Санкт-Петербургский государственный университет
математико-механический факультет
кафедра системного программирования

Курсовая работа на тему:

Разработка метода сбора информации о ходе исполнения программы, который использует возможность модификации памяти процесса

Студент 445 группы
Булычев Иван Дмитриевич
Научный руководитель:
Преподаватель кафедры Системного Программирования
Баклановский Максим Викторович

Оценка:

Санкт-Петербург
2010

Содержание

Введение

В настоящее время значительная часть промышленного программирования осуществляется на managed языках, таких как JAVA, C#, managed C++ и т.д. В силу своих особенностей они не позволяют оптимальным образом управлять ресурсами компьютера. Из-за работы сборщика мусора, JIT компиляции, контроля типов и других различных проверок производительность приложения падает в разы. С другой стороны, эти языки позволяют писать огромные объемы безопасного кода, который легко отлаживается и дополняется, который легко поддается рефакторингу. Можно привести множество других преимуществ, из-за которых при разработке приложения выбираются именно эти языки.

Проблема с производительностью решается следующим образом: функции, для которых время исполнения критично, реализуют с помощью оптимальных алгоритмов на более быстрых языках (ANSI C++, ассемблер). Как правило, таких критических участков в программе значительно меньше, чем остального кода. В качестве примера можно привести локальные и удаленные базы данных, Platform Invocation Services (в Dot.NET) и Java Native Interface (в JAVA).

Поэтому задача разработки быстрых реализаций различных алгоритмов, требования к которым по времени и по памяти критичны, стоит особенно остро. Существует множество программ и утилит, которые занимаются тестированием производительности и оптимизацией приложения. В некоторых компаниях существуют должности, которые отвечают за производительность отдельных логических частей программы и системы в целом.

В этой работе будет рассмотрен метод, который позволяет оценивать эффективность алгоритма в плане использования им ресурса процессора.

Работа имеет две содержательные части: изложение метода и пример реализации. Суть метода, основные идеи и приемы изложены в пятой главе “Идея метода”. Один из вариантов использования метода приведен в шестой главе “Реализация метода”. Также в этой главе и приложениях приведены исходные тексты некоторых ключевых частей реализации.

Все материалы работы находятся по адресу:

http://willzyx-edu-project.googlecode.com/svn/trunk/Proj1/

Глава 1. Введение в предметную область

1.1. Архитектура фон Неймана

30 июня 1945 года Джон фон Нейман написал статью “First Draft of a Report on the EDVAC” [8]. Этот отчет описывает компьютер, состоящий из четырех основных частей: центрального арифметического устройства, центрального управляющего устройства, памяти и средств ввода-вывода. Это первая публикация, которая содержит описание логического дизайна компьютера, реализующего *концепцию однородности памяти*, при которой программы и данные хранятся в одной и той же памяти. Такая архитектура в дальнейшем стала называться архитектурой фон Неймана.

Помимо однородности памяти, идея архитектуры включает в себя:

- *принцип двоичного кодирования*, когда вся информация, поступающая в ЭВМ, кодируется с помощью двоичных сигналов,
- *принцип адресности* – основная память состоит из пронумерованных ячеек и процессору в произвольный момент времени доступна любая ячейка, и
- *принцип программного управления*, который определяет программу как набор команд, которые выполняются процессором автоматически друг за другом в определенной последовательности.

Существует и альтернативный вариант архитектуры – гарвардская архитектура. Она была разработана Говардом Эйкеном в конце 1930-х годов в Гарвардском университете. Главное отличие от фон-неймановской заключается в раздельном хранении и раздельной обработке команд и данных. Этот подход имеет ряд преимуществ и недостатков. Но из-за сложности такой архитектуры, ее серьезное развитие началось лишь с конца 70-х годов.

1.2. История развития микропроцессоров

Сегодня, более полувека спустя, почти все микропроцессоры имеют фон-неймановскую архитектуру.

1971 год. По заказу небольшой японской компании Nippon Calculating Machine, Ltd, занимающейся производством калькуляторов, компанией Intel был выпущен первый в мире микропроцессор. Intel 4004 стал первым 4-битным коммерчески доступным однокристальным микропроцессором.

1972 год. Компания Intel разрабатывает первый 8-битный центральный процессор - Intel 8008. Процессор позиционировался как процессор для продвинутых калькуляторов общего назначения, терминалов ввода-вывода и автоматов бутылочного разлива.

1974 год. Компания Intel выпускает 8-битный микропроцессор Intel 8080. Он представлял собой усовершенствованную версию процессора Intel 8008. По заверениям Intel, этот процессор обеспечивал десятикратный прирост производительности по сравнению с микропроцессором Intel 8008. Благодаря 16-разрядной адресной шине процессор позволял производить адресацию 64 Кбайт памяти, которая не разделялась на память команд и данных. На базе нового микропроцессора Intel 8080 фирмой MITS был выпущен первый в мире миникомпьютерный комплект (персональный компьютер) Altair-8800.

1978 год. Intel 8086 — первый 16-битный микропроцессор выпущенный компанией Intel. Процессор имел набор команд, который применяется и в современных процессорах, именно от этого процессора берёт своё начало известная на сегодня архитектура x86. Размер шины адреса был увеличен с 16 бит до 20 бит, что позволило адресовать 1 Мбайт памяти

Для того чтобы адресовать больший, чем i8080, объём памяти, потребовалось изменить способ адресации памяти. Поэтому для адресации 1 Мбайт памяти применили следующую схему. На шину адреса подавался физический адрес размером 20 бит, который формировался путём сложения содержимого одного из сегментных регистров (16 бит), умноженного на 16, с содержимым указательного регистра: таким образом, адресация ячейки памяти производилась по номеру сегмента и эффективному адресу ячейки в сегменте (называемому также смещением). Этот метод впоследствии назвали реальным режимом адресации процессора, такой режим позволяет адресовать до 1 Мбайт памяти.

1982 год. Intel 80286 (также известный как i286) — 16-битный x86-совместимый микропроцессор второго поколения фирмы Intel.

В процессоре i286 было реализовано два режима работы — защищённый режим и реальный режим. В реальном режиме работы процессор был полностью совместим с процессорами x86, выпускавшимися до этого, то есть процессор мог выполнять программы, предназначенные для Intel 8086/8088/8018x с минимальными модификациями. В формировании адреса участвовали только 20 линий шины адреса, поэтому максимальный объём адресуемой памяти, в этом режиме, остался прежним — 1 Мбайт.

В защищённом режиме процессор мог адресовать до 16 Мбайт виртуальной памяти, за счёт изменения механизма адресации памяти. Программист и разрабатываемые им программы используют логическое адресное пространство (виртуальное адресное пространство), размер которого может составлять 16 Мбайт. Логический адрес преобразуется в физический адрес аппаратно с помощью блока управления памятью (MMU). Благодаря защищённому режиму, в памяти можно хранить только ту часть данных, которая необходима в данный момент, а остальная часть могла храниться во внешней памяти (например, на жёстком диске). Когда программа обращается к тем данным, которых нет в

физической памяти, операционная система может приостановить ее исполнение, загрузить требуемую секцию из внешней памяти и возобновить выполнение программы.

Для защиты от выполнения привилегированных команд, которые могут обрушить всю систему, для защиты доступа к данным и для защиты сегментов кода в процессоре i286 была введена защита по привилегиям [6]. Было выделено 4 уровня привилегий (кольца защиты, режимы работы): от самого привилегированного 0 уровня (Ring 0, режим ядра), предназначенного для ядра системы, до наименее привилегированного 3 уровня (Ring 3, пользовательский режим), предназначенного для прикладных программ.

1985 год. Intel 80386 - 32-битный x86-совместимый процессор третьего поколения фирмы Intel.

Процессор i386 полностью совместим со своими предшественниками — процессорами 8086-80286. Он выполняет программы, предназначенные для них без необходимости модификации кода и перекомпиляции (или с минимальными модификациями).

Адресация в защищенном режиме стала 32-битной (с возможностью создания 16-битных сегментов, для совместимости с 80286). Она позволила впервые со времени появления 8086 забыть о сегментации, а точнее, ограничении размера сегмента 64 килобайтами (ограничение 16-битного адреса). До появления i386 программы и операционные системы использовали несколько моделей организации памяти, различающихся по организации в памяти сегментов кода, стека и данных. 32-битный адрес позволил вместо них использовать одну простую плоскую модель, в которой все сегменты задачи находятся в одном и том же месте адресного пространства памяти. Плоская модель расширяет размер такого “общего” сегмента до 4 Гбайт. И для того, чтобы иметь возможность адресовать такой огромный по тем временам объем данных, был разработан механизм виртуальной памяти.

Дальнейшее развитие процессоров. В последующих моделях процессоров компания Intel занималась оптимизацией работы с памятью, ускорением и распараллеливанием выполнения инструкций, совершенствованием архитектуры:

- ввод кэша первого и второго уровней;
- раздельное кэширование программного кода и данных;
- многоядерные процессоры.

1.3. Организация памяти

В настоящее время существует множество физических устройств хранения данных, различающихся по скорости доступа, отказоустойчивости, объему хранимой информации.

Развитие компьютерных технологий в этом направлении сопровождалось появлением задач распределения и оптимизации памяти.

Работа с оперативной памятью в фон-неймановских архитектурах имела свой собственный особый путь развития:

1. Сначала, когда память измерялась десятками и сотнями килобайт, очень остро стояла проблема размещения данных и исполняемого кода в таком малом объеме памяти.
2. С появлением защищенного режима проблема малого объема оперативной памяти решилась сама собой (4-х Гб адресного пространства хватало для всех мыслимых задач). Был реализован механизм виртуализации памяти, а с ним появилось множество задач по его оптимизации.

1.3.1. Оверлейная загрузка программ

Процессоры фирмы Intel, начиная с версии 8086, вплоть модели Intel 80286 использовали реальный режим адресации. Объем памяти в этом режиме был ограничен одним мегабайтом. Также невозможно было создание полноценных многозадачных операционных систем. Но все же такие компьютеры в свое время широко использовались. Конечно, тогда программы были по размеру значительно меньше, чем современные, но очень часто возникала такая ситуация, что мегабайта для них было либо недостаточно, либо они занимали такое значительное место в памяти, что такая растрата была просто неприемлемой. Поэтому появлялись различные методы частичной загрузки приложения в оперативную память. Отметим один из них – оверлейную загрузку программ. Ранее она была очень распространена. Например, ее поддержка была введена в Turbo Pascal 3.0.

Смысл оверлея состоит в том, чтобы не загружать программу в память целиком, а разбить ее на несколько модулей и помещать их в память по мере необходимости. При этом на одни и те же адреса в различные моменты времени будут отображены разные модули. Оверлеи могут значительно сократить объем памяти, необходимый для выполнения программы. Фактически, так как в любой момент времени в памяти размещаются только части программы, с помощью оверлеев можно выполнять программы, многократно превосходящие по объему доступную память.

Один из примеров использования механизма оверлеев:

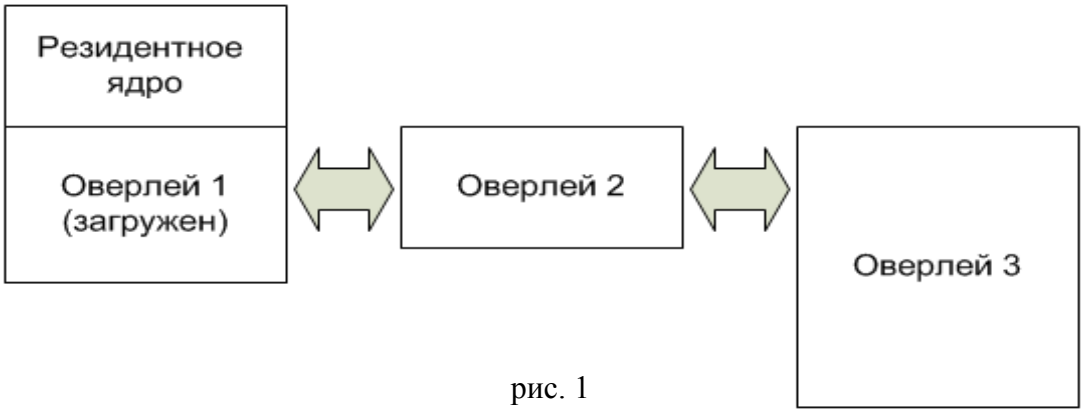

рис. 1

Здесь (см. рис. 1) в оперативной памяти всегда находится только один оверлей, и, при вызове процедуры из другого оверлея, происходит замещение в памяти одного другим.

Уделялось много внимания тому, как распределять процедуры между оверлейными модулями: т.к. доступ к внешней памяти выполняется всегда в разы дольше, чем к оперативной, то нужно стремиться к минимизации загрузок модулей с диска. Для этого необходимо, чтобы каждый оверлейный модуль был как можно более самодостаточным. Если это невозможно, стараются вынести процедуры, на которые ссылаются из нескольких оверлеев, в отдельный модуль, называемый резидентной частью или резидентным ядром. Это модуль, который всегда находится в памяти и не разделяет свои адреса ни с каким другим оверлеем. Естественно, оверлейный менеджер должен быть частью этого ядра.

Потребность в таком способе загрузки появляется, если у нас виртуальное адресное пространство мало, например 1 Мбайт или даже всего 64 Кбайт, а программа относительно велика. На современных 32-разрядных системах виртуальное адресное пространство измеряется гигабайтами, и большинству программ этого хватает, а проблемы с нехваткой можно решать совсем другими способами. Тем не менее, существуют различные системы, даже и 32-разрядные, в которых нет устройства управления памятью, и размер виртуальной не может превышать объема оперативной памяти, установленных на плате.

Несмотря на определенное сходство между задачами, решаемыми механизмом оверлеев и виртуальной адресацией, одно ни в коем случае не является разновидностью другого. При виртуальной адресации мы решаем задачу отображения большого адресного пространства на ограниченную оперативную память. При использовании оверлея мы решаем задачу отображения большого количества объектов в ограниченное адресное пространство.

1.3.2. Виртуальная память

Появление защищенного режима в 32-разрядных процессорах позволило адресовать все 4 Гб, а также аппаратно были реализованы функции, которые обеспечивают

необходимую защиту памяти, позволяют реализовать в операционных системах виртуальное адресное пространство и многозадачность.

