**BC/NW 2021№ 2 (37):9.1**

**ЗАДАЧИ ДИЗАСЕМБЛИРОВАНИЯ ПРИКЛАДНЫХ ПРОГРАММ**

Абросимов Л.И., Качалин М. Н., Орлов Д.А.

1. форматы исполняемых файлов: - PE (используется в windows) - ELF(используется в unix)

PE

Сигнатура

Первые 2 байта PE-файла содержат сигнатуру 0x4D 0x5A — «MZ» (как наследник MZ-формата). Далее — двойное слово по смещению 0x3C содержит адрес PE-заголовка. Последний начинается с сигнатуры 0x50 0x45 — «PE».

Структура

Файл PE состоит из нескольких заголовков и секций, которые указывают динамическому компоновщику, как отображать файл в память. Исполняемый образ состоит из нескольких различных областей (секций), каждая из которых требует различных прав доступа к памяти; таким образом, начало каждой секции должно быть выровнено по границе страницы. Например, обычно секция .text, которая содержит код программы, отображена как исполняемая/доступная только для чтения, а секция .data, содержащая глобальные переменные, отображена как неисполняемая/доступная для чтения и записи. Однако, чтобы не тратить впустую пространство на жёстком диске, различные секции на нём на границу страницы не выровнены. Часть работы динамического компоновщика состоит в том, чтобы отобразить каждую секцию в память отдельно и присвоить корректные права доступа получившимся областям согласно указаниям, содержащимся в заголовках.

Таблица импорта

Одна из известных секций — таблица адресов импорта (IAT — Import Address Table), которая используется в качестве таблицы поиска, когда приложение вызывает функцию из другого модуля. Это может быть сделано и в форме импорта по порядковому номеру функции (ordinal), и импорта по её имени. Поскольку скомпилированной программе неизвестно расположение библиотек, от которых она зависит, то требуется производить косвенный переход всякий раз, когда происходит вызов API-функции. Когда динамический компоновщик загружает модули и объединяет их, он записывает действительные адреса в область IAT так, чтобы они указали на ячейки памяти соответствующих библиотечных функций. Хотя это добавляет дополнительный переход внутри модуля, приводящий к потере производительности, это предоставляет ключевое преимущество: количество страниц памяти, которые должны быть скопированы загрузчиком при записи, минимизировано, что приводит к экономии памяти и дискового времени ввода-вывода. Если компилятору будет известно заранее, что вызов будет межмодульным (через атрибут dllimport), то он сможет произвести более оптимизированный код, который просто приводит к коду операции косвенного вызова.

Таблица экспорта

Таблица адресов экспорта (EAT — Export Address Table) нужна для того, чтобы один модуль (обычно это динамически загружаемая библиотека) мог указать другим модулям, какие функции они могут из него импортировать, и по каким адресам последние расположены.

Таблица перемещений

Файлы PE не содержат позиционно-независимого кода. Вместо этого они скомпилированы для предпочтительного базового адреса, и все адреса, генерируемые компилятором/компоновщиком, заранее фиксированы. Если PE-файл не может быть загружен по своему предпочтительному адресу (потому что он уже занят чем-то ещё), операционная система будет перебазировать его. Это включает в себя перевычисление каждого абсолютного адреса и изменение кода для того, чтобы использовать новые значения. Загрузчик делает это, сравнивая предпочтительный и фактический адреса загрузки, и вычисляя значение разности. Тогда для получения нового адреса ячейки памяти эта разность складывается с предпочтительным адресом. Базовые адреса перемещений хранятся в списке и при необходимости добавляются к существующей ячейке памяти. Полученный код является теперь отдельным по отношению к процессу и не является больше разделяемым, так что при таком способе теряются многие из преимуществ экономии памяти динамически загружаемых библиотек. Такой способ также значительно замедляет загрузку модуля. По этой причине следует избегать перебазирования везде, где это возможно; например, библиотеки, поставляемые Microsoft, имеют предварительно вычисленные неперекрывающиеся базовые адреса. В случае отсутствия необходимости перебазировании PE-файлы имеют преимущество очень эффективного кода, но при наличии перебазирования издержки в использовании памяти могут быть значительными. Это отличает формат PE от ELF, который использует полностью позиционно-независимый код и глобальную таблицу смещений, которая жертвует временем выполнения в пользу расходования памяти.

ELF

Заголовок файла

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. Заголовок имеет размер 52 байта для 32-битных файлов или 64 для 64-битных. Данное различие обуславливается тем, что в заголовке файла содержится три поля, имеющих размер указателя, который составляет 4 и 8 байт для 32- и 64-битных процессоров соответственно. Такими полями являются e\_entry, e\_phoff и e\_shoff.

Таблица заголовков программы

Таблица заголовков программы содержит заголовки, каждый из которых описывает отдельный сегмент программы и его атрибуты либо другую информацию, необходимую операционной системе для подготовки программы к исполнению. Данная таблица может располагаться в любом месте файла, её местоположение (смещение относительно начала файла) описывается в поле e\_phoff заголовка ELF.

