesk ~/Develop $ gdb -q first
Reading symbols from /home/bgt/Develop/first...done.
(gdb) list
1       #include <stdio.h>
2
3       int main ()
4       {
5           system ("ls -l");
6           system ("touch newfile");
7           system ("ls -l");
8
9           return 0;
10      }(gdb)
```

Рисунок 9. Просмотр состояния регистров RAX

```
bgt@freedesk ~/Develop $ gdb -q first
Reading symbols from /home/bgt/Develop/first...done.
(gdb) break main
Breakpoint 1 at 0x4004f8: file first.c, line 5.
(gdb) run
Starting program: /home/bgt/Develop/first

Breakpoint 1, main () at first.c:5
5           system ("ls -l");
(gdb) i r rax
rax            0x7ffff7dd8ec8   140737351880392
(gdb) i r eax
eax            0xf7dd8ec8       -136474936
(gdb) i r ax
ax             0x8ec8   -28984
(gdb) i r ah
ah             0x8e     -114
(gdb) i r al
al             0xc8     -56
(gdb)
```

```
$ objdump -d tiny-crackme
```

Программа выдала ошибку (не распознан):

```
objdump: tiny-crackme: File format not recognized
```

Функции по дизассемблированию есть и в GDB, но в ситуации с tiny-crackme отладчик ведет себя не лучше objdump:

```
$ gdb -q tiny-crackme

"/home/bgt/Develop/tiny-crackme": not in executable format:
Формат файла не распознан
(gdb)
```

***

Архитектура AMD64 принесла большое количество новшеств. Среди них, например, расширенный набор регистров. Все это делает изучение возможностей архитектуры AMD64 интересным и познавательным. А свободное программное обеспечение – в качестве инструментов и официальная документация от AMD могут стать отправной точкой на пути совершенствования ИТ-специалистов. EOF

1. Официальная документация по разработке для AMD64 – http://developer.amd.com/documentation/guides/Pages/default.aspx.
2. AMD64 Architecture Programmer's Manual: Volume 1: Application Programming – http://support.amd.com/us/Processor_TechDocs/24592.pdf.
3. AMD64 Architecture Programmer's Manual: Volume 3: General-Purpose and System Instructions – http://support.amd.com/us/Processor_TechDocs/24594.pdf.
4. Подробное описание регистров процессоров – Зубков С.В. Assembler для DOS, Windows и UNIX. – М.: «ДМК Пресс», 2000. – С. 20-24; Юров В.И. Assembler/учебник для вузов – 2-е изд. – СПб.: «Питер», 2005. – С. 42-49; Эриксон Д. Хакинг: искусство эксплойта/пер. с англ. – 2-е изд. – СПб: «Символ-Плюс», 2010. – С. 37-38.
5. Официальный сайт проекта GCC – http://gcc.gnu.org; домашняя страница проекта GDB – http://www.gnu.org/software/gdb.
6. Анонс о включении фронтенда для языка Go в GCC – http://gcc.gnu.org/ml/gcc/2010-01/msg00500.html; официальный сайт нового языка Go – http://golang.org.
7. Официальная документация по GCC – http://gcc.gnu.org/onlinedocs; книга посвященная GCC – Гриффитс А. GCC. Настольная книга пользователей, программистов и системных администраторов. – К.: ООО «ТИД «ДС», 2004. – 624 с.
8. Официальное руководство по GDB – http://www.gnu.org/software/gdb/documentation; неофициальный перевод его на русский язык – http://mitya.pp.ru/gdb/gdb_toc.html.
9. Официальный сайт проекта GNU Binutils – http://www.gnu.org/software/binutils; официальный сайт проекта Dissy – http://code.google.com/p/dissy;
10. Домашняя страница проекта EDB и других проектов Эвана Терана – http://www.codef00.com/projects.php; Эван Теран о себе – http://www.codef00.com/about.php.
11. Официальный сайт pcalc – http://sourceforge.net/projects/pcalc.
12. Описание опций GCC для x86 и x86-64 – http://gcc.gnu.org/onlinedocs/gcc-4.5.1/gcc/i386-and-x86_002d64-Options.html#i386-and-x86_002d64-Options.
13. Официальная документация GNU Binutils – http://sourceware.org/binutils/docs-2.20 и, в частности, по readelf – http://sourceware.org/binutils/docs-2.20/binutils/readelf.html#readelf.
14. Официальное руководство по библиотеке BFD – http://sourceware.org/binutils/docs-2.20/bfd/index.html.
15. Загрузить архив с программой tiny-crackme можно по адресу – http://www.crackmes.de/users/yanisto/tiny_crackme.

Рисунок 9. Элементы интерфейса EDB

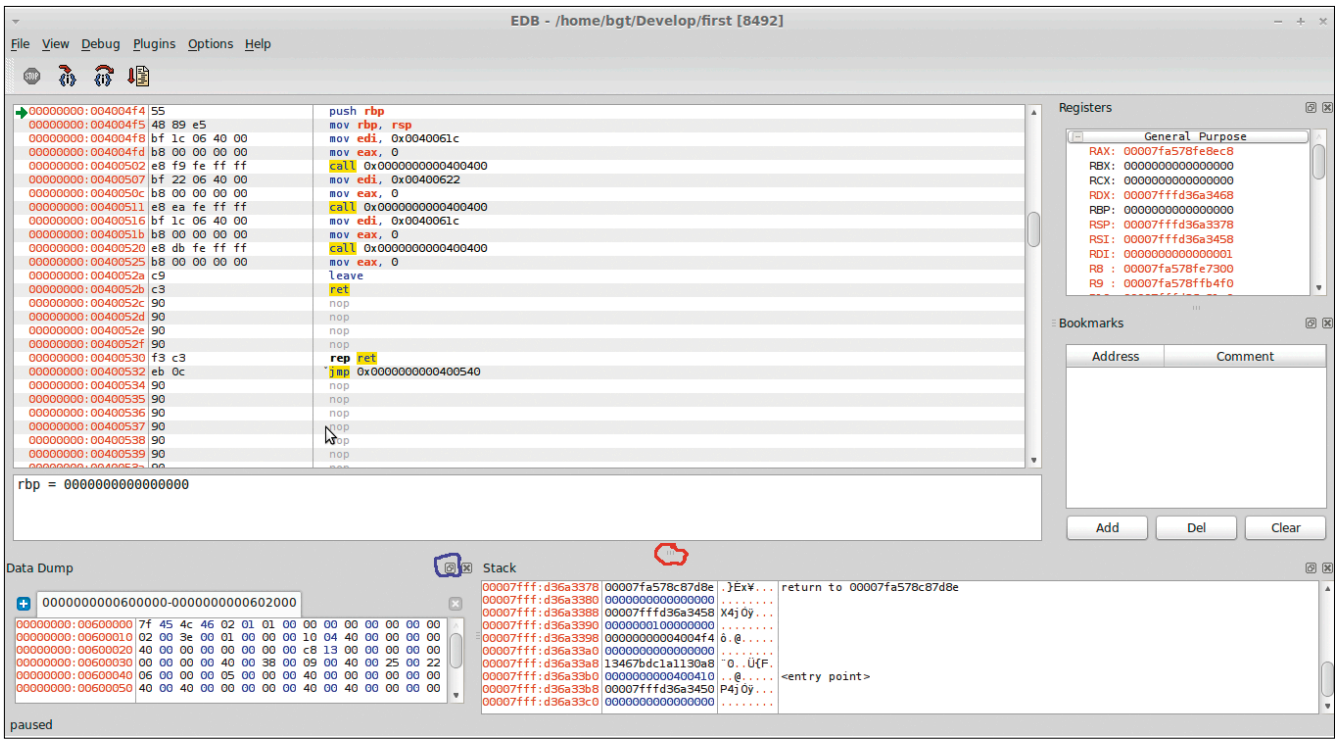


Рисунок 10. Область просмотра регистров «превратилась» в окно

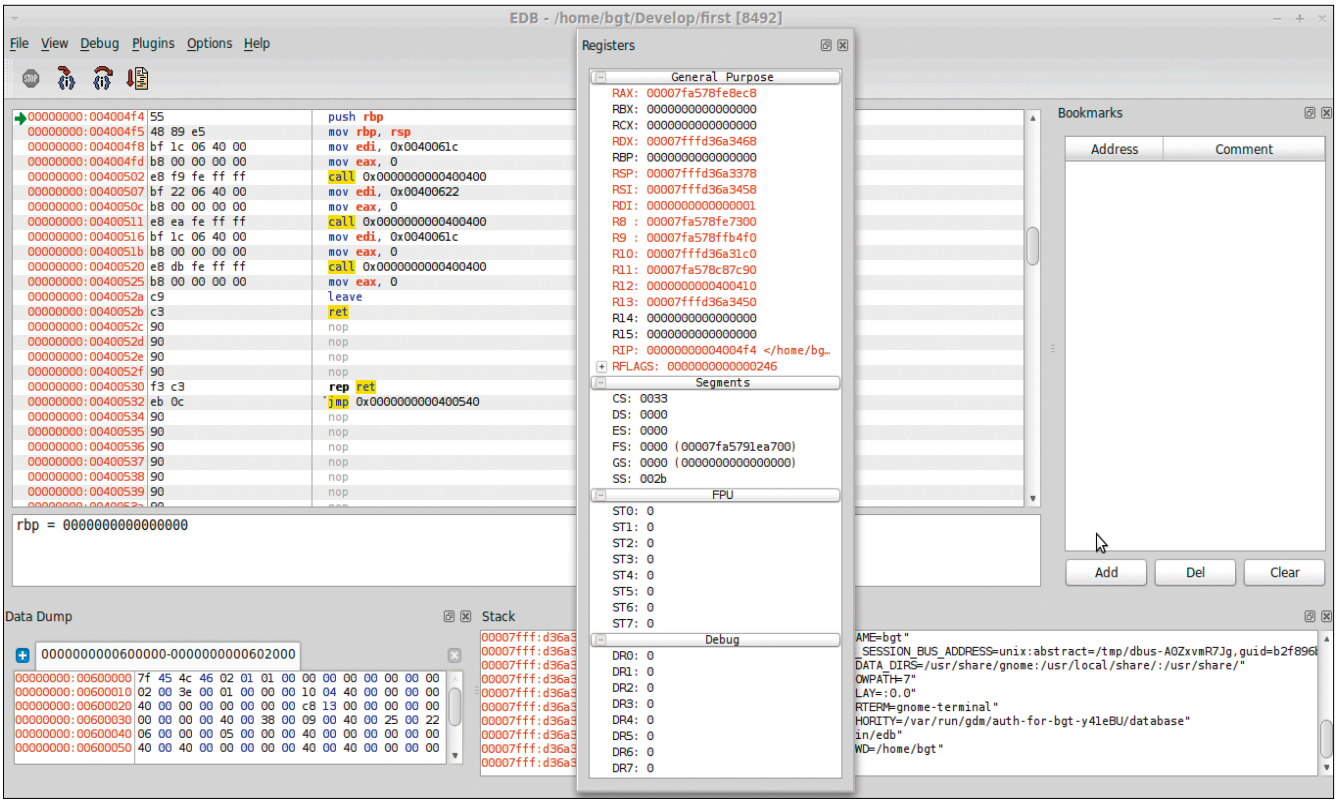


Рисунок 11. Dissy за работой