Для адресации операндов в физическом адресном пространстве прикладные программы используют логическую адресацию, т.е. адреса, используемые программистом, необязательно совпадают с физическими. Блок управления памятью транслирует логические адреса программ в физические и посылает их на системную шину. Все это происходит на аппаратном уровне. Логическое адресное пространство организуется как набор из элементарных структур - байтов, сегментов, страниц. Наиболее популярны следующие модели:

- *Плоское (линейное)* логическое адресное пространство. Это массив байтов, не имеющий определенной структуры. Трансляция адреса не требуется, поскольку логический адрес совпадает с физическим.
- *Сегментированное* логическое адресное пространство. Состоит из нескольких сегментов, каждый из которых может быть произвольной длины. Логический адрес содержит идентификатор сегмента и смещение внутри сегмента.
- *Страничное* логическое адресное пространство. Состоит из большого числа страниц, каждая из которых включает фиксированное число байтов. Логический адрес состоит из идентификатора (номера) страницы и смещения внутри страницы.
- *Сегментно-страничное* логическое адресное пространство. Состоит из сегментов, которые в свою очередь состоят из страниц. Логический адрес транслируется в номер страницы и смещение в ней, которые затем транслируются в физический адрес.

1.3.3. Страничная организация памяти

В настоящее время очень широко распространен вариант страничной организации памяти. Поддержка такого режима присутствует в большинстве 32-битных и 64-битных процессоров. Такой режим является классическим для почти всех современных ОС, в том числе Windows и семейства UNIX. Он впервые был реализован фирмой DEC в процессорах VAX и ОС VMS в конце 70-х годов. В семействе x86 поддержка появилась с поколения 386, оно же первое 32-битное поколение.

Суть его заключается в том, что оперативная память делится на страницы: области памяти фиксированной длины (например, 4096 байт), которые являются минимальной единицей выделяемой памяти. Процесс обращается к памяти с помощью адреса виртуальной памяти, который содержит в себе номер страницы и смещение внутри страницы.

Операционная система преобразует виртуальный адрес в физический, при необходимости подгружая страницу из внешней памяти в оперативную. При запросе на выделение памяти или при загрузке сохраненной страницы операционная система может “сбросить” на жёсткий диск страницы, к которым давно не было обращений.

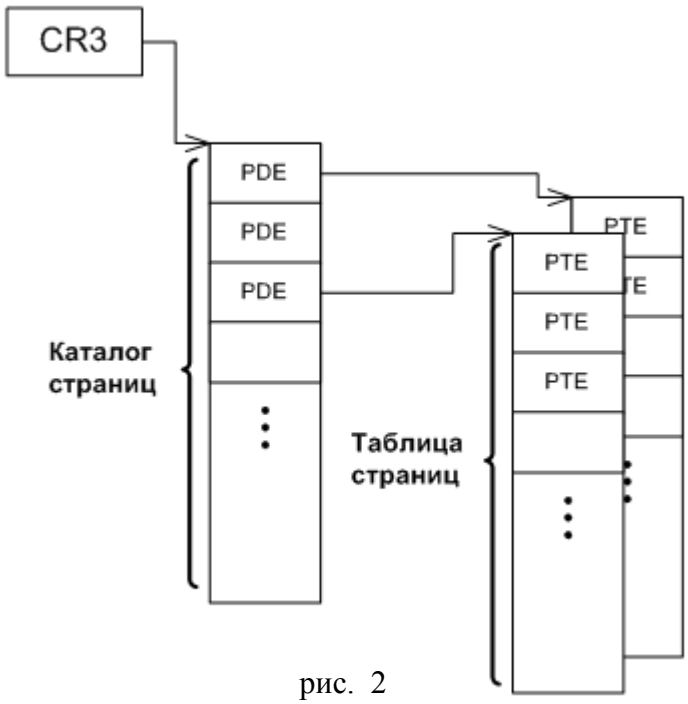

рис. 2

Сейчас рассмотрим процесс трансляции виртуального адреса в физический (см. рис. 2). Для 32-битных систем, адрес делится на три части: старшие 10 бит, средние 10 бит и младшие 12 бит. Первое число определяет номер записи в каталоге страниц. Каталог страниц состоит из 1024 записей по 4 байта – PDE (page directory entries), каждая из которых адресует соответствующую таблицу страниц. Следующее 10-битное число в виртуальном адресе определяет номер записи в таблице страниц. Каждая такая запись также состоит их 4-х байт и является дескриптором виртуальной страницы (PTE – page table entries). В этом дескрипторе хранится номер соответствующей страницы в физической памяти и поле флагов, описывающих состояние и атрибуты защиты страницы. Последние 12 бит определяют смещение внутри страницы.

На долю процессора в этом механизме виртуализации приходится порождение исключений #PF (PAGE FAULT) и #GP (GENERAL PROTECTION FAULT), доступ к памяти в соответствии с настройками дескрипторов страниц, хранящимися в иерархичной структуре в физической памяти, и использование буфера ассоциативной трансляции – TLB [5].

В семействе операционных систем Windows виртуальная память зародилась, начиная с версии 3.0 [11]. В реализациях, которые существуют сейчас, операционная система отвечает за загрузку и выгрузку страниц во внешнюю физическую память: обрабатывая исключения #PF, в оперативную память загружается отсутствующая страница, а если места для нее там нет, другая страница, использование которой не ожидается в ближайшее время, выгружается на внешний носитель. Так, в Windows NT на выбранных разделах создаются страничные файлы pagefile.sys, в которые и записываются неиспользуемые в настоящее время страницы. В некоторых других операционных системах для таких нужд создаются целые отдельные разделы.

1.4. Многозадачность

Существует другой вид виртуализации – многозадачность. Вытесняющая многозадачность, при которой сама операционная система решает, когда и сколько процессорного времени отдать задаче, у Microsoft в ОС Windows впервые появилась в версиях 3.x, в следующих – многократно совершенствовалась. И то совпадение, что начало развития виртуальной памяти совпадает с началом развития многозадачности, не случайно – эти два понятия очень тесно связаны и вместе составляют мощный инструмент управления ресурсами компьютера. За более чем 20 лет этот механизм виртуализации претерпел значительные изменения, появилось много достойных реализаций в других операционных системах.

1.5. Процесс в ОС

Попробуем раскрыть понятие процесса для последних версий операционных систем серии Windows NT: Windows 2000 и последующих версий.

В общем случае задача, или процесс – это выполнение инструкций компьютерной программы на процессоре ЭВМ. В нашем случае, процесс представляет собой сложную структуру, которая включает:

- структуру данных, содержащую всю информацию о процессе, в том числе список открытых дескрипторов различных системных ресурсов, уникальный идентификатор процесса, различную статистическую информацию и т.д.;
- адресное пространство - диапазон адресов виртуальной памяти, которым может пользоваться процесс;
- исполняемую программу и данные, проецируемые на виртуальное адресное пространство процесса. К таким данным относятся исполняемые файлы (EXE) и файлы динамических библиотек (DLL), отображаемые в адресное пространство процесса специальным образом, память стека и память, динамически выделенную из кучи.

1.6. PE файл

Portable Executable – формат исполняемых файлов, объектного кода и динамических библиотек, используемый в 32- и 64-битных версиях операционной системы Microsoft Windows. Формат PE представляет собой структуру данных, содержащую всю информацию, необходимую загрузчику для запуска программы, которая находится в данном файле. Наиболее часто встречаются три вида файлов в формате PE: исполнимые модули (*.EXE),

динамически подключаемые библиотеки (*.DLL), драйверы устройств, работающие в режиме ядра (Kernel mode drivers).

Общеизвестно, что Windows NT многое унаследовала от VAX VMS и UNIX. И стандарт PE не является исключением – за основу был взят формат COFF (Common Object File Format — стандартный формат объектного файла), который впервые был введен в UNIX. Но COFF к тому времени несколько устарел, и, чтобы удовлетворить потребностям новых операционных систем, было решено внести в него небольшие изменения и дать ему название PE.

Portable Executable – дословно переводится как “переносимый исполняемый”. Этот формат называется переносимым, так как все реализации Windows NT в различных системах (Intel 386, MIPS, Alpha, Power PC и т.д.) используют один и тот же исполняемый формат. Конечно, имеются различия, например, связанные с двоичной кодировкой команд процессора: нельзя запустить на Intel исполняемый PE-файл, откомпилированный в MIPS. Тем не менее, существенно, что нет нужды полностью переписывать загрузчик операционной системы и программные средства для каждого нового процессора.

Самое важное из того, что следует знать о PE-файлах, это то, что исполняемый файл на диске и модуль, получаемый после загрузки, очень похожи. Причиной этого является то, что загрузчик попросту использует отображение файлов в память, чтобы загрузить соответствующие части PE-файла в адресное пространство программы. После того как EXE (или DLL) модуль загружен, Windows обращается с ним так же, как и с другими отображенными в память файлами.

Еще одним важным понятием является секция. Именно из секций состоит большая часть PE файла. Они содержат либо код программы, либо данные. Некоторые секции содержат код и данные, непосредственно объявляемые и используемые программами, тогда как другие секции данных создаются компоновщиками специально для операционной системы и содержат информацию, необходимую для правильной загрузки программы в память.

Для работы с PE-файлами используются три различных схемы адресации: физические адреса (называемые также смещениями raw pointers или raw offset), отсчитываемые от начала файла, виртуальные адреса (virtual address или VA), отсчитываемые от начала адресного пространства процесса и относительные виртуальные адреса (relative virtual address или RVA), отсчитываемые от базового адреса загрузки модуля.

Итак, структура PE-файла имеет следующий вид (см. рис. 3):

- В начале файла расположен заголовок MS DOS,

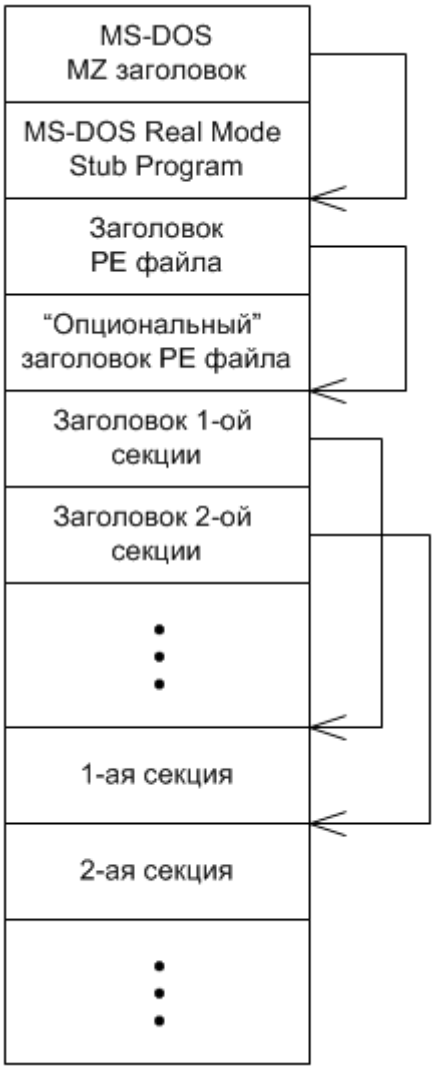

рис. 3

- Заголовок PE, смещение которого от начала файла хранится в заголовке MS DOS,
- “Опциональный” заголовок PE-файла,
- Заголовки секций, количество которых указано в заголовке PE,
- Сами секции, смещенные в файле на значение, указанное в заголовке секции.

Заголовок MS-DOS и программа-заглушка (Stub Program) служит только для выдачи сообщения о том, что программа не может работать в среде MS-DOS, при попытке ее заставить работать в этом режиме. Заголовок представляет собой запись типа IMAGE_DOS_HEADER (структура, объявленная в WinNT.h). Поле e_lfanew – ключевое в этой структуре, оно содержит смещение (от начала файла) основного PE-заголовка.

Итак, отступив e_lfanew байт от начала файла, мы получим основной заголовок PE файла. Он полностью описывается структурой IMAGE_NT_HEADERS:

- Первые 4 байта в этой структуре (поле Signature) – сигнатура “PE\0\0”, которая подтверждает, что это действительно файл PE.
- Главный заголовок PE. Структура IMAGE_FILE_HEADER. Она содержит информацию
  - о количестве секций (поле NumberOfSections),
  - о размере “опционального” заголовка (поле SizeOfOptionalHeader),
  - о типе центрального процессора, под который скомпилирован файл (поле Machine, для i386-машин должен быть выставлен в 0x14C).
  - …
- “Опциональный” заголовок PE. Структура IMAGE_OPTIONAL_HEADER. Неизвестно, почему этот заголовок назвали опциональным – без него загрузка файла абсолютно невозможна. В нем содержится
  - абсолютный базовый адрес загрузки модуля (поле ImageBase), отсчитываемый от начала сегмента; в терминологии спецификации, preferred address (предпочтительный адрес загрузки),

- RVA адрес точки входа (поле AddressOfEntryPoint),
- кратность выравнивания секций на диске и в памяти (поля FileAlignment / SectionAlignment),
- объем зарезервированной / выделенной памяти под стек и кучу в байтах,
- таблица DATA_DIRECTORY, содержащая указатели на такие структуры как таблицы импорта и экспорта в модуле, таблицу ресурсов, таблицу релокаций и прочие. Из-за того, что документация не задает жестко размер этой структуры, размер всего “опционального” заголовка необходимо определять по соответствующему полю в главном заголовке.

Сразу после “опционального” заголовка один за другим идут заголовки секций. Каждый заголовок представляет собой структуру IMAGE_SECTION_HEADER. Важные поля в этой структуре:

- PointerToRawData – смещение секции относительно начала файла,
- SizeOfRawData – размер секции в файле,
- VirtualAddress – RVA-адрес начала секции в памяти,
- Misc.VirtualSize – размер секции в памяти,
- Characteristics – 4-х байтное поле, определяет атрибуты доступа к секции и особенности ее загрузки. Отметим самые важные из них:
  - Первая группа атрибутов описывает содержимое секции
    - IMAGE_SCN_CNT_CODE (0x20) – как секцию кода,
    - IMAGE_SCN_CNT_INITIALIZED_DATA (0x40) – как секцию инициализированных данных,
    - IMAGE_SCN_CNT_UNINITIALIZED_DATA (0x80) – как секцию неинициализированных данных.