При анализе структуры заголовка программы можно обнаружить различное местоположение поля p\_flags для 32- и 64-битных ELF файлов. Данное различие обуславливается выравниванием структуры для увеличения эффективности обработки.

Таблица заголовков секций

Таблица заголовков секций содержит атрибуты секций файла. Данная таблица необходима только компоновщику, исполняемые файлы в наличии этой таблицы не нуждаются (ELF загрузчик её игнорирует). Предоставленную в таблице заголовков секций информацию компоновщик использует для оптимального размещения данных секций по сегментам при сборке файла с учётом их атрибутов.

2. Изучить механизмы загрузки динамических библиотек

Динамические библиотеки

Общие концепции

Преимущества

• Обновления и исправления ошибок. Если нужно обновить код, который вынесен в динамическую библиотеку, то достаточно обновить только библиотеку, и все программы, что её используют, получат новую версию, их не надо пересобирать.

• Экономия памяти. Если одну и ту же библиотеку использует несколько приложений, в оперативной памяти может храниться только один ее экземпляр, доступный этим приложениям. Пример — библиотека C/C++. Ею пользуются 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 динамической версией библиотеки C/C++, в памяти будет присутствовать лишь одна копия кода этих функций, что позволит гораздо эффективнее использовать оперативную память.

• Экономия места на диске. Аналогично.

• Общие данные. Библиотеки могут содержать такие ресурсы, как строки, значки и растровые изображения. Эти ресурсы доступны любым программам.

• Расширение функциональности приложения. Библиотеки 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 библиотек от других компаний.

• Возможность использования разных языков программирования. У вас есть выбор, на каком языке писать ту или иную часть приложения. Так, пользовательский интерфейс — на одном, прикладную логику — на другом.

• Реализация специфических возможностей. Определенная функциональность доступна только при использовании динамических библиотек. Например, в Windows 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 COM-объектов, существование которых допустимо только в DLL. Если говорить о UNIX, то примером специфической для динамических библиотек является возможность LD\_PRELOAD.

• Ускорение процесса сборки. Каждый раз при сборке анализируются изменения только в коде приложения, выполняется более лёгкая компоновка.

• Разработка в большой команде. Если в процессе разработки программного продукта отдельные его модули создаются разными группами, то при использовании динамических библиотек таким проектом управлять проще.

Недостатки

• Разработка в большой команде. Затрудняется внесение изменений в код, нужно постоянно думать об обратной совместимости.

• Неудобства при сборке. Например, нужно собрать код программы с каким-то флагом компилятора, при этом приходится следить, чтобы все нужные библиотеки были также пересобраны с флагом и подхватились при запуске.

• Неудобство развёртывания. Статически скомпонованная программа — один самодостаточный файл.

• Dependency hell. Многочисленные проблемы разного вида. Например, две версии одной библиотеки, половина программ требуют первую, другая половина программ — вторую.

• Потери производительности. Вызов функции из динамической библиотеки получается медленнее.

Файлы динамических (разделяемых) библиотек

Внутренний формат представления динамических библиотек в современных системах аналогичен тому, что используется для исполняемых файлов, что позволяет упростить систему. Библиотеки не предназначены для запуска напрямую. Загружаются в память процесса загрузчиком программ операционной системы либо при создании процесса, либо по запросу уже работающего процесса, то есть динамически.

В UNIX-системах библиотеки имеют расширение so (shared object), в Windows — расширение dll (dynamic link library).

Динамическая загрузка

Динамические библиотеки могут использоваться двумя способами:

• динамическая компоновка (dynamic linking)

• динамическая загрузка (dynamic loading)

При динамической загрузке программа сама загружает конкретную библиотеку, указывая путь к ней, затем находит в библиотеке нужную функцию по имени и вызывает её. Это частый паттерн использования в программах, которые поддерживают плагины.

Переадресация (relocation)

Разные программы имеют различные размеры и различный набор подгружаемых динамических библиотек, и если разделяемая библиотека отображается в адресное пространство различных программ, она будет иметь различные адреса. Это в свою очередь означает, что все функции и переменные в библиотеке будут на различных местах. Если все обращения к адресам относительные («значение +1020 байта отсюда») нежели абсолютные («значение в 0x102218BF»), то это не проблема, однако так бывает не всегда. В таких случаях всем абсолютным адресам необходимо прибавить подходящий офсет — это и есть relocation.

Это практически всегда скрыто от C/C++ программиста — очень редко проблемы компоновки вызваны трудностями переадресации.

Таблица перемещений (relocation table) — это список указателей, созданный транслятором (компилятором или ассемблером) и хранимый в объектном или исполняемом файле. Каждая запись в таблице, или «fixup», является указателем на абсолютный адрес в объектном коде, который должен быть изменен, когда загрузчик перемещает программу так, чтобы она ссылалась на правильное местоположение. Fixup'ы предназначены для поддержки переноса программы в виде цельной единицы.

