**Лекция 6 Статический и динамический анализ программ. Статический анализ кода. Динамический анализ программ. Совместное использование статического и динамического анализа программ.**

**Этапы и средства исследования ПО.** Исследователю часто приходится иметь дело с исполняемыми кодами ПО и не очень подробной пользовательской документацией. В связи с этим основной особенностью исследования ПО является практическая трудность получения исходных текстов программ. Следовательно, одна из важнейших задач тестирования (проверки) программ может быть сформулирована следующим образом: установить, содержит ли данная программа функцию разрушения (нанесения ущерба). По сути данная задача сводится к задаче исследования программы, задаваемой ее объектным или исполняемым кодом.

Задачу изучения программы в общем случае можно сформулировать следующим образом. Мы имеем бинарный код программы (например, ЕХЕ-файл) и минимальную информацию о том, что эта программа делает. Нам нужно получить более детальную информацию о функционировании этой программы. Другими словами, мы знаем, *что* делает программа, и хотим узнать, *как* она это делает.

Конечно, во многих случаях аналитику доступна более детальная информация об анализируемой программе (техническая документация, исходный текст и т.д.), что существенно упрощает процесс анализа. Но мы будем рассматривать наиболее общий (и наиболее сложный) случай, когда ничего, кроме кода программы, аналитику неизвестно.

Актуальность задачи анализа программных реализаций алгоритмов защиты обусловливается следующими факторами.

1. *Компьютеризация всех областей национальной экономики Казахстана.* В настоящее время компьютерные технологии активно применяются практически во всех новых областях нашей жизни. Автоматизированные средства обработки информации используются как в высших эшелонах государственной власти и управления, так и в многочисленных предприятиях и организациях, банках и т.п. Практически каждая организациям имеет у себя конфиденциальную информацию, которую надо защищать.

2. *Многообразие программных средств обработки информации и используемых в них средств защиты.* Средства защиты встраиваются во многие программные продукты самого различного назначения (операционные системы, системы управления базы данных, системы электронного документооборота, разнообразные утилиты и т.п.). При этом подробные технические описания, а также данные независимых экспертиз качества используемой системы защиты далеко не всегда доступны пользователю. В этих условия пользователю информационной системы жизненно важно знать, насколько надежна применяемая им защита.

3. *Большой разброс в уровне подготовленности разработчиков.* Разработкой систем защиты информации занимаются различные люди и организации, порой разительно отличающиеся по своему научно-техническому потенциалу, уровню подготовки, техническим возможностям.

4. *Широкое распространение программных средств зарубежного производства.* Среди всех программных средств, применяемых в Казахстане в настоящее время, львиную долю составляют программные средства зарубежного производства. В ряде стран, например в США и Франции, имеются ограничения на экспорт стойких систем защиты информации.

5. *Регулярное появление новых версий программных продуктов, которые могут отличаться и средствами защиты*

Работа по анализу программы включает в себя три основных этапа.

1. *Подготовительный этап.* На данном этапе аналитик проводит первичное знакомство с анализируемой программой, изучает доступную документацию, планирует дальнейшие исследования, подбирает коллектив и организует его работу. Важность этого этапа нельзя недооценивать.

2. *Восстановление алгоритмов функционирования программы.* На данном этапе, собственно, и производится изучение программы.

3. *Проверка полученных результатов.* Обычно эта проверка заключается в написании тестовой программы, которая реализует восстановленные алгоритмы анализируемой программы. Если поведение тестовой программы не отличается от поведения анализируемой в отношении анализируемых алгоритмов, задачу можно считать решенной. Если же поведение тестовой программы отличается от поведения анализируемой, это означает, что в анализе программы допущены ошибки, которые необходимо устранить. Как правило, правильно восстановить анализируемые алгоритмы с первого раза не удается.

Исследование программы, задаваемой ее исполняемым кодом, включает два этапа:

1. Выделение алгоритма программы (или какой-либо его части, интересующей исследователя) и представление его на языке, удобном для последующего анализа (обычно это язык высокого уровня).