    Стоит отметить, что эти атрибуты никак не влияют на загрузку файла: секция неинициализированных данных загружается в память так же, как и инициализированных данных, в секции кода могут содержаться данные, а в секции данных может содержаться исполняемый код.
  - Следующий набор атрибутов настраивает права доступа, проставляемые загрузчиком всем страницам секции в памяти:
    - IMAGE_SCN_MEM_EXECUTE (0x20000000) – права на исполнения,
    - IMAGE_SCN_MEM_READ (0x40000000) – права на чтение,
    - IMAGE_SCN_MEM_WRITE (0x80000000) – права на запись.

Важная особенность процессоров фирмы Intel: при 32-битной страничной адресации атрибуты чтения и исполнения полностью эквивалентны [7].

- IMAGE_SCN_MEM_DISCARDABLE (0x2000000) – после загрузки файла в память секция может быть выгружена. Обычно этот атрибут проставляется секциям, содержащим вспомогательные для загрузки данные (например, таблица релокаций).
- IMAGE_SCN_MEM_SHARED (0x10000000) – секция является разделяемой между процессами, использующими этот модуль.

Вот мы рассмотрели структуру заголовка секции. По нему системный загрузчик находит в PE файле секцию и определяет, в какую область виртуальной памяти и как ее необходимо отобразить стандартными средствами мэппинга [13].

Глава 2. Постановка задачи

В этой работе мы рассмотрим метод модификации памяти процесса и генерации новых последовательностей машинных инструкций для того, чтобы иметь возможность оценить частоту исполнения тех или иных участков кода программы.

Итак, у нас исходный текст некоторой программы, и мы хотим определить, какие процедуры и функции в нем исполняются чаще всего, и, соответственно, те, которые стоит стараться оптимизировать.

Метод будет работать непосредственно внутри процесса, который подвергается анализу. Вся логика будет реализована в одной динамически подключаемой библиотеке, которая будет загружаться в память по инициативе самого тестируемого кода (т.е. в него нужно будет добавить соответствующие команды).

Метод накладывает следующие ограничения:

- на аппаратную платформу:
  - архитектура микропроцессора x86;
  - соответственно 32-битная адресация памяти;
- на операционную систему:
  - ОС Windows NT начиная с версии Windows 2000;
- на анализируемый исполняемый модуль:
  - должен быть PE файлом (DLL или EXE);
  - исполняемый код не должен изменяться (код не должен быть самомодифицирующимся);
  - исполняемый код должен находиться в секциях кода PE файла и только там;
  - исполняемый код должен состоять только из инструкций общего набора x86 (инструкции расширений, таких как MMX, SSE, 3DNow! – недопустимы).

Впрочем, ограничения, накладываемые на исполняемый модуль, можно сузить лишь до части кода, который мы хотим проанализировать. Не обязательно пытаться проанализировать весь модуль.

После работы метода, результаты должны быть записаны в файл на диск.

Глава 3. Терминология

1) Исполняемый модуль – файл, готовый к непосредственному исполнению (выполняется на процессоре или интерпретируется другой программой).
2) Тестируемый код – код, который подвергается анализу и тестированию. Генерируется компилятором.
3) Исходный код – тестируемый код, загруженный в память и находящийся в секциях кода PE-файла. При работе алгоритма блокируется вызовом функции VirtualProtect с параметром PAGE_NOACCESS (см. параграф 5.2.2, “Блокировка секций кода”).
4) Исходный текст программы – программа на одном из языков программирования (C / C++, Pascal, JAVA и т.д.)
5) T-код (T-инструкции) – код, который внедряется в тестируемый код. Собирает информацию о ходе исполнения программы.
6) Модифицированный (или сгенерированный) код – тестируемый код с внедренными в него T-инструкциями. Он уже размещен в памяти и готов к исполнению.
7) Управляющие исключения – исключения, которые инициализируются при обращении к заблокированным секциям кода (параграф 5.2.2, “Блокировка секций кода”). Нужны для управления процессом построения нового кода.
8) Глобальный обработчик исключений – функция, которая вызывается при возникновении всех исключений любого вида (параграф 5.2.1, “Глобальный обработчик исключений”). Нужен, в основном, для обработки управляющих исключений.
9) Сплайсинг [31] функции – способ перехвата вызовов функции, суть которого заключается в изменении первых ее байт. Подробнее будет описан в параграфе 5.2.1 “Глобальный обработчик исключений”.
10) Исключение типа Access Violation – исключение нарушения прав доступа. Возникает, например, при обращении к невыделенной памяти или при попытке записи в область памяти, для которой разрешено только чтение.
11) Регистр EIP – регистр процессора, содержащий адрес-смещение следующей инструкции, подлежащей исполнению.
12) Точка останова – это преднамеренное прерывание выполнения программы, при котором выполняется вызов отладчика.
13) Функция KiUserExceptionDispatcher – функция системной динамической библиотеки ntdll.dll. Является важной частью механизма обработки исключений в пользовательском режиме. Более подробное описание в параграфе 5.2.1 “Глобальный обработчик исключений”.

14) Профилирование – сбор характеристик работы программы, таких как время выполнения отдельных фрагментов (обычно подпрограмм), число верно предсказанных условных переходов, число кэш промахов, информации об использовании памяти и прочее.
15) Профайлер (профилировщик) – инструмент, выполняющий профайлинг (профилирование) программ.
16) Семплирование (в профайлинге) – статистический метод, который выявляет узкие места в производительности. Сбор сведений выполняется с определенной частотой или при возникновении определенных событий.
17) Динамическая рекомпиляция – переписывание в памяти участков кода программы во время ее исполнения. Применяется в профайлерах, эмуляторах, виртуальных машинах для добавления в исходный код дополнительных инструкций, трансляции кода одной платформы в другую и т.д.
18) Инструкции перехода – инструкции типа CALL, JMP, Jxx, LOOP / LOOPxx и RET.

Глава 4. Аналогичные продукты

Начнем обзор с классических отладчиков. В настоящее время весьма популярны следующие разработки для ОС Windows:

- IDA Pro Disassembler – интерактивный дизассемблер, который широко используется для реверс-инжиниринга. Чрезвычайно гибок и удобен в использовании.
- OllyDbg – отладчик уровня ассемблера, который отличается от всех остальных своей простотой и удобством в использовании.
- SoftICE – это универсальный [29] отладчик режима ядра, которым можно отладить любой код, включая подпрограммы прерывания и драйверы ввода-вывода.
- WinDbg – отладчик режима ядра, разработанный компанией Microsoft.

Как ни странно, они тоже занимаются модификацией исполняемого кода в памяти: даже самый простейший отладчик умеет оперировать точками останова. Существует два вида точек останова: аппаратные [4] и программные (инструкция INT 3). Но из-за того, что количество аппаратных точек ограничено четырьмя, то чаще всего используют инструкцию INT 3. Фактически происходит внедрение этих инструкции в программный код, и с их помощью приложение останавливается в определенный момент и производится его анализ. А это и есть тема курсовой работы.

Теперь нам нужно упомянуть некоторые известные профилировщики, которые используются наиболее часто.

Intel VTune Performance Analyzer [22] – коммерческое приложение для анализа производительности программ на персональных компьютерах, использующих процессоры фирмы Intel. Инструмент выполняет множество функций профилировки:

- определяет последовательность вызовов функций и строит ее графическое изображение, что помогает выявить критические функции и временные затраты в приложении;
- использует метод семплирования для нахождения участков кода программ, которые потребляют значительную долю ресурсов процессора, или в которых происходят промахи кэширования, ошибки предсказания ветвления и другие проблемы производительности;
- позволяет просматривать ассемблерный листинг и исходные тексты программы;

- имеет профилировщик потоков, который показывает активность потоков и их взаимодействие, подсказывает, в каких местах программы возникают ошибки, связанные с реализацией многопоточности;
- прочие возможности.

Большинство из перечисленных функций выполняет и продукт AMD CodeAnalyst [2] компании AMD, хотя по возможностям до профайлера VTune ему еще очень далеко. В отличие от VTune, он выполняет профилировку только на процессорах AMD. Является бесплатным.

Рассмотрим другие профиляторы, в основе которых лежит динамическая рекомпиляция. Вот некоторые из них:

- DynInst [3] – мультиплатформенная библиотека патчинга исполнимого кода. Она позволяет вносить изменения в уже работающую программу, тем самым, обходясь без перекомпиляции исходного текста программы. Работает на платформах Power/PowerPC, SPARC, x86, IA-64. Его API используется многими утилитами [20], которые специализируется на анализе производительности программ, использования ими памяти и других видов профилирования.
- Valgrind [1] – схожий инструмент для ОС Linux, который сначала преобразует двоичный код программы в более простую для себя форму, затем применяет к этому промежуточному коду один из ряда инструментов и транслирует обратно в машинный код. Некоторые из доступных инструментов [21]:
    - Memcheck – позволяет находить проблемы с работой с памятью (утечки памяти, попытки использования неинициализированной памяти и прочее);
    - Massif – профилировщик кучи;
    - Helgrind и DRD – способны отслеживать состояние гонки и подобные ошибки в многопоточном коде;
    - Cachegrind — профилировщик кэша.

Видим, что динамическая перекомпиляция успешно применяется для анализа производительности программ и анализа использования памяти (поиск утечек и др.). На самом деле, использовать технику перекомпиляции кода можно гораздо шире. В качестве примера можно отметить виртуальные машины, такие как VMware и VirtualBox, которые используют эту технику для эмуляции привилегированных инструкций [19].

Глава 5. Идея метода

Основная идея метода заключается в модификации секций кода PE файлов, загруженных в память, добавлении необходимых нам инструкций таким образом, чтобы отредактированная программа сама собирала всю статистику, что нас интересует.

5.1. Способы внедрения

Записать необходимые инструкции в исполняемый код можно несколькими способами:

- непосредственно модифицируя секцию кода в PE файле на диске,
- редактируя страницы памяти, содержащие исполняемый код, после загрузки PE файла,
- комбинация двух предыдущих способов.

5.1.1. Принципиально возможные способы внедрения

Начать можно со способа, который включает в себя физическое изменение PE файла на жестком диске. Т.е. еще незагруженный файл мы должны проанализировать, добавить в него наш код, отправить на исполнение и некоторым способом получать необходимую нам информацию. В этом случае файл модифицируется только однажды, но технически реализовать это крайне сложно из-за того, что полный разбор генерируемого сборщиками кода – крайне тяжелая задача, требующая написания эмулятора исполнения, и даже если такая работа будет произведена, это не обережет программу от возможных ошибок, возникающих после добавления в нее новых инструкций. Необходим контроль над действиями программы, которые могут привести к ее краху, и своевременное предупреждение пользователя о них.

Далее рассмотрим способ, в котором изменения вносятся в адресное пространство процесса, т.е. уже после загрузки исполняемого модуля в память. В этом случае нам открывается огромное множество средств для достижения нашей цели.

Один из возможных способов внедрения в чужой код – непосредственное редактирование физической памяти или страничного файла (файла подкачки). Этот способ сильно привязан к архитектуре системы, и для реализации нужно иметь полное представление о принципах ее работы. Итак, предположим, что программа загружена в память, и мы хотим внести в ее исполняемый код некоторые изменения. Начнем с того, что в любой момент времени, за исключением некоторых случаев (ОС может по тем или иным причинам запрещать выгрузку страниц), необходимая нам страница может находиться как в оперативной памяти, так и быть выгруженной на диск. Следующая проблема заключается в

необратимости процесса трансляции виртуального адреса в физический, который был описан в параграфе 1.3.3 “Страничная организация памяти”. Архитектура процессора позволяет проводить преобразование только в одну сторону, в обратную сторону аппаратной реализации не существует (для промышленных нужд не нужна), поэтому придется делать это самим. Из-за таких особенностей системы, задача значительно усложняется. К счастью, в Windows NT существует достаточно средств, чтобы избежать столь близкого общения с операционной системой. К тому же, в Windows прямой доступ к физической памяти (объект Device\PhysicalMemory) из пользовательского режима был запрещен [30]. Из всего сказанного можно заключить, что этот метод можно успешно применять [25] разве что в вирусах и троянах, потому что им не нужно производить очень тщательный анализ памяти: чаще всего они обходятся поиском заранее определенных сигнатур.

5.1.2. Реализуемый метод

В способе, которому посвящена эта работа, мы не будем пользоваться такими низкоуровневыми особенностями системы и писать собственные драйвера.

Применять метод можно будет:

- для собственных программ (в исходный текст тестируемой программы вносятся незначительные изменения)
- для сторонних программ (не имеем доступа к исходным текстам тестируемой программы)

В первом случае наш код будет загружаться в адресное пространство тестируемого процесса средствами ОС как DLL модуль с помощью вызова системной API функции – LoadLibrary (этот вызов должен производиться самой тестируемой программой). Затем при исполнении процесс будем останавливать, анализировать с текущего положения (положение определяется регистром EIP) насколько это возможно, создавать копию этой части кода с необходимыми нам изменениями и продолжать выполнение уже на созданной последовательности инструкций, ожидая пока процессор не передаст исполнение на неанализированный код, что повлечет за собой очередной запуск алгоритма анализа бинарного исполняемого кода. Более подробно процесс внедрения будет описан далее.

Во втором случае достаточно особым способом загрузить вышеуказанную DLL в адресное пространство процесса. Это будет производиться с использованием системных API: выделение памяти в чужом адресном пространстве (VirtualAllocEx), запись данных в эту память (WriteProcessMemory) и создание потока, принадлежащего чужому процессу (CreateRemoteThread). В работе этот вариант использования метода мы рассматривать не будем.

5.2. Метод модификации кода

В нашем методе мы не будем изменять исполняемый код – мы будем его писать в памяти заново, только с нужными нам изменениями. Опишем более подробно этапы работы метода.

5.2.1. Глобальный обработчик исключений

Первым шагом метода будет созданием процедуры, которая получает управление при любом возникающем в процессе исключения.