ASLR

ASLR (англ. address space layout randomization — «рандомизация размещения адресного пространства») — технология, применяемая в операционных системах, при использовании которой случайным образом изменяется расположение в адресном пространстве процесса важных структур данных, а именно образов исполняемого файла, подгружаемых библиотек, кучи и стека.

Технология ASLR создана для усложнения эксплуатации нескольких типов уязвимостей. Например, если при помощи переполнения буфера или другим методом атакующий получит возможность передать управление по произвольному адресу, ему нужно будет угадать, по какому именно адресу расположен стек, куча или другие структуры данных, в которые можно поместить шелл-код. Если не удастся угадать правильный адрес, приложение скорее всего аварийно завершится, тем самым лишив атакующего возможности повторной атаки и привлекая внимание системного администратора.

API и ABI

API: Application Program Interface

Это набор публичных типов, переменных, функций, которые вы делаете видимыми из вашего приложения или библиотеки.

В C и C++ API обычно поставляется в виде заголовочного файла (h) вместе с библиотекой.

С API работают люди, когда пишут код.

ABI: Application Binary Interface

Детали реализации этого интерфейса. Определяет такие вещи, как

• Способ передачи параметров в функции (регистры, стек).

• Кто извлекает параметры из стека (вызывающий код или вызываемый, caller/callee).

• Как происходит возврат значений из функции.

• Как реализован механизм исключений.

• Декорирование имён в C++ (mangling).

ABI важно, когда приложение использует внешние библиотеки. Если при обновлении библиотеки ABI не меняется, то менять программу не надо. API может остаться тем же, но поменяется ABI. Две версии библиотеки, имеющие один ABI, называют binary compatible (бинарно совместимыми): старую версию библиотеки можно заменить на новую без проблем.

Иногда без изменений ABI не обойтись. Тогда приходится перекомпилировать зависящие программы. Если ABI библиотеки меняется, а API нет, то версии называют source compatible.

Разработчики библиотеки стараются поддерживать ABI стабильным. Новые функции и типы данных могут добавляться, но старые должны сохраняться.

Несмотря на то что общие принципы разделяемых библиотек примерно одинаковы как на платформах UNIX, так и на Windows, всё же есть несколько существенных различий.

Экспортируемые символы

Самое большое отличие заключается в том, что в библиотеках Windows символы не экспортируются автоматически. В UNIX все символы всех объектных файлов, которые были подлинкованы к разделяемой библиотеке, видны пользователю этой библиотеки. В Windows программист должен явно делать некоторые символы видимыми, т. е. экспортировать их.

Динамические библиотеки и язык C++

Поддержка бинарной совместимости

Что делать можно

• Можно добавить новую функцию, новый невиртуальный метод класса, новый конструктор и новые статические поля.

• Можно добавлять новые перегрузки к существующим перегруженным функциям, создавать новые перегруженные функции или перегружать private-методы.

• Можно добавлять новые виртуальные функции в конец класса, не имеющего наследников.

• Можно изменить тело любой непоинлайненной функции (т. е. не определённой в публичном заголовочном файле).

• Можно добавлять новые классы.

• Можно произвольным образом изменять, добавлять и удалять friend declarations.

• Можно переименовывать private-поля классов.

• Можно делать всё, что угодно, с классами и функциями, не являющимися частью API (размещёнными в cpp-файле и имеющими static или обёрнутыми в анонимный namespace).

Что делать не стоит

• Можно изменить тело поинлайненной функции или унести её в .cpp. Однако функциональность менять при этом нельзя (даже исправлять ошибки), поскольку собранные с библиотекой бинарники продолжат пользоваться старой версией функции.

• Можно переопределить виртуальный метод, определённый в базовом классе. Однако функциональность менять при этом, опять же, нельзя, т. к. в ряде мест компилятор может заменять виртуальный вызов прямым (напр. Derived d; d.foo();).

• Можно удалить невиртуальный закрытый (private) метод класса (или private static поле). Но перед этим нужно убедиться, что метод никогда за время жизни библиотеки не вызывался (или поле не использовалось) ни из одной inline-функции.

• Можно изменять значения по умолчанию для функций и методов; однако уже собранные бинарники продолжат передавать старые значения по умолчанию.

• Можно добавлять перегруженные варианты существующих неперегруженных public-методов. Это сохраняет бинарную совместимость, но код, который брал адрес такой функции (auto x = &MyClass::method) перестанет компилироваться.

Чего делать нельзя

• Нельзя удалять существующие классы, являющиеся частью API.

• У существующих классов нельзя удалять открытые, защищённые или виртуальные методы.

• Нельзя как-либо менять существующую иерархию классов (т. е. изменять, добавлять, удалять или переупорядочивать базовые классы).

• Нельзя как-либо менять список шаблонных параметров для существующих шаблонных классов и функций.

• Нельзя менять сигнатуру существующих функций и методов.

• Нельзя добавлять виртуальные методы в середину класса.

• Нельзя добавлять виртуальные методы в класс, не имеющий виртуальных функций или виртуального деструктора.

• Нельзя добавлять виртуальные методы в класс, имеющий наследников.

• Нельзя добавлять, переупорядочивать или удалять поля существующих классов.