2. Семантический анализ полученного алгоритма для ответа на интересующие вопросы, например о правильности программы, степени ее надежности или наличии в ней непротоколированных (недекларированных) функций.

Два наиболее известных типа программ, предназначенных для исследования ПО: это отладчик (динамическое средство) и дизассемблер (статическое средство). *Отладчик* широко применяется пользователем для отладки собственных программ, при этом задачи построения алгоритма для отладчика вторичны и реализуются самим пользователем. *Дизассемблер* предназначен исключительно для решения задач построения алгоритма и формирует на выходе ассемблерный текст алгоритма.

Наряду с отладчиком и дизассемблером используются и другие инструменты исследования:

• *декомпиляторы* — программы, генерирующие из исполняемого кода программу на языке высокого уровня;

• *трассировщики* — программы, сначала запоминающие каждую инструкцию, проходящую через процессор, а затем переводящие набор инструкций в форму, удобную для статического исследования, автоматически выделяя циклы, подпрограммы и т.п.;

• *следящие системы* — программы, запоминающие и анализирующие трассу уже не инструкции, а других характеристик, например, вызванных программой прерываний.

В настоящее время сформировались следующие подходы к восстановлению алгоритмов, реализуемых программой:

* метод экспериментов;
* статический метод;
* динамический метод.

В методе экспериментов программа рассматривается как «черный ящик», осуществляющий определенные преобразования в зависимости от поступающего на него входа. Аналитик проводит многократные эксперименты, манипулируя входными данными, анализируя и сравнивая получаемые результаты. На основе этих экспериментов он восстанавливает, а точнее, угадывает алгоритмы преобразований.

В статическом методе по файлам программного обеспечения восстанавливаются основные элементы исходной программы, которая затем анализируется с целью получения описания собственно алгоритмов защиты. Основным инструментом статического метода служат программы дизассемблирования, восстанавливающие по исполняемым файлам листинги программы на языке ассемблер. Одним из наиболее удачных дизассемблеров в настоящее время считается пакет IDA.

В динамическом методе анализируемая программа запускается под контролем других специализированных программных средств. Наличие этих средств позволяет прогонять программу в пошаговом режиме, останавливать ее работу при осуществлении тех или иных событий, что значительно облегчает поиск и анализ фрагментов программы, реализующие преобразования, связанные с защитой информации. Основным инструментом динамического метода являются так называемые программы-отладчики.

Перечисленные методы имеют свои достоинства и недостатки и во многом дополняют друг друга

**2. Метод экспериментов с «черным ящиком»**

Термин «черный ящик» взят из математической теории автоматов, где ставится задача, наблюдая вход и выход автомата, построить автомат, эквивалентный данному, т. е. такой, что, подавая одинаковый вход на исходный автомат и построенный нами, мы получим одинаковый выход.

Различают два варианта задачи:

1) вход на автомат является случайным, и его можно только наблюдать;

2) исследователь может по своему усмотрению задавать вход автомата и наблюдать выход (метод прозванивания).

Метод экспериментов с «черным ящиком» сводится к проведению по определенной методике многократных экспериментов (например, по зашифрованию различных текстов на различных ключах с помощью анализируемой системы криптографической защиты) и сравнительному анализу получаемых результатов (например, шифртекстов). Для сравнительного анализа результатов полезно разработать специальные программные средства. К таким средствам относится, например, программа, которая осуществляет операцию покоординатного суммирования байтов двух файлов по модулю 2 и занесение результатов в третий файл. Назовем эту программу XOR. Другой программой такого рода является программа SUB\_256, осуществляющая вычитание по модулю 256 байт второго файла из байтов первого с занесением результатов в третий файл. Эти и другие подобные программы в определенной мере автоматизируют процедуры сравнительного анализа.

Эффективность метода экспериментов слабо зависит от программной реализации системы защиты и определяется в первую очередь сложностью анализируемых алгоритмов. Метод экспериментов эффективен при анализе программ, реализующих относительно простые алгоритмы.