Существует два механизма Win32 обработки исключений:

- SEH [16] – механизм структурной обработки исключений (Structured Exception Handling). Старейший и наиболее широко используемый механизм.
- VEH [17] – механизм векторной обработки исключений (Vectored Exception Handling). Сравнительно новый механизм. Был введен в Windows XP.

Можно найти множество статей для обоих из этих механизмов, поэтому их описание в этой работе будет опущено.

Весь цикл жизни [9] исключений в системе выглядит так:

1. Процессор генерирует исключение.
2. Ядро операционной системы перехватывает и обрабатывает сгенерированное исключение.
3. Управление передается функции KiUserExceptionDispatcher из системной библиотеки ntdll.dll, расположенной на прикладном уровне. В качестве параметров в стек помещаются структуры, содержащие описание исключения (EXCEPTION_RECORD) и контекст потока на момент генерации процессором исключения (CONTEXT).
4. Далее приложение, используя один из двух механизмов – SEH или VEH, обрабатывает исключение (или не обрабатывает, что приводит к аварийному завершению программы)

Из этого больше всего нас будет интересовать функция KiUserExceptionDispatcher. Все исключения, которые попадают на пользовательский уровень, проходят через эту функцию. Следовательно, нужно научиться перехватывать ее вызовы.

Введем понятие сплайсинга [31] функций. Грубо говоря, сплайсинг это подмена кода функции. Этот метод часто используется при перехвате вызовов API функций: определяется адрес перехватываемой функции, и её первые 5 байт заменяются длинным JMP – переходом по адресу обработчика.

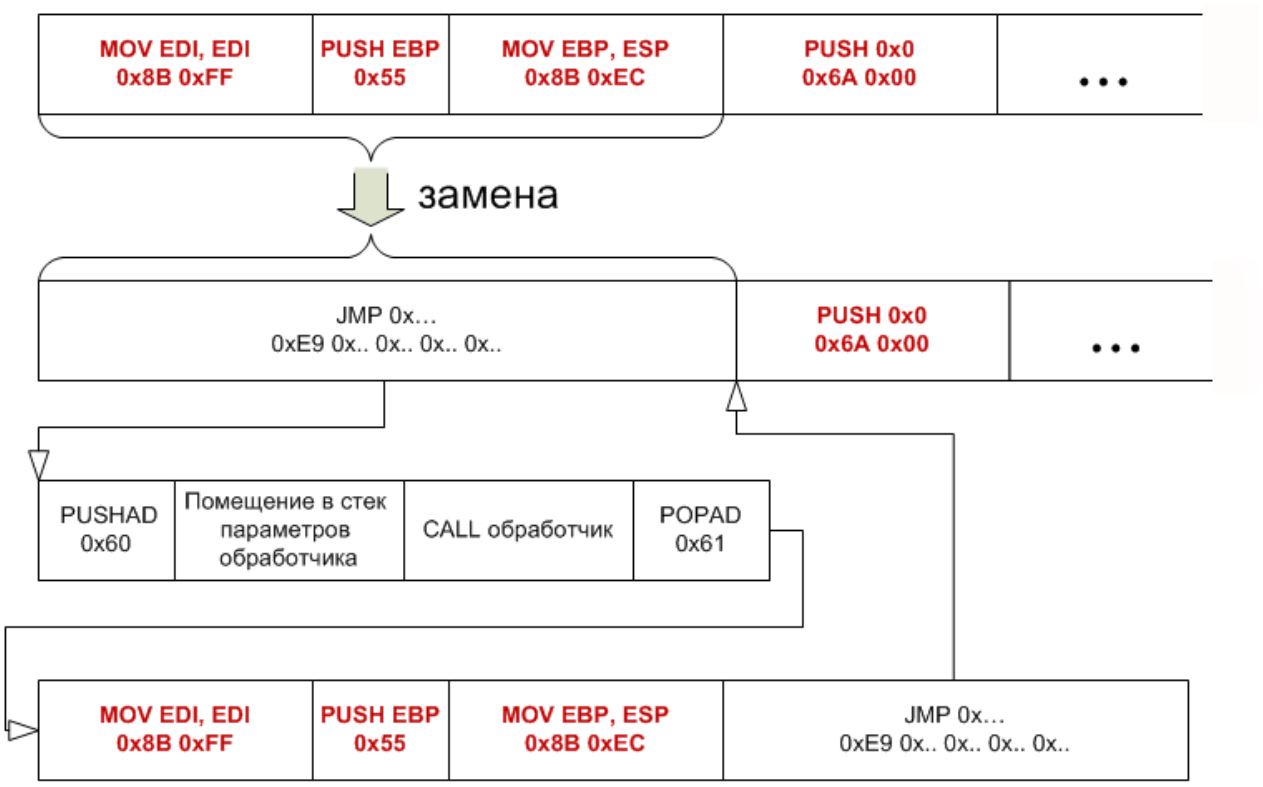

рис. 4

На схеме (см. рис. 4) видно, что первые три инструкции функции заменяются на одну пятибайтовую инструкцию JMP – переход уже на наш код, в котором можно сделать все что угодно. На примере, изображенном на схеме, мы сначала сохраняем все регистры в стеке инструкцией PUSHAD (при необходимости можно еще сохранять там же флаги – инструкция PUSHF), вызываем обработчик, предварительно поместив в стек его параметры, восстанавливаем значения регистров (инструкция POPAD), а затем начинаем исполнение перекрытых инструкций и остального кода, как будто и не было никакого внедрения.

Этим приемом воспользуемся и мы. Чтобы перехватить все пользовательские исключения, возникающие в процессе, применим метод сплайсинга API функций для функции KiUserExceptionDispatcher.

5.2.2. Блокировка секций кода

Рассмотрим функцию VirtualProtect [18] из библиотеки kernel32.dll. Она позволяет изменять атрибуты защиты указанного региона виртуального адресного пространства. В качестве одного из параметров ей передается 4-х байтовое число – атрибуты защиты [15]:

- PAGE_NOACCESS – запрещен любой вид доступа,
- PAGE_READWRITE – чтение и запись,
- PAGE_READONLY – только чтение,
- PAGE_EXECUTE – только исполнение программного кода,

- PAGE_EXECUTE_READ – исполнение и чтение,
- PAGE_EXECUTE_READWRITE – исполнение, чтение и запись,
- …

На всякий случай еще раз повторимся, что при 32-битной страничной адресации на процессорах фирмы Intel атрибуты чтения и исполнения полностью эквивалентны, поэтому невозможно запретить чтение при установленных правах на исполнение и наоборот.

Итак, при запуске PE файла, каждой секции, загружаемой в память, проставляются атрибуты защиты, которые зависят от соответствующего параметра в заголовке секции. Этот же параметр определяет, содержит ли секция код или данные (по правде сказать, если проставлен флаг секции кода, то это еще ничего не значит, поэтому будем полагаться на добросовестность компоновщика).

Итак, вторым этапом в методе будет проставление всем страницам секций кода атрибута PAGE_NOACCESS, при этом любое обращение к ним будет инициализировать в процессоре исключение #GP или Access Violation на пользовательском уровне, которое мы сможем перехватить с помощью нашего глобального обработчика исключений.

5.2.3. Инициализация управляющего исключения

После блокировки секций кода, любое обращение к страницам памяти, относящимся к этим секциям, будет вызывать исключение. Поэтому во всех потоках процесса, в которых в данный момент исполняется код данного модуля, будет инициализировано управляющее исключение и управление передастся глобальному обработчику исключений.

5.2.4. Перехват управляющего исключения

Предположим, что глобальный обработчик поймал исключение. Проверим, что:

- исключение является исключением нарушения доступа (Access Violation),
- адрес возникновения исключения совпадает с адресом, обращение к которому вызвало это исключения (из чего можно сделать вывод, что система пыталась выполнить код, расположенный по этому адресу).

Если не выполнено первое условие, то исключение было вызвано чем-то другим в программе, не из-за выполненных модификаций. Исключение необходимо передать дальше функции KiUserExceptionDispatcher.

Если же не выполнено второе условие, то нужно проверять адрес, к которому происходило обращение. В случае если адрес указывает не на секцию кода, то ошибка произошла не по нашей вине, и нужно передать исключение дальше. В противном случае

делаем вывод, что в секции кода содержатся данные, которые приложение пытается прочитать или перезаписать. Тут нам ничего не остается, как снять запрет на доступ к странице или завершить исполнение приложения.

Итак, если оба условия выполнены, то делаем вывод о том, что нам пришло управляющее исключение.

5.2.5. Пошаговый анализ тестируемого кода

Следующий шаг начинается сразу после того, как мы осознали, что перехватили управляющее исключение. И из этого исключения мы извлекаем информацию об адресе его возникновения. Возможно два варианта:

- код по этому адресу обрабатывался,
- код по этому адресу не обрабатывался.

Начнем со второго случая – код не обрабатывался. Тогда начинаем анализ машинного кода, начиная с этого адреса. Конечно, нужно предварительно сохранить отдельно все содержимое секций кода (из-за того, что из заблокированных страниц чтение будет запрещено и нашему коду). Имея на руках дизассемблер длин инструкций (“аппарат”, который рассматривает машинный код процессора только для оценки количества байт, которые занимает текущая команда, и определяет, когда начинается следующая инструкция), будем просматривать команду за командой, уделяя особое внимание операторам перехода:

- JMP – инструкция безусловного перехода,
- Jxx – множество (более 30) различных видов инструкций условного перехода,
- CALL – безусловный переход, при котором в стек помещается адрес следующей инструкции (адрес возврата),
- RET – переход по адресу, который лежит на вершине стека (адрес удаляется из стека),
- LOOP / LOOPxx – инструкция, реализующая цикл на уровне инструкций.

Для всех переходов существует два способа использования: переход по фиксированному и вычисляемому адресу. Когда адрес фиксирован, то на этапе анализа мы можем проанализировать и код, на который осуществляется переход. Другая ситуация обстоит с вычисляемыми переходами (например, инструкции ”CALL EAX” или ”JMP [ECX + 0x4]”). Их можно встретить, например, в реализации виртуальных методов. В этом случае дальнейший статический анализ невозможен. Но в этом нет ничего страшного: переход осуществляется по абсолютному адресу, и на этот абсолютный адрес перемещение кода никак повлиять не может (существуют небольшие нюансы в этом утверждении, о них чуть позже). Т.е. если выполнение того же кода будет происходить по другому адресу, то

адрес перехода не изменится, и после перехода в секцию кода снова будет инициализировано управляющее исключение, после чего можно продолжать анализ.

Важно учитывать участки в исходном коде, которые уже были проанализированы. При инициации управляющего исключения на них, повторную обработку проводить не нужно. Для этого будем некоторым образом сопоставлять адреса инструкций в PE файле и адреса инструкций в коде, который был сгенерирован во время анализа (с внедренными T-инструкциями), и при перехвате вышеупомянутого исключения достаточно будет просто найти соответствующий адрес в сгенерированном коде и передать на него управление.

5.2.6. Внедрение T-кода

Мы описали процесс анализа кода. Параллельно с ним будет проводиться построение модифицированного кода. Такой код должен работать точно так же, как и соответствующий ему код в PE-файле, но при этом должен содержать дополнительные инструкции, задачей которых и будет произведение необходимых вычислений, инкрементирование счетчиков.

При тестировании программы мы будем в ней для каждой инструкции определять количество раз, сколько она была исполнена. Для этого в код будут встраиваться команды, которые будут увеличивать заранее созданные счетчики. Но мы не будем вставлять T-инструкции перед каждой инструкцией в тестируемом коде – это очень сильно скажется на скорости исполнения программы. Можно разработать множество алгоритмов и структур данных для размещения и быстрого обращения к этим счетчикам. При этом нужно учитывать следующие аспекты:

- какую информацию необходимо собирать
    - информацию о переходах (сколько раз, откуда и куда производились переходы) – является, пожалуй, наиболее полным анализом;
    - информацию о частоте исполнения каждой инструкции;
    - другое…
- будет ли статистика собираться отдельно для каждого потока в процессе или она будет объединяться для них;
- накладные расходы
    - память для хранения всех структур данных;
    - время на анализ кода;
    - время, затрачиваемое на инкрементацию счетчиков;
    - время, которое теряется при переходе из одного блока модифицированного кода в другой (изначальную структуру

исполняемого кода сохранить не получится, поэтому связывание участков, возможно, происходит динамически);

- сложность дополнения модифицированного кода новым проанализированным кодом или дополнительными счетчиками;

5.2.7. Продолжение исполнения программы

Вот мы построили и записали в память код с необходимыми нам модификациями. Осталось только передать на него исполнение – сбор информации будет происходить автоматически.

5.2.8. Маскировка внесенных изменений

Важный аспект в подобном внедрении – насколько изменения незаметны для тестируемой программы, и что достаточно сделать, чтобы обнаружить вторжение.

В нашем случае, из-за серьезных ограничений на анализируемую программу и из-за очень явных изменений в коде, программа, которая специально делает проверки целостности, наверняка определит, что к ней кто-то прицепился. Но у нас немного другая задача: внести в код необходимые нам изменения так, чтобы с большой вероятностью он остался рабочим.

Рассмотрим два случая, которые могут привести к крушению программы.

1) После того, как исполнение кода переместилось из секций кода PE файла в память, выделенную из кучи, изменилось поведение инструкций CALL – они стали помещать в стек не те адреса, что раньше. Конечно, маловероятно, что эти изменившиеся адреса на что-то повлияют, но не исключено, что какой-то очень хитрый компилятор не решит проверять их, чтобы “побороть” возможную уязвимость переполнения буфера.

Решение. Все CALL инструкции заменять на сочетание инструкций PUSH и JMP – такая замена эквивалентна, если не учитывать флаг NT, который используется для переключения задач [28]. После того, как выполнится соответствующая инструкция RET, управление передастся на заблокированный код, потому что в стек инструкцией PUSH был помещен такой адрес возврата, какой он был бы, если не было бы никаких модификаций. Ну а дальше будет вызвано управляющее исключение, и можно продолжать исполнение программы.

2) Второй случай заключается в том, что модифицированный код по-прежнему способен генерировать исключения, как штатные, так и внештатные. И лучше приложению не показывать реальный адрес, по которому возникло исключение.

Решение. Будем просто в глобальном обработчике подменять адрес возникновения исключения и сохраненный регистр EIP. Для этого должна быть реализована обратная трансляция адреса из модифицированного кода в адрес в исходном коде.