При реализации метода нередко встречаются серьезные трудности:

* нет гарантий, что угаданный результат соответствует действительности. Трудно проверить, что во всех мыслимых ситуациях алгоритм защиты работает именно так, как было восстановлено в ходе анализа;
* как правило, метод приводит к успеху лишь в том случае, когда анализируемый алгоритм защиты достаточно прост. Для сложных алгоритмов обычно удается установить только самые общие факты;
* в современных программных продуктах информация нередко записывается в сложном формате, который к тому же не документирован. Особенно большие проблемы возникают при применении метода экспериментов к программным средствам, файлы которых содержат «шум» от ранее удаленных данных, «мусор» от выделенной, но не инициализированной памяти и т.п. Так, например, восстановить с помощью метода экспериментов форматы почтовых баз Microsoft Outlook и Lotus Notes практически невозможно;
* часто входные данные программы содержат избыточную инфомрацию, используемую для проверки их корректности. Это затрудняет подготовку входных данных для нескольких последовательных экспериментов.

Отмеченные недостатки в ряде случаев создают трудности при применении метода экспериментов с «черным ящиком». Часть из них удается преодолеть, если применять метод в комбинации с динамическим методом анализа.

Метод экспериментов редко применяется в чистом виде. Чаще этот метод применяется как дополнение к динамическому или статическому методу.

т.е. немного раньше той позиции, в которой различаются открытые тексты.

В настоящее время программные средства, легко восстанавливаемые методом экспериментов с «черным ящиком» без привлечения других методов, встречаются редко.

**3. Статический метод**

Статический метод заключается в восстановлении алгоритма защиты посредством анализа имеющегося в распоряжении программного обеспечения. Исполняемые файлы программы обычно состоят из заголовка, в котором содержится необходимая для работы вспомогательная информация, и последовательности исполняемых команд, записанных в машинных кодах команд процессора. В файлах программного обеспечения заключены все данные, нужные для восстановления алгоритма. Задача состоит лишь в том, чтобы найти соответствующие участки программы и перевести их на язык, понятный аналитику. В настоящее время имеется целый ряд программ анализа, которые облегчают перевод.

Основу для такого перевода составляют программы дизассемблирования — *дизассемблеры,* которые переводят последовательность машинных кодов в листинг, близкий к исходному тексту программы на языке ассемблера.

Дальнейшая работа после дизассемблирования сводится к анализу полученных листингов, поиску участков, отвечающих за защиту информации, и переводу описания процедур функционирования алгоритмов защиты на понятный аналитику язык.

Статический метод при отсутствии в программе специальных средств защиты от дизассемблирования позволяет полностью восстановить алгоритмы защиты. Он дает возможность понять структуру программы, способ вызова и взаимодействия отдельных модулей.

Для применения статического метода достаточно иметь в своем распоряжении программное обеспечение. Не требуется, чтобы анализируемая программа, имеющаяся у аналитика, была работоспособна. Это очень удобно в случаях, когда предметом исследований является программный комплекс, требующий дорогостоящих технических средств, которые отсутствуют у аналитика.

При практической реализации алгоритмов дизассемблирования возникают следующие проблемы.

1. *Проблема восстановления символических имен.* В программе, написанной на языке ассемблера, все переменные, метки, процедуры и сегменты имеют символические имена. При компиляции программы эти имена заменяются физическими адресами. В скомпилированном машинном коде не остается информации о символических именах, если, конечно, она специально не помещена туда для отладки или для взаимодействия с другими программами (например, динамически подгружаемые библиотеки Windows содержат таблицу имен импортируемых функций, необходимую для их импорта программами). Обычно эта проблема решается путем «придумывания» дизассемблером символических имен типа var1, label2, ргос3 и т.д.

2. *Проблема различения команд и данных.* В скомпилированной программе машинный код и данные отличаются друг от друга только по контексту использования. Например, машинная команда может непосредственно следовать за глобальной переменной. Если дизассемблер принимает код за данные или наоборот, текст, выдаваемый дизассемблером, становится бессмысленным.

3. *Проблема определения границы машинной команды.* Команды машинного кода, предназначенного для выполнения на CISC- процессорах (процессоры семейств х86 и х86-64 относятся к CISC- процессорам), следуют друг за другом подряд непосредственно, без каких-либо разделителей. При выполнении кода процессор начинает считывать очередную команду с байта, непосредственно следующего за только что выполненной командой. Если дизассемблер неправильно определяет границу команды, он неправильно восстанавливает эту команду и несколько команд, следующих за ней.

Программы дизассемблирования в своей работе могут допускать серьезные ошибки.

Дизассемблеры условно можно разделить на «глупые» (dumb) и «умные» (smart). «Глупые» дизассемблеры даже не пытаются решить вышеописанные проблемы, в результате чего результат работы дизассемблера напоминает ассемблеровский текст весьма отдаленно. В то же время «глупые» дизассемблеры преобразуют машинный код очень быстро (почти мгновенно). К «глупым» дизассемблерам относятся встроенные дизассемблеры отладчиков и антивирусных утилит, предназначенные для интерактивного просмотра небольших участков машинного кода.

«Умные» дизассемблеры преобразуют машинный код в ассемблеровский настолько точно, что в отдельных случаях повторное ассемблирование приводит к тому же самому машинному коду. «Умные» дизассемблеры различают команды и данные, практически всегда правильно определяют границы машинных команд, выделяют в машинном коде отдельные функции, отслеживают перекрестные ссылки. Однако за такие возможности приходится платить. Время дизассемблирования программы «умным» дизассемблером может составлять десятки минут.

Любой дизассемблер не в состоянии восстановить машинный код самораскрывающейся или самошифрующейся программы. В этом случае дизассемблер либо выдает бессмысленную последовательность команд, либо интерпретирует большую часть программы как данные. Существует ряд приемов программирования, при применении которых генерируется машинный код, некорректно воспринимаемый дизассемблером.

В настоящее время статический метод используется чаще всего как вспомогательный инструмент для проверки предположений о восстанавливаемых алгоритмах защиты. Большинство аналитиков отдают предпочтение динамическому методу.

**4. Динамический метод**

**4.1. Программные отладочные средства**

Динамический метод восстановления алгоритмов защиты основан на использовании программных отладочных средств (отладчиков). *Отладчик* — это программа, которая загружает в память другую программу и предоставляет пользователю возможность наблюдать за ходом выполнения этой программы.

Отладчики обычно реализуют следующие возможности:

* пошаговый проход программы (трассировка);
* пошаговый проход программы, при котором вызов функции обрабатывается как одна команда;
* пошаговый проход программы с «провалом» в прерывания;
* установка/снятие точки останова с заданной команды, многие отладчики допускают установку *условных* точек останова, срабатывающих лишь в случае выполнения некоторого дополнительного условия (например, еах = 0). Условные точки останова реализуются как программное расширение обычных (программных или аппаратных) точек останова — точка останова срабатывает вне зависимости от выполнения заданного условия, а затем отладчик принимает решение, следует ли приостановить отлаживаемую программу или продолжить ее выполнение;
* установка/удаление аппаратной точки останова;
* установка/удаление точки останова с порта/портов;
* выполнение участка программы до ближайшей точки останова;
* выполнение участка программы до команды, подсвеченной курсором, или до ближайшей точки останова;
* перезагрузка программы;
* просмотр кода/памяти/регистров/флагов/стека;
* поиск строки в памяти;
* поиск машинной команды в памяти;
* изменение содержимого регистров/флагов/памяти/кода.

Перечислены лишь те функции отладчика, которые могут использоваться для анализа машинного кода, для которого неизвестен исходный текст.

В момент остановки отладчики позволяют просматривать содержимое оперативной памяти как в шестнадцатеричном виде, так и в виде дизассемблированного кода. Встроенные дизассемблеры отладчиков почти всегда относятся к числу «глупых» (dumb), однако это практически не затрудняет работу с отладчиком, поскольку данные легко отличаются от команд по контексту обращений к ним.