5.2.9. Уменьшение числа управляющих исключений

Активное использование исключений в приложениях может сказаться на ее производительности, а откровенное злоупотребление может замедлить так, что программа будет зависать на ровном месте. Это вызвано тем, что выполняется много мелкой работы: проверки и определение, кому дальше передавать исключение, запись в стек состояния потока (регистры, флаги) и информации об исключении, переключение между режимом ядра и пользовательским режимом и проч. В результате тратятся тысячи тактов процессора, что, при некоторых обстоятельствах, может стать очень существенным.

В конце параграфа 5.2.5 “Пошаговый анализ тестируемого кода” упоминалось о сопоставлении адресов инструкций в исходном и модифицированном кодах. Это мы и будем использовать для оптимизации использования исключений.

При использовании инструкций переходов по вычисляемым адресам и, учитывая предыдущий параграф, инструкции RET, происходит переход на код, который принадлежит заблокированной секции кода PE файла, что сразу же вызывает управляющее исключение.

В качестве решения этой проблемы можно использовать такой способ: вместо того, чтобы совершить переход по вычисленному адресу, будем помещать этот адрес в стек (кстати, для случая с инструкцией RET адрес перехода уже находится в стеке) и вызывать нашу собственную процедуру, в которой будем искать соответствующий адрес в модифицированном коде и передавать на него управление. Стоит отметить, что поиск нужного адреса в любом случае пришлось бы делать, но в таком случае мы не тратим время на ненужную обработку исключения в ядре системы.

5.2.10. Визуализация результатов профилировки

Итак, наполнив тестируемый код нашими инструкциями, мы получим аналогичную программу, но она, в дополнение ко всему, собирает статистическую информацию о ходе своего исполнения.

Осталось только сделать ее вывод в удобном виде. Тут есть множество вариантов и их комбинаций:

- вывод в файл;
- вывод статистики на экран в числовом виде, отображение графиков;

- вывод ассемблерного листинга, а при наличии отладочной информации делать соответствующие пометки;
- вывод статистики о вызовах функций, расположенных в других исполняемых модулях (количество вызовов, параметры и результаты вызовов).

Глава 6. Реализация метода

Программирование описанного метода будет происходить в среде разработки Microsoft Visual Studio 2008 с использованием языка C++ (unmanaged). Также будут использоваться ассемблерные вставки. Отметим, что в 64-битном компиляторе VC++ ассемблер был запрещен [14].

Как уже упоминалось, вся логика метода будет запрограммирована в одном DLL файле для того, чтобы без особого труда загрузить в адресное пространство процесса весь код, что мы будем использовать для анализа.

6.1. Глобальный обработчик исключений

Итак, при загрузке DLL файла в память происходит вызов функции DllMain [12]. DllMain – дополнительная точка входа в динамически-подключаемую библиотеку, управление на которую передается тогда, когда процесс или поток инициализируется или завершает работу, а также при вызове функций LoadLibrary и FreeLibrary.

Синтаксис функции DllMain:

```
BOOL WINAPI DllMain(
  __in  HINSTANCE hinstDLL, // дескриптор модуля DLL
  __in  DWORD fdwReason,    // причина вызова функции
  __in  LPVOID lpvReserved  // зарезервированный
);
```

Вот в этой функции мы и будем производить внедрение перехватчика исключений, когда наш DLL файл загружается в память процесса (ну и, конечно, нельзя забывать про снятие этого обработчика при выгрузке DLL из памяти).

Сначала выделим кусочек памяти, необходимый для сплайсинга функции KiUserExceptionDispatcher, и частично проинициализируем его указанными инструкциями:

```
UCHAR code[0x40] = {
 0xFF, 0x74, 0x24, 0x04,       // 0x00: PUSH [ESP + 4]
 0xFF, 0x74, 0x24, 0x04,       // 0x04: PUSH [ESP + 4]
 0x00, 0x00, 0x00, 0x00, 0x00, // 0x08: CALL handler
 0x85, 0xC0,                   // 0x0D: TEST EAX, EAX
 0x74, 0x0F,                   // 0x0F: JZ not_processed
 0x6A, 0x00,                   // 0x11: PUSH 0x00
 0xFF, 0x74, 0x24, 0x08,       // 0x13: PUSH [ESP + 8]
 0x00, 0x00, 0x00, 0x00, 0x00, // 0x17: CALL NtContinue
 0xCC, 0xCC, 0xCC, 0xCC,       // 0x1C: code alignment
                               // not_processed:
 0x00, 0x00, 0x00, 0x00        // 0x20: instructions from
                               // KiUserExceptionDispatcher
                               // beginning and JMP to
                               // KiUserExceptionDispatcher
                               // continue
};
```

Итак, первые две инструкции записывают в стек аргументы для нашего глобального обработчика исключений. Вот его сигнатура:

```
BOOL WINAPI GlobalExceptionHandler(
  PEXCEPTION_RECORD pExcptRec,
  CONTEXT* pContext
);
```

Далее оставлено пять свободных байт для инструкции, которая передаст управления обработчику. Первый байт – 0xE8 (опкод инструкции CALL), а байты со второго по пятый должны содержать относительный адрес функции-обработчика. Его можно вычислить так: <абсолютный адрес функции> – <адрес инструкции CALL> – 5.

Далее происходит проверка возвращенного значения. Если это значение не ноль, то будет вызвана функция NtContinue из библиотеки ntdll.dll, которая восстанавливает состояние, в котором находился поток во время возникновения исключения. Это сохраненное состояние, состоящее из регистров и флагов, мы с легкостью могли изменить в глобальном обработчике. Если же функция вернула ноль, то будут произведен переход по метке “`not_processed`”. Поместим под этой меткой первые инструкции функции KiUserExceptionDispatcher, которые нам придется перекрыть одной инструкцией JMP, занимающей пять байт. И после этого блока инструкций поместим команду JMP для продолжения исполнения функции KiUserExceptionDispatcher.

Осталось только, используя API функцию VirtualProtect, разрешить запись в область памяти, в которой находится KiUserExceptionDispatcher, и переписать первые пять байт, заменив их инструкцией JMP, осуществляющую переход на начало блока `code`, который был описан выше.

Таким образом, любое исключение, возникшее в процессе, будет тут же перехвачено и передано глобальному обработчику.

Важно будет упомянуть о том, что на первые пять байт функции KiUserExceptionDispatcher может прийтись нецелое число инструкций. Поэтому мы будем использовать дизассемблер длин (см. параграф 6.2 “Дизассемблер длин инструкций”) для определения инструкций, которые будут перекрыты командой JMP. Также нужно рассмотреть случай, когда среди них встречается инструкция перехода. Вероятность этого близка к нулю, но все же необходимо добавить соответствующие проверки, чтобы не допустить сплайсинг и сообщить пользователю о невозможности внедрения обработчика.

6.2. Дизассемблер длин инструкций

Очень важной составляющей частью анализатора является дизассемблер длин инструкций. Это одна функция, которая определяет, какое количество байт занимает

текущая инструкция и с какой позиции начинается следующая. Заметим, что при установке глобального обработчика исключений этот инструмент также необходим. Если конкретнее, то мы его используем для определения первых инструкций функции KiUserExceptionDispatcher, которые будут замещены инструкцией JMP.

Для поставленной задачи будем использовать стороннюю разработку [23]. Она представляет собой одну функцию на языке C++:

```
bool GetInstructionSize(
  PBYTE pOpCode,
  PDWORD pdwInstructionSize
);
```

Функция принимает в качестве параметров указатель на начало инструкции в памяти и указатель на 4-х байтовую переменную, в которую будет записываться результирующая длина инструкции. Возвращаемое значение – успешно ли определена длина.

6.3. Фильтрация управляющих исключений

В параграфе 6.1 “Глобальный обработчик исключений” была упомянута сигнатура глобального обработчика: в качестве параметров функции передается указатель на структуру `EXCEPTION_RECORD` (содержит информацию об исключении) и указатель на структуру `CONTEXT` (сохраненный контекст потока).

Необходимо научить обработчик выделять управляющие исключения из общей массы исключений, что возникают в процессе.

Во-первых, исключение должно быть типа Access Violation:

```
if (pExcptRec -> ExceptionCode == EXCEPTION_ACCESS_VIOLATION)
```

Константа `EXCEPTION_ACCESS_VIOLATION` определена в заголовочном файле WinNT.h и равна 0xC0000005.

Во-вторых, адрес возникновения исключения должен совпадать с адресом, обращение по которому вызвало ошибку:

```
if ((DWORD)(pExcptRec -> ExceptionAddress) ==
     pExcptRec -> ExceptionInformation[1]))
```

Если и это условие выполнено, то проверим, что адрес исключения содержится внутри какой-нибудь заблокированной секции кода. Только после этого можно утверждать, что исключение является управляющим.

6.4. Внедрение T-кода

Итак, перехватив управляющее исключение и определив адрес инструкции, которая его инициализировала, начинаем анализ исходного кода и построение модифицированного кода, начиная с этой инструкции.

6.4.1. Анализ исходного кода

Как уже упоминалось в параграфе 5.2.5 “Пошаговый анализ тестируемого кода”, перед блокировкой страниц секций кода, необходимо создать копию данных, содержащихся в них, т.к. после блокировки доступ к ним будет запрещен как самой программе, так и нашей библиотеке.

Итак, у нас есть адрес инструкции, которая вызвала управляющее исключение. Начинаем анализ с нее.

Для поставленной задачи достаточно будет отслеживать только инструкции, исполнение которых может вызвать переход на инструкцию, находящуюся не непосредственно за этой. К такому типу инструкций относятся CALL, JMP, Jxx, LOOP / LOOPxx и RET. После выполнения на процессоре других инструкций, регистр EIP передвигается на следующую инструкцию в памяти. Существуют, конечно, инструкции SYSENTER, INT 3 и некоторые другие, которые также нарушают линейность выполнения машинного кода, но из-за особенностей построения модифицированного кода будем относить их к “обычным” инструкциям.

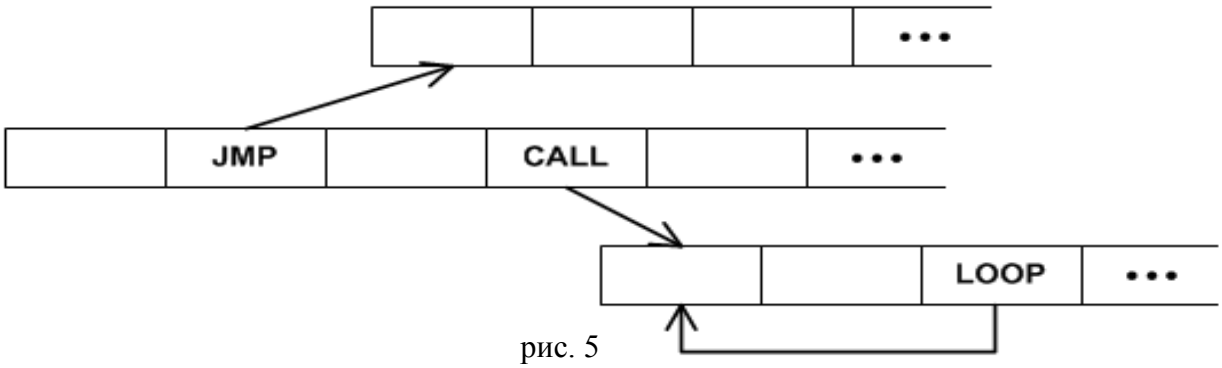

рис. 5

Сам анализ будет происходить последовательно, инструкция за инструкцией, в том порядке, в котором они находятся в памяти. И для того чтобы иметь возможность анализировать код, который находится в ветвлениях этого алгоритма, создадим список адресов, хранящий адреса инструкций, с которых следует начинать очередной процесс анализа. Изначально в этот список помещается единственный адрес – адрес инструкции, которая инициализировала управляющее исключение. В дальнейшем туда будут помещаться адреса команд, на которые осуществляется переход найденными при анализе инструкциями вышеупомянутого типа (если для них явно указан относительный адрес перехода). На каждой итерации будем извлекать из начала этого списка адрес (одновременно удаляя его оттуда). Когда список станет пустым, анализ можно считать законченным.

А от самого анализа требуется лишь поиск и классификация обозначенных выше инструкций. Эта информация будет использоваться для построения модифицированного кода.

Основной проход анализа можно найти в Приложении 5 “Процедура анализа”.

6.4.2. Построение модифицированного кода

Модифицированный код будем строить в специально выделяемой для этого памяти. Для этого будем по мере необходимости постранично выделять память. Когда очередная порция модифицированного кода не будет помещаться в последнюю выделенную страницу, то нужно выделить новую страницу памяти, записать туда модифицированный код и связать предыдущую страницу с ней инструкцией перехода JMP.

Ниже приведено описание класса `code_row`, который будет заниматься построением кода. На каждый исполняемый модуль, загруженный в память, создается не более одного объекта этого типа.

Листинг описания класса `code_row` (файл code_row.h)

```
class code_row {
private:
  typedef std::map<PVOID, PVOID> addr_map;
  page_pool pages;
  addr_map trans, trans_r;
  addr_map reloc_rel, reloc_abs;
  void *curr_page;
  int curr_len, curr_pos;

public:
  code_row();
  void add_instr(PUCHAR instr, int instr_len, PVOID instr_addr);

  void add_ret(PUCHAR instr, int instr_len, PVOID instr_addr);
  void add_call(PUCHAR instr, int instr_len, PVOID instr_addr);
  void add_jmp(PUCHAR instr, int instr_len, PVOID instr_addr);
  void add_comp_jmp(PUCHAR instr, int instr_len, PVOID
instr_addr);
  void add_loop(PUCHAR instr, int instr_len, PVOID instr_addr);
  void update_reloc();

  PVOID get_trans_addr(PVOID real_addr);
  PVOID get_trans_r_addr(PVOID addr);
};
```

Адрес в репозитории:
http://willzyx-edu-project.googlecode.com/svn/trunk/Proj1/src/profiler/code_row.h

Опишем поля и методы, что содержатся в этом классе:

- `pages` – объект, который по запросу выделяет область памяти с помощью API функции VirtualAlloc.
- `trans`, `trans_r` – ассоциативные массивы, в которых адресу инструкции в исходном коде сопоставляется адрес в модифицированном коде и наоборот соответственно.
- `reloc_rel`. Рассмотрим инструкцию “JNZ +0x3F”. Во время последовательного анализа невозможно определить новый относительный адрес перехода, потому что неизвестно, насколько увеличится объем тех инструкций, что находятся в этих 0x3F байтах исходного кода. К тому же генерация модифицированного кода может перейти на новую страницу, тогда относительный адрес невозможно предугадать. Поэтому такая инструкция будет заменяться инструкцией длинного условного перехода. Такая команда состоит из шести байт, последние четыре которой – относительный адрес перехода. Ссылки на такие 4-х байтовые относительные адреса, неопределенные на этапе анализа и построения, заносятся структуру `reloc_rel`. Им сопоставляются абсолютные адреса инструкций в исходном коде, на которые должен происходить переход, чтобы после окончания анализа, используя таблицу `trans`, вычислить необходимые относительные адреса.
- `reloc_abs` – этот объект имеет схожее назначение, что и `reloc_rel`, но, в отличие от него, служит для восстановления абсолютных адресов.
- `curr_page`, `curr_len`, `curr_pos` – указатель на участок выделенной памяти, в который происходит запись кода, его длина и текущая позиция записи соответственно.
- `add_instr`, `add_ret`, `add_call`, `add_jmp`, `add_comp_jmp`, `add_loop` – методы, которые генерируют модифицированный код. Всем им на вход передаются указатель на инструкцию (откуда можно проводить чтение), длину инструкции и ее адрес в исходном коде.
    - `add_instr` – просто копирует инструкцию;
    - `add_ret` – добавляет инструкцию типа RET (однобайтовый и трехбайтовый ее варианты);
    - `add_call` – добавляет инструкцию типа CALL, заменяет ее комбинацией инструкций PUSH и JMP (см. параграф 5.2.8 “Маскировка внесенных изменений”);

- `add_jmp` – добавляет инструкцию типа JMP;
- `add_comp_jmp` – добавляет инструкцию типа Jxx (условный переход);
- `add_loop` – добавляет инструкцию типа LOOP / LOOPxx, заменяет ее собственной реализацией на основе условных переходов (из-за того, что оригинальная инструкция может осуществлять только короткий переход). Ниже приведены эти замены (скорее для примера, т.к. возможна оптимальная реализация с помощью одной инструкции LOOP / LOOPxx и двух инструкций безусловного перехода):

табл. 1

`loop_start:`
…
`LOOP/LOOPE/LOOPNE loop_start`		

`LOOP` `0xE2`	`LOOPE` `0xE1`	`LOOPNE` `0xE0`

`loop_start:`
…
`PUSHF`
`DEC ECX`
`JZ @1`
`POPF`
`JMP loop_start`
`@1:`
`POPF`
	`loop_start:`
	…
	`PUSHF`
	`JNZ @1`
	`DEC ECX`
	`JZ @2`
	`POPF`
	`JMP loop_start`
	`@1:`
	`DEC ECX`
	`@2:`
	`POPF`
		`loop_start:`
		…
		`PUSHF`
		`JZ @1`
		`DEC ECX`
		`JZ @2`
		`POPF`
		`JMP loop_start`
		`@1:`
		`DEC ECX`
		`@2:`
		`POPF`

- `update_reloc` – вызывается по окончании процесса анализа. Проставляет относительные и абсолютные адреса инструкциям в сгенерированном коде, используя таблицу `reloc_rel`, `reloc_abs` и `trans`.
- `get_trans_addr` – переводит адрес инструкции исходного кода в адрес инструкции или комбинации инструкций в модифицированном коде.
- `get_trans_r_addr` – функция, обратная функции `get_trans_addr`.

Поэтому во время анализа для построения кода должны вызываться соответствующие функции добавления инструкций. После того как весь список ответвлений (параграф 6.4.1 “Анализ исходного кода”) будет проанализирован, необходимо обновить относительные и абсолютные адреса в сгенерированном коде, которые не были определены во время построения кода, вызовом `update_reloc`.

6.4.3. Добавление счетчиков

Итак, в предыдущем параграфе был описан принцип построение нового исполняемого кода. Теперь осталось научиться динамически внедрять полезные Т-инструкции. Не будем заботиться о сохранении быстродействия исходной программы – такой задачи перед нами не стоит.

Ниже приведено описание класса `code_counters` – класса, экземпляр которого будет содержать результаты профилировки для одного исполняемого модуля:

Листинг описания класса `code_counters` (файл code_counters.h)

```
class code_counters {
private:
     typedef std::map<UINT, INT64> addr_counters;
     addr_counters counters;
public:
     bool write_file(const char* file_name);
     void inc(UINT addr);
     void dec(UINT addr);
};
```

Адрес в репозитории:
http://willzyx-edu-project.googlecode.com/svn/trunk/Proj1/src/profiler/code_counters.h

Как видим, структура класса очень проста:

- в ассоциативном массиве `counters` хранятся значения счетчиков по соответствующим адресам;
- методы `inc` и `dec` инкрементируют и декрементируют счетчики по соответствующим адресам;
- метод `write_file` выводит всю накопленную информацию в файл в виде отчета (здесь возможно добавление отладочной информации для большей наглядности).

Общая схема работы со счетчиками будет выглядеть так:

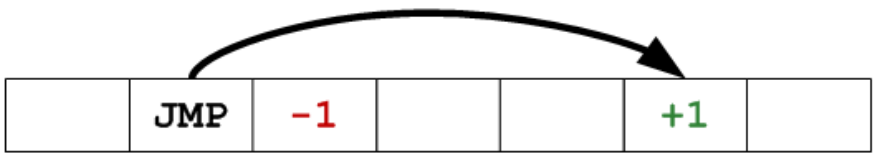

рис. 6

Таким образом, при любых переходах должен инкрементироваться счетчик по адресу инструкции, на которую производится переход, и декрементироваться счетчик по адресу байта, который находится сразу после инструкции перехода (этот байт может и не быть

байтом какой-либо инструкции). И для того, чтобы определить сколько раз была исполнена конкретная инструкция, нужно найти сумму показателей всех счетчиков, расположенных по адресам не выше адреса этой инструкции.

Для работы со счетчиками были реализованы процедуры, инструкции вызова которых будут вставляться непосредственно в код. Если не вдаваться в подробности, то эти процедуры будут вызываться в генерируемом коде вместо инструкций перехода. Перед их вызовом в стек помещаются параметры, а непосредственным переходом будет заниматься сама процедура. Т.е. теперь, фактически, любая инструкция перехода будет заменена комбинацией нескольких команд PUSH и одной командой JMP. Более подробное описание и реализацию этих функций можно найти в файлах **x_func.h** и **x_func.cpp** (Приложение 6 “Внедряемые процедуры”).

6.5. Продолжение исполнения

После завершения анализа нужно лишь найти инструкцию в модифицированном коде, которая соответствует инструкции в исходном коде, вызвавшей управляющее исключение, и передать на нее управление. Для поиска нужно использовать метод `get_trans_addr` класса `code_row`. А так как у нас есть сохраненный контекст потока (ссылка на него была передана в качестве параметра глобальному обработчику исключений), то обновим у него поле `Eip`:

```
pContext -> Eip = …
```

И тогда после восстановления контекста потока регистр EIP будет указывать на инструкцию в модифицированном коде. Следовательно, исполняться будет уже код с внедренными дополнительными T-инструкциями.

Глава 7. Результаты применения

Рассмотрим результат работы профайлера, для наглядности, на специально созданном примере. Он будет содержать несколько видов сортировок: пузырьковую сортировку, быструю сортировку (собственная и STL реализации). Полный текст тестового примера приведен в Приложении 10 “Тестовый пример”.

Рассмотрим пример более подробно.

Первое, на что можно обратить внимание, это объявление импортируемых функций нашей динамической библиотеки:

- `subscribe_me` – после вызова этой процедуры происходит блокировка всех секций кода текущего исполняемого модуля, и программа начинает исполняться под контролем профайлера.
- `unsubscribe_me` – отключаем профайлинг текущего исполняемого модуля.
- `flush_counters` – вывод отчета в файл, имя которого передается в качестве параметра.

Обратим внимание еще на директиву `#pragma auto_inline(off)`. Ей мы запрещаем компилятору автоматически для оптимизации встраивать код функции в место, где она вызывается, если мы явно его об этом не попросим (конечно, компилятор даже тогда этого может не сделать).

Массив, который будет сортироваться, состоит из 10000 4-байтных записей типа `int`. Перед каждой сортировкой он будет заполняться случайными числами, а после сортировки проверяться, правильно ли он был отсортирован.

Тестовый пример будем компилировать с параметром “Use MFC in a Shared DLL”, иначе во время исполнения будет выдано сообщение “Неизвестная инструкция”, и исполнение программы завершится. Объясняется это тем, что используемый дизассемблер длин инструкций рассчитан только на стандартный набор x86, и он ничего не знает про расширения MMX, SSE, 3DNow! и прочих. А во время исполнения STL-сортировки как раз используются инструкции Multimedia Extensions (MMX). Параметром “Use MFC in a Shared DLL” мы вынудим приложение выполнять эти инструкции в другом исполняемом модуле (другом DLL файле, который будет загружен в память).

После компиляции будет создан файл с расширением PDB. В него сохраняется отладочная информация, которая будет использоваться при создании отчета.

Начнем рассмотрение с процедуры пузырьковой сортировки. Вот ее реализация:

```
void bubble_sort(int *A, int n) {
    while (--n) {
         for (int i = 0; i < n; i++)
              if (A[i] > A[i + 1])
                   swap<int>(A[i], A[i + 1]);
    }
}
```

В отчет были выведены такие строки:

```
Test!bubble_sort
0x004011e0-0x004011e6: 1
0x004011e6-0x004011f0: 9999
0x004011f0-0x00401200: 49995000
0x00401200-0x00401215: 24995736
0x00401215-0x0040121a: 49995000
0x0040121a-0x0040121f: 9999
0x0040121f-0x00401221: 1
```

Конечно, в данной реализации отладочной информацией мы лишь отделяем одну процедуру от другой. Далее пользователю приходится работать уже с адресами инструкций. Но даже сейчас мы можем наблюдать некоторые закономерности:

1. Первые и последние инструкции процедуры выполнялись лишь однажды. Из этого можно сделать вывод, что процедура вызывается один раз.
2. По второй и предпоследней строке можно сказать, что внешний цикл выполнялся 9999 раз.
3. Используя те же соображения, утверждаем, что вложенный цикл в общем счете сделал 49995000 итераций.
4. А число 24995736 показывает, сколько раз во время сортировки пришлось делать перестановку элементов.

Для процедуры быстрой сортировки все гораздо сложнее. Вот ее реализация, которая используется в тестовой программе:

```
void qsort(int *A, int n) {
    int i = 0;
    int j = n - 1;
    int x = A[j >> 1];
    do {
         while (A[i] < x) i++;
         while (x < A[j]) j--;
         if (i <= j) {
              swap<int>(A[i], A[j]);
              i++; j--;
         }
    } while (i <= j);
    if (0 < j) qsort(A, j + 1);
    if (i < n - 1) qsort(A + i, n - i);
}
```

Счетчики в данном случае показывают такие цифры:

```
Test!qsort
0x00401170-0x00401178: 4327
0x00401178-0x00401188: 8822
0x00401188-0x0040118d: 36610
0x0040118d-0x00401190: 17478
0x00401190-0x00401196: 53564
0x00401196-0x0040119b: 36610
0x0040119b-0x0040119d: 16641
0x004011a0-0x004011a6: 47523
0x004011a6-0x004011aa: 36610
0x004011aa-0x004011bc: 33913
0x004011bc-0x004011c0: 8822
0x004011c0-0x004011c8: 4326
0x004011c8-0x004011cb: 4326
0x004011cb-0x004011cf: 8822
0x004011cf-0x004011d8: 4495
0x004011d8-0x004011dd: 4327
```

В примере с пузырьковой сортировкой мы оценивали количество вызовов процедуры, используя счетчик в начале процедуры. В этом же случае счетчик показывает число 4327, хотя в действительности процедура вызывалась 8822 раза. Это объясняется тем, что оптимизатор C++ заменяет последний рекурсивный вызов в процедуре `qsort` инструкцией JMP на адрес `0x00401178`. Т.е. счетчик, который показывает действительное количество вызовов процедуры `qsort`, находится по адресу `0x00401178`. Но даже несмотря на это, мы можем предельно точно по показателям счетчиков оценить асимптотическую сложность реализованного алгоритма.

Последнюю реализацию, что мы рассмотрим, будет реализация быстрой сортировки средствами STL (Standard Template Library). Вот так происходит ее вызов:

```
void stl_sort(int *A, int n) {
    typedef std::vector<int> int_vector;
    int_vector a;
    for (int i = 0; i < n; i++) a.push_back(A[i]);
    std::sort(a.begin(), a.end());
    for (int i = 0; i < n; i++) A[i] = a[i];
}
```

Сначала все числа из массива помещаются в вектор, происходит его сортировка, и в конце все числа заносятся обратно в массив.

После профилировки в отчете появляется информация о следующих процедурах:

- `Test!stl_sort`
- `Test!std::vector<int,std::allocator<int> >::insert`
- `Test!std::vector<int,std::allocator<int> >::_Insert_n`
- `Test!std::allocator<int>::allocate`
- `Test!std::_Sort<int *,int>`
- `Test!std::_Unguarded_partition<int *>`

- `Test!std::_Median<int *>`
- `Test!std::_Insertion_sort1<int *,int>`

Не будем подробно рассматривать показатели счетчиков – не зная, что делает каждая процедура, эти цифры нам ничего не скажут. Можно лишь отметить, что асимптотическая сложность реализации близка к асимптотике нашей собственной реализации, о которой говорилось выше.

Ссылку на полный файл отчета профилировки можно найти в приложении 10.7 “Тестовый пример”.

7.1. Корректность результатов

В этом параграфе мы попробуем оценить степень корректности результатов нашей профилировки, насколько они пригодны для анализа реализаций алгоритмов.

Сначала рассмотрим этот вопрос с точки зрения машинных инструкций.