Следует иметь в виду, что неаккуратное обращение с отладчиком может привести к аварийному завершению работы операционной системы, а в отдельных (очень редких) случаях и к разрушению файловой системы дисков компьютера.

Возможности отладчиков дают простой и понятный в идейном плане подход к восстановлению алгоритмов защиты, получивший название «динамический метод». Для реализации данного подхода необходимо запустить программу отладчика, открыть исследуемую программу, затем установить несколько точек останова и, двигаясь по программе, по тем или иным признакам (вид экрана, открытие файлов и т.п.) выделить и проанализировать интересующие нас фрагменты программы. При скрупулезной и последовательной реализации данного пути не остается места для эвристических предположений, тех или иных допущений.

По сравнению со статическим методом анализа динамический метод позволяет проводить поиск алгоритмов защиты более целенаправленно. Аналитик может запускать и просматривать именно те участки программного обеспечения, в которых, как ему известно, используются интересующие его алгоритмы защиты. Динамический метод позволяет в моменты останова программы наблюдать состояние процессора и оперативной памяти, что облегчает понимание реализуемых программой преобразований.

Можно комбинировать динамический метод с методом экспериментов с «черным ящиком», произвольно меняя заполнение тех или иных регистров или областей памяти и наблюдая, к каким результатам это приводит. При этом имеется возможность «прозванивать» не только всю программу в целом, но и отдельные ее блоки и узлы, вносить изменения во входные данные уже после того, как программа провела их проверку на наличие искажений. Это заметно повышает эффективность экспериментов.

Также удобно комбинировать динамический метод со статическим. В этом случае аналитик заранее подготавливает листинг дизассемблера, а затем, в ходе анализа программы под отладчиком, регулярно заглядывает в этот листинг. Листинг, сгенерированный «умным» дизассемблером, намного информативнее, чем набор ассемблеровских команд, который можно наблюдать в соответствующем окне отладчика. Использование внешнего листинга часто позволяет быстро понять суть действий, выполняемых теми или иными блоками и узлами анализируемой программы, даже не начиная их трассировки.

Перечислим некоторые факторы, ограничивающие возможности отладчика.

1. Отладчик не может установить точку останова (кроме аппаратных) в ПЗУ, поскольку данный участок памяти физически не может быть изменен. Если программа на низком уровне работает с системными буферами базовой системы ввода-вывода (BIOS), расположенной в ПЗУ, может возникнуть необходимость подробного анализа отдельных функций BIOS. Поскольку при этом доступны лишь пошаговый проход и аппаратные точки останова, подобный анализ является весьма нетривиальной задачей. В более общем случае, отладчик не может устанавливать программные точки останова в областях памяти, недоступных отладчику для записи. Однако, поскольку отладчики обычно выполняются с административными полномочиями, это ограничение редко является существенным.

2. Отладчик может зависнуть, если хотя бы одна точка останова установлена внутри самого отладчика или функции операционной системы, используемой отладчиком. В этом случае происходит зацикливание: точка останова передает управление в функцию отладчика, анализирующую изменения в регистрах и памяти, в процессе анализа отладчик снова встречает ту же самую точку останова, снова передает управление в ту же функцию анализа и т.д. Другими словами, отладчик не может отлаживать сам себя.

3. Зацикливание также может произойти, если аппаратная точка останова установлена так, что срабатывает при изменении некоторого участка памяти в сегменте стека. Это связано с тем, что исследуемая программа, отладчик и операционная система могут использовать общий сегмент стека. Поэтому аппаратная точка останова может сработать внутри отладчика или операционной системы, что, как в предыдущем случае, приводит к зацикливанию.

4. Если программа имеет оверлейную структуру, т.е. в каждый момент времени в памяти находятся только те функции и данные, которые необходимы программе в данный момент, возможна «потеря точки останова», когда программа записывает поверх функции, содержащей точку останова, другую функцию.

5. Если программа работает в многозадачной операционной системе с принудительным переключением задач, возможны проблемы, связанные с тем, что в системе одновременно выполняются несколько экземпляров отлаживаемого кода.