Реализация метода, описанная в 6-ой главе, заменяет все инструкции переходов в исходном коде на инструкции вызова наших процедур (параграф 6.4.3 “Добавление счетчиков”), в которых уже и происходит учет всех прыжков в коде. Если имеет место переход извне, то он может осуществляться исключительно на исходный код (обращение к которому вызывает управляющее исключение) – адрес инструкции из модифицированного кода “посторонний” модуль получить никак не может (см. параграф 5.2.8 “Маскировка внесенных изменений”). Это небольшое резюме опять подтверждает, что, фактически, нет никакой возможности совершить прыжок в анализируемой программе без нашего ведома. Поэтому такая мера сложности алгоритма, как количество исполненных инструкций, нашим методом измеряется абсолютно точно (при условии, что исполняемый модуль соответствует всем накладываемым ограничениям).

А теперь посмотрим, что происходит на практике. В рассмотренном примере с сортировками, результат профилирования пузырьковой сортировки соответствует всем математическим оценкам. Исполнение происходило только в анализируемом модуле. Но вот реализация сортировки средствами STL, как мы выяснили, вызывает функцию из внешней библиотеки (функцию быстрого копирования с использованием MMX). Т.е. при подсчете количества инструкций, которое было выполнено за время работы тестируемого алгоритма, мы уже потеряли точность. И самое плохое, что неизвестно, насколько велика эта погрешность. Поэтому необходимо подвергать анализу все модули, которые используются при работе исследуемого алгоритма.

Углубимся еще больше в архитектурные особенности системы. Самой правдивой (с точки зрения ЭВМ) величиной, которой измеряется время работы некоторого исполняемого

кода, это количество тактов процессора. С такой точки зрения оценка, которую производит наш метод, имеет очень большую погрешность. Объясняется это тем, что количество тактов, затрачиваемых на исполнение инструкции, неодинаково для всех инструкций. Эта величина может быть различной даже для одной и той же инструкции в разные моменты времени. Факторы, которые влияют на эту величину:

- конвейерное исполнение команд на процессоре;
- предсказания переходов (branch prediction);
- кэши первого и второго уровней;
- прочее.

Поэтому для анализа отдельных инструкций и небольших участков кода метод, описанный в этой работе, не пригоден.

Но тут стоит отметить, что, сделав два замера количества тактов для одного участка кода, запущенного дважды, мы вряд ли получим одинаковые значения. Для этого нужно специально создать идеальные условия. Но наш метод будет выдавать одинаковые результаты всегда, что, несомненно, является преимуществом.

Итак, процессор оптимизирует исполнение отдельных инструкций и их небольших последовательностей, что вносит сильные погрешности в измерения нашего метода. Но оптимизировать циклы, вызовы процедур и другие высокоуровневые конструкции он не в силах. Вот здесь проявляются все преимущества внедренных счетчиков: мы можем точно сказать, сколько итераций сделал тот или иной цикл, сколько произошло рекурсивных вызовов процедуры и т.д.

Из этого всего следует простой вывод: метод ценен для исследования и анализа **алгоритма**, а не его реализации.

Заключение

В этой работе был рассмотрен метод, который, используя возможность модификации памяти процесса и возможность динамической генерации исполняемого кода, позволяет добиться того или иного уровня контроля над приложением. В главе 7 “Реализация метода” был представлен способ реализации описанного метода.

В главе 2 “Постановка задачи” был указан ряд ограничений накладываемых на исполняемый модуль, при которых метод будет успешно работать. Но, к сожалению, они далеко не всегда выполнены. Чаще всего нарушаются правила:

a. исполнение инструкций не из стандартного набора x86;
b. чтение / запись данных секции кода.

После нарушения ограничения, пользователь получает сообщение-предупреждение, и исполнение программы завершается.

Можно обозначить следующие направления для дальнейших работ:

- Расширение множества известных инструкций. Здесь можно попробовать подобрать другой дизассемблер длин инструкций. Например, Catchy32 вполне может удовлетворить всем нашим потребностям.
- Для решения проблемы b можно придумать способ исполнения таких инструкций, которые пытаются обратиться к секции кода для чтения или записи. Некоторые из возможных решений:
  - разблокирование страницы памяти секции кода для исполнения инструкции;
  - эмуляция исполнения инструкции.
- Оптимизация структуры данных, хранящей счетчики. Сейчас инкрементация и декрементация счетчиков производится крайне неэффективно.
- Сбор статистики раздельно для потоков.
- Вывод отчетов, более ориентированных на обычного разработчика, который занимается профилированием собственного приложения (или реализованного алгоритма). Поддержка файлов MAP, содержащих отладочную информацию. Графики. Вывод ассемблерных инструкций и соответствующих строк исходного текста программы.

Если мыслить еще шире, то аналогичные приемы можно использовать для более глубокого анализа: поиск утечек памяти, оценка покрытия кода при исполнении, создание автоматических систем для снятия защит с программного обеспечения и прочее.

В качестве заключения хочу сказать, что неанализируемых приложений не существует. Неважно, свое приложение это или стороннее. Эта работа показывает, с какой легкостью мы можем внедриться в приложение, полностью перестроить весь его исполняемый код, и при этом оно даже не будет об этом подозревать. Принципиально невозможно создать для своего приложения стопроцентную защиту от анализа посторонними лицами, можно лишь максимально его затруднить.

Литература

1. “About Valgrind”. http://www.valgrind.org/info/about.html;
2. “AMD CodeAnalyst Performance Analyzer”. http://developer.amd.com/CPU/CODEANALYST/Pages/default.aspx;
3. Dyninst API, “An API for Runtime Code Patching”. http://www.dyninst.org/papers/apiPreprint.pdf;
4. Intel Software Developer’s Manual, 3 том, глава 16, 16.2 “Debug Registers”;
5. Intel Software Developer’s Manual, 3 том, глава 4 “Paging”;
6. Intel Software Developer’s Manual, 3 том, глава 5, 5.5. “Privilege Levels”;
7. Intel Software Developer’s Manual, 3 том, глава 4, 4.1.1 “Three Paging Modes”;
8. John von Neumann, “First Draft of a Report on the EDVAC”. http://qss.stanford.edu/~godfrey/vonNeumann/vnedvac.pdf;
9. Matt Pietrek, “Win32 SEH изнутри”. http://www.wasm.ru/article.php?article=Win32SEHPietrek1;
10. “Microsoft Portable Executable and Common Object File Format Specification”. http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx;
11. “Microsoft Windows 3.0 & MACH 20 Accelerator Card”. http://www.gaby.de/ftp/pub/win3x/archive/softlib/1997w3x.pdf;
12. MSDN, “DllMain Callback Function”. http://msdn.microsoft.com/en-us/library/ms682583(VS.85).aspx;
13. MSDN, “File Mapping”. http://msdn.microsoft.com/en-us/library/aa366556(VS.85).aspx;
14. MSDN, “Inline Assembler”. http://msdn.microsoft.com/en-us/library/4ks26t93(VS.80).aspx;
15. MSDN, “Memory Protection Constants”. http://msdn.microsoft.com/en-us/library/aa366786(VS.85).aspx;
16. MSDN, “Structured Exception Handling”. http://msdn.microsoft.com/en-us/library/ms680657(VS.85).aspx;
17. MSDN, “Vectored Exception Handling”. http://msdn.microsoft.com/en-us/library/ms681420(VS.85).aspx;
18. MSDN, “VirtualProtect” API function. http://msdn.microsoft.com/en-us/library/aa366898(VS.85).aspx;
19. “The VirtualBox architecture”. http://www.virtualbox.org/wiki/VirtualBox_architecture;
20. “Tools Using Dyninst”. http://www.dyninst.org/tools.html;
21. “Valgrind’s Tool Suite”. http://valgrind.org/info/tools.html;

22. “Использование VTune Performance Analyzer на системах с процессором Intel Atom под управлением Windows XP”. http://software.intel.com/ru-ru/articles/using-vtune-atom-windows/;
23. Исходный код дизассемблера длин инструкций. http://www.rsdn.ru/forum/src/3120789.1.aspx;
24. Крис Касперски, “Взлом через покрытие”. http://www.insidepro.com/kk/111/111r.shtml;
25. Крис Касперски, “Вторжение в чужое адресное пространство и защита от него”. http://www.xakep.ru/post/39074/default.asp;
26. Крис Касперски, “Путь воина – внедрение в pe/coff файлы”. http://www.insidepro.com/kk/019/019r.shtml;
27. Марк Руссинович, “Windows Internals Fifth Edition”, глава 9, “Memory Management”;
28. “Механизмы реализации мультизадачности в Intel x86”. http://x86.migera.ru/text/glava_5.html;
29. Москалев Игорь, “SoftIce – руководство”. http://www.hackzone.ru/articles/softice.html;
30. “Объект Device\PhysicalMemory”. http://technet.microsoft.com/ru-ru/library/cc787565(WS.10).aspx;
31. “Перехват API функций в Windows NT (часть 1). Основы перехвата”. http://www.wasm.ru/print.php?article=apihook_1;

Приложения

Приложение 1. Утилита №1. Разделяемая секция PE файла

Данный пример демонстрирует способ ручной настройки Characteristics-атрибутов секции PE-файла. Эти атрибуты предназначены для управления правами доступа, назначаемые системным загрузчиком по умолчанию ко всем страницам секции.

В данном конкретном примере демонстрируется выставление атрибута IMAGE_SCN_MEM_SHARED некоторой секции данных, после чего эта секция становится разделяемой для всех запущенных процессов. В такой разделяемой секции выделяется память под переменную, и после каждого запуска приложения происходит ее инкремент и выведение текущего значения на экран.

Листинг файла Main.cpp

```
#pragma data_seg(".shared")
__declspec(allocate(".shared")) volatile int counter;
#pragma data_seg()

#pragma comment(linker, "/SECTION:.shared,RWS")

#include <iostream>
#include <windows.h>

void main() {
     counter++;
     while (1) {
          std::cout << "Counter: " << counter << std::endl;
          Sleep(500);
     }
}
```

Исходный код и файлы проекта можно найти по адресу:
http://willzyx-edu-project.googlecode.com/svn/trunk/Proj1/samples/01-SharedPESection/

Приложение 2. Утилита №2. Атрибуты прав доступа страниц памяти

Данная программа демонстрирует особенности действия флагов доступности страниц памяти [15] при 32-х битном режиме адресации.

Утилита показывает доступные операции с памятью (читать, писать, исполнять) для различных атрибутов. Для каждой пары “операция – атрибуты“ выполняется тест:

- выделяется страница памяти;
- записывается туда инструкция RET (нужна для случая execute);
- изменяются права доступа к странице в соответствии с тестом;
- выполняется операция;
- если операция не разрешена, то будет инициализировано исключение Access Violation;
- исключение обрабатывается векторным обработчиком.

Листинг файла Main.cpp

```
#include <stdlib.h>
#include <windows.h>
#include <iostream>

#define PAGE_SIZE (4 * 1024)

void *err_handler = 0;

bool except_flag;

enum action_type{ act_read, act_write, act_execute};

LONG NTAPI VectoredExceptionHandler(PEXCEPTION_POINTERS e) {
     except_flag = true;
     e -> ContextRecord -> Eip = e -> ContextRecord -> Esi;
     return EXCEPTION_CONTINUE_EXECUTION;
}
```

```
bool Test(DWORD page_access, action_type action) {
     void *p = VirtualAlloc(0, PAGE_SIZE, MEM_COMMIT,
PAGE_READWRITE);
     *((unsigned char*)p) = 0xC3;
     VirtualProtect(p, PAGE_SIZE, page_access, &page_access);
     except_flag = false;
     switch (action) {
          case act_read: {
               __asm {
                    PUSH ESI;
                    MOV ESI, lbl1;

                    MOV EAX, p;
                    MOV EAX, [EAX];

                    lbl1:
                    POP ESI;
               }
               break;
          }
          case act_write: {
               __asm {
                    PUSH ESI;
                    MOV ESI, lbl2;

                    MOV EAX, p;
                    MOV [EAX], EAX;

                    lbl2:
                    POP ESI;
               }
               break;
          }
          case act_execute: {
               __asm {
                    PUSH ESI;
                    MOV ESI, lbl3;

                    MOV EAX, p;
                    CALL EAX;
                    SUB ESP, 4;

                    lbl3:
                    ADD ESP, 4;
                    POP ESI;
               }
               break;
          }
     }
     VirtualFree(p, PAGE_SIZE, MEM_RELEASE);
     return !except_flag;
}
```

```
void Test(char *text, DWORD page_access) {
     std::cout << "Flag:    " << text << std::endl;
     std::cout << "Allowed: ";
     if (Test(page_access, act_read))
          std::cout << "read ";

     if (Test(page_access, act_write))
          std::cout << "write ";

     if (Test(page_access, act_execute))
          std::cout << "execute ";

    std::cout << std::endl << std::endl;
}

void main() {
     err_handler = AddVectoredExceptionHandler(1,
VectoredExceptionHandler);

     Test("PAGE_NOACCESS", PAGE_NOACCESS);
     Test("PAGE_READONLY", PAGE_READONLY);
     Test("PAGE_READWRITE", PAGE_READWRITE);
     Test("PAGE_EXECUTE", PAGE_EXECUTE);
     Test("PAGE_EXECUTE_READ", PAGE_EXECUTE_READ);
     Test("PAGE_EXECUTE_READWRITE", PAGE_EXECUTE_READWRITE);

     RemoveVectoredExceptionHandler(err_handler);
     std::cin.get();
}
```

Исходный код и файлы проекта можно найти по адресу:
http://willzyx-edu-project.googlecode.com/svn/trunk/Proj1/samples/02-PageAccessFlags/

Приложение 3. Утилита №3. Редактирование секции кода PE файла

Демонстрационная программа, которая показывает, что возможно одновременное выставление флага секции кода и флага прав на редактирование секции для PE файла.

Здесь функция `main` при линковке помещается в секцию с атрибутом кода и правами на запись. Во время исполнения происходит модификация этой функции: добавляется инструкция вызова другой функции `foo` и при исполнении происходит ее вызов.

Листинг файла Main.cpp

```
#include <iostream>
#include <windows.h>

void *err_handler = 0;

void halt_program() {
     std::cout << "Press ENTER to exit…";
     std::cin.get();
     TerminateProcess(GetCurrentProcess(), 0);
}

LONG NTAPI VectoredExceptionHandler(PEXCEPTION_POINTERS e) {
     if (err_handler) {
          RemoveVectoredExceptionHandler(err_handler);
          err_handler = 0;
     }
     std::cout << "Exception " << std::hex <<
          e->ExceptionRecord->ExceptionCode << " catched" <<
          std::endl;
     halt_program();
     return ExceptionContinueSearch;
}

void foo() {
     std::cout << "foo procedure called" << std::endl;
     halt_program();
}
```

```
#pragma code_seg(".modif")

void main() {
     err_handler = AddVectoredExceptionHandler(1,
          VectoredExceptionHandler);

     __asm {
          MOV EAX, proclbl;
          MOV EBX, foo - 5;
          SUB EBX, EAX;
          MOV BYTE PTR [EAX], 0xE8;
          MOV DWORD PTR [EAX + 1], EBX;
     }

     std::cout << "Code section was successful edited" <<
          std::endl;

     __asm {
          proclbl:
          NOP;
          NOP;
          NOP;
          NOP;
          NOP;
     }
}

#pragma code_seg()
#pragma comment(linker, "/SECTION:.modif,RW")
```

Исходный код и файлы проекта можно найти по адресу:
http://willzyx-edu-project.googlecode.com/svn/trunk/Proj1/samples/03-CodeSectionRewrite/

Приложение 4. Дизассемблер длин инструкций

В проекте использовался сторонний дизассемблер длин инструкций. Более подробно он был описан в параграфе 6.2 “Дизассемблер длин инструкций”.

Листинг файла oplength.h

```
#ifndef OPLENGTH_
#define OPLENGTH_

#include <windows.h>

bool GetInstructionSize(PBYTE pOpCode, PDWORD pdwInstructionSize);

#endif
```

Адрес в репозитории:
http://willzyx-edu-project.googlecode.com/svn/trunk/Proj1/src/profiler/oplength.h

Файл oplength.cpp
Содержит реализацию функции `GetInstructionSize`.
Адрес в репозитории:
http://willzyx-edu-project.googlecode.com/svn/trunk/Proj1/src/profiler/oplength.cpp

Приложение 5. Процедура анализа

Ниже приведена процедура пошагового анализа инструкций в памяти, находящихся в виде машинного кода. На вход функции подается адрес, с которого следует начать анализ, и ссылка на список, куда необходимо помещать все адреса, на которые происходят ответвления. Более подробно процесс описан в параграфе 6.4.1 “Анализ исходного кода”.

`code` – объект, который занимается построением модифицированного кода.

Листинг метода `scan` класса `pe_analyzer` (файл pe_analyzer.cpp)

```
bool pe_analyzer::scan(PVOID addr, addr_list &jmp_list) {
  while (true) {
    DWORD oplen;
    // readable copy of memory at [addr]
    PUCHAR op_code = VA() + (DWORD)addr;
    if (!GetInstructionSize(op_code, &oplen)) {
      set_error("Неизвестная инструкция", addr);
      return false;
    }

    bool fl = true;
    switch (op_code[0]) {
      case 0xC3: case 0xC2:
        // RET
        code.add_ret(op_code, oplen, addr);
        return true;
      case 0xE8:
        // relative fixed CALL
        code.add_call(op_code, oplen, addr);
        jmp_list.push_back((PCHAR)addr + oplen + *(PINT)(op_code + 1));
        return true;
      case 0xFF: {
        switch (op_code[1] & 0xF0) {
          case 0xD0:
            // call like "CALL EAX"
          case 0x10:
            // call like "CALL [EAX]" or "CALL [EAX + 4 * EBX]"
          case 0x50:
            // call like "CALL [EAX + 4 * EBX + 1]"
            code.add_call(op_code, oplen, addr);
            return true;
          case 0xE0:
            // jump like "JMP EAX"
          case 0x20:
            // jump like "JMP [EAX]" or "JMP [EAX + 4 * EBX]"
          case 0x60:
            // jump like "JMP [EAX + 4 * EBX + 1]"
            code.add_jmp(op_code, oplen, addr);
            return true;
        }
        break;
      }
      case 0xEB:
        // short JMP
        jmp_list.push_front((PCHAR)addr + oplen + *(PCHAR)(op_code + 1));
        code.add_jmp(op_code, oplen, addr);
        fl = false;
        break;
```

```
      case 0xE9:
        // long JMP
        jmp_list.push_back((PCHAR)addr + oplen + *(PINT)(op_code + 1));
        code.add_jmp(op_code, oplen, addr);
        return true;
      case 0xE0: case 0xE1: case 0xE2:
        // LOOP / LOOPxx
        jmp_list.push_front((PCHAR)addr + oplen + *(PCHAR)(op_code + 1));
        code.add_loop(op_code, oplen, addr);
        fl = false;
        break;
      case 0x70: case 0x71: case 0x72: case 0x73: case 0x74: case 0x75:
      case 0x76: case 0x77: case 0x78: case 0x79: case 0x7A: case 0x7B:
      case 0x7C: case 0x7D: case 0x7E: case 0x7F:
        // short condition jump
        jmp_list.push_front((PCHAR)addr + oplen + *(PCHAR)(op_code + 1));
        code.add_comp_jmp(op_code, oplen, addr);
        fl = false;
        break;
      case 0x0F:
        if ((op_code[1] >= 0x80) && (op_code[1] < 0x90)) {
          // long condition jump
          jmp_list.push_front((PCHAR)addr + oplen + *(PINT)(op_code + 2));
          code.add_comp_jmp(op_code, oplen, addr);
          fl = false;
        }
        break;
    }
    if (fl) {
      code.add_instr(op_code, oplen, addr);
    }
    addr = (PCHAR)addr + oplen;
  }
}
```

Адрес в репозитории:
http://willzyx-edu-project.googlecode.com/svn/trunk/Proj1/src/profiler/pe_analyzer.cpp

Приложение 6. Внедряемые процедуры

Ниже приведен листинг файла **x_func.h** и частично файла **x_func.cpp**. Они содержат несколько процедур, которые будут вызываться непосредственно из генерируемого кода.

Листинг файла x_func.h

```
#ifndef X_FUNC__
#define X_FUNC__

#include "types.h"

// PUSH jmp_addr;
// PUSH next_instr_addr
void xJMP_r();

// PUSH next_instr_addr;
// PUSH jmp_addr;
// PUSH jmp_addr_fake;
void xJMP_f();

// PUSH next_instr_addr;
// PUSH call_addr;
void xCALL_r();

// PUSH next_instr_addr;
// PUSH call_addr;
// PUSH call_addr_fake;
void xCALL_f();

// PUSH ret_addr;  // already there
// PUSH next_instr_addr
#define xRET xJMP_r

// PUSH ret_addr; // already there
// PUSH next_instr_addr
// PUSH n
void xRETn();

#endif
```

Адрес в репозитории:
http://willzyx-edu-project.googlecode.com/svn/trunk/Proj1/src/profiler/x_func.h

- `xJMP_r` – процедура перехода по адресу в исходном коде. В качестве параметров ей передаются два адреса из исходного кода: адрес, по которому должен произойти переход (`jmp_addr`), и адрес байта, следующий сразу за инструкцией перехода (`next_instr_addr` – нужен исключительно для декремента

счетчиков). При необходимости вызывается процедура анализа по адресу перехода (чтобы избежать генерации исключения).

- `xJMP_f` – процедура перехода по адресу в сгенерированном коде. Смысл параметров тот же, что и у `xJMP_r`. Добавлен параметр `jmp_addr_fake` – соответствующий адрес в сгенерированном коде, на который должен произойти переход. В отличие от `xJMP_r` не инициализирует процесс анализа исходного кода.
- `xCALL_r` – схожа с процедурой `xJMP_r`, но предназначена для CALL-подобных вызовов.
- `xCALL_f` – схожа с процедурой `xJMP_f`.
- `xRET` – по сути является процедурой `xJMP_r`. Предназначена для замены инструкции RET.
- `xRETn` – процедура, предназначенная для замены инструкции RET (трехбайтовый вариант). Добавляется параметр `n` – количество байт, удаляемых из вершины стека.

Ниже приведена реализация двух процедур: `xJMP_r` и `xCALL_f`.

Листинг части файла x_func.cpp

```
#define NAKED __declspec(naked)

NAKED void xCALL_f() {
     __asm {
          PUSH EBP;
          MOV EBP, ESP;
          // call_addr_fake = [EBP + 0x4]
          // call_addr = [EBP + 0x8]
          // next_instr_addr = [EBP + 0xC]
            PUSHF;
            PUSHAD;

            PUSH [EBP + 0xC];
            CALL dec_image_counter;
            PUSH [EBP + 0x8];
            CALL inc_image_counter;

            POPAD;
            POPF;
          POP EBP;
          RET 0x4;
     }
}
```

```
NAKED void xJMP_r() {
     __asm {
          PUSH EBP;
          MOV EBP, ESP;
          // next_instr_addr = [EBP + 0x4],
          // jmp_addr = [EBP + 0x8]
            PUSHF;
            PUSHAD;

            PUSH [EBP + 0x4];
            CALL dec_image_counter;

            PUSH [EBP + 0x8];
            PUSH ESP;
            CALL try_analyze_with_counter_inc;
            TEST EAX, EAX;
            JZ image_not_found;

            POP EAX;
            MOV [EBP + 0x4], EAX;
            JMP end_process;

            image_not_found:
            POP EAX;
            MOV EAX, [EBP + 0x8];
            MOV [EBP + 0x4], EAX;

            end_process:
            POPAD;
            POPF;
          POP EBP;
          RET 0x4;
     }
}
```

Адрес в репозитории:
http://willzyx-edu-project.googlecode.com/svn/trunk/Proj1/src/profiler/x_func.cpp

Общего в этих двух процедурах можно выделить то, что при вызове каждой из них создается новый стековый фрейм и в него сохраняются значения регистров и флагов, чтобы после некоторых действий иметь возможность восстановить их. Также стековый фрейм позволяет адресовать передаваемые процедуре параметры.

Рассмотрим процедуру `xJMP_r`.

1. Сначала делаем декремент счетчика по адресу, который хранится в `[EBP + 0x4]` (т.е. `next_instr_addr`).
2. Выделяем память в стеке для параметра, который будет передаваться по ссылке, и помещаем туда адрес, хранящийся в `[EBP + 0x8]` (`jmp_addr`). Сама ссылка передается командой `PUSH ESP`.

3. Запускаем процедуру анализа `try_analyze_with_counter_inc`. Если процедура возвращает ненулевое значение, то анализ успешно завершен, и адрес, переданный по ссылке, заменен на соответствующий адрес в сгенерированном коде, на который должен произойти переход. Также в эту процедуру встроена автоматическая инкрементация соответствующего счетчика.
4. В зависимости от результата анализа записываем нужное значение по адресу `[EBP + 0x4]`. Таким образом, получается, что финальная инструкция `RET 0x4` произведет переход по нужному адресу.

С процедурой `xCALL_f` все гораздо проще: ей не нужно производить никакой анализ, т.к. он уже был произведен и адрес перехода нам явно передают. Стоит лишь отметить, что в этом случае финальная инструкция `RET 0x4` удаляет не все параметры из стека: последний (`next_instr_addr`) там остается. Он и является адресом возврата, который должен был быть помещен в стек, если никакого внедрения бы не проводилось.

Приложение 7. Тестовый пример

Ниже приведена демонстрационная программа, для которой в главе 7 “Результаты применения” приводятся результаты работы профайлера.

Листинг файла Main.cpp тестового примера

```
#pragma auto_inline(off)
#include <iostream>
#include <vector>
#include <algorithm>
#include <stdlib.h>

extern "C" int __stdcall subscribe_me();
extern "C" int __stdcall unsubscribe_me();
extern "C" int __stdcall flush_counters(const char *file_name);

template <class T>
inline void swap(T &a, T &b) {
     T d = a;
     a = b;
     b = d;
}

void fill_random(int *A, int n) {
     while (n--)
          *(A++) = rand();
}

bool check_sort(int *A, int n) {
     while (--n) {
          if (A[0] > A[1]) return false;
          A++;
     }
     return true;
}

void qsort(int *A, int n) {
     int i = 0;
     int j = n - 1;
     int x = A[j >> 1];
     do {
          while (A[i] < x) i++;
          while (x < A[j]) j--;
          if (i <= j) {
               swap<int>(A[i], A[j]);
               i++; j--;
          }
     } while (i <= j);
     if (0 < j) qsort(A, j + 1);
     if (i < n - 1) qsort(A + i, n - i);
}
```

```
void bubble_sort(int *A, int n) {
     while (--n) {
          for (int i = 0; i < n; i++)
               if (A[i] > A[i + 1])
                    swap<int>(A[i], A[i + 1]);
     }
}

void stl_sort(int *A, int n) {
     typedef std::vector<int> int_vector;
     int_vector a;
     for (int i = 0; i < n; i++) a.push_back(A[i]);
     std::sort(a.begin(), a.end());
     for (int i = 0; i < n; i++) A[i] = a[i];
}

const int n = 10000;
int A[n];

void main() {
     fill_random(A, n);
     subscribe_me();
     qsort(A, n);
     unsubscribe_me();
     std::cout << "qsort. "
          << (check_sort(A, n) ? "success" : "fail")
          << std::endl;

     fill_random(A, n);
     subscribe_me();
     bubble_sort(A, n);
     unsubscribe_me();
     std::cout << "bubble_sort. "
          << (check_sort(A, n) ? "success" : "fail")
          << std::endl;

     fill_random(A, n);
     subscribe_me();
     stl_sort(A, n);
     unsubscribe_me();
     std::cout << "stl_sort. "
          << (check_sort(A, n) ? "success" : "fail")
          << std::endl;

     flush_counters("result.log");
     std::cout << std::endl << "Press ENTER to exit...";
     std::cin.get();
}
```

Адрес тестового примера в репозитории:
http://willzyx-edu-project.googlecode.com/svn/trunk/Proj1/src/Test/
Адрес файла результатов тестирования в репозитории:
http://willzyx-edu-project.googlecode.com/svn/trunk/Proj1/src/Test/result.log