6. Существует множество приемов программирования, которые позволяют так модифицировать программу, что она не может быть исследована под отладчиком. Говорят, что такая программа защищена от отладчика. Более подробно эти приемы будут рассмотрены далее.

**4.2. Методика изучения программ динамическим методом**

Анализ программы динамическим методом разбивается на три основных этапа:

1) поиск подходов к интересующим функциям программы (найти путь, по которому идти);

2) поиск интересующих функций программы (пройти этот самый путь);

3) анализ интересующих функций программы.

При анализе машинного кода задачей первого этапа является выявление в необозримом массиве машинного кода программы некоторых «зацепок», позволяющих достаточно быстро подобраться к интересующим функциям. Существуют два метода решения этой задачи: метод маяков и метод Step-Trace первого этапа.

**Метод маяков**

Метод маяков основан на установке точек останова на так называемые *маяки —* точки программы, в которых программа выполняет действия, легко понимаемые без знания контекста, в котором эти действия выполняются. В роли маяков обычно выступают системные вызовы.

При применении метода маяков аналитик должен выбрать в изучаемой программе интересующие его маяки (например, если анализируется программа шифрования файлов, в качестве маяков целесообразно выбрать обращения программы к файловой системе) и установить на них точки останова.

Это можно сделать двумя способами:

1) найти в программе все маяки и установить на каждый точку останова;

2) установить точку останова в обработчик соответствующего системного вызова.

Второй способ намного проще (требуется устанавливать меньшее количество точек останова), но может привести к «зависанию» отладчика. Если отладчик использует данный системный вызов для своих нужд, то внутри отладчика возможно зацикливание. Кроме того, поведение многих системных вызовов операционных систем (например, системного вызова Windows NtQuerySystemInformation) существенно зависит от параметров вызова. Если поставить точку останова в обработчик такого вызова, она будет срабатывать при каждом обращении к данной системной функции, что во многих случаях неудобно.

Первый способ менее удобен, но более надежен — при его использовании сбои отладчика исключены. Его следует использовать в тех случаях, когда второй способ неприменим.

После того как на интересующие маяки установлены точки останова, программа запускается под отладчиком. При каждом проходе очередного маяка выполнение программы приостанавливается, и аналитик, просматривая содержимое регистров и памяти, получает информацию о ходе выполнения программы (порядок выполнения системных вызовов, параметры этих вызовов, расположение в памяти передаваемых операционной системе или получаемых от операционной системы буферов). Объем и ценность получаемой информации определяются выбором маяков.

**Метод Step-Trace первого этапа**

Метод Step-Trace получил свое название по двум режимам трассировки программы отладчиком: с входом во вложенные функции программы (Trace) и без входа (Step). Суть метода заключается в пошаговом проходе изучаемой программы с попеременным использованием обоих режимов.

Метод может быть использован для поиска в анализируемой программе функции *х,* удовлетворяющей следующим условиям:

а) функция *х* реализует интересующие аналитика алгоритмы;

б) вызов функции *х* легко обнаруживается аналитиком по внешним проявлениям программы.

Этим условиям удовлетворяет, например, функция ввода пароля с клавиатуры, или функция, выводящая на экран компьютера результат проверки пароля.

При применении метода Step-Trace сначала в режиме Step покомандно проходится главная функция программы. При проходе через вызов одной из вложенных функций, назовем ее *f1,* мы убеждаемся, что внутри этого вызова вызывается интересующая нас функция, Например программа запрашивает у пользователя пароль.

На втором этапе программа перезагружается, главная функция проходится в режиме Step до команды call *f1.* Затем с помощью команды Trace производится переход внутрь этой функции и тело функции *f1* пошагово проходится в режиме Step до тех пор, пока не встретится вызов функции, скажем, *f2,* внутри которого вызывается функция *x.*

После того как обнаружена функция *f2,* программа снова перезагружается, главная функция проходится в режиме Step до вызова *f1,* в режиме Trace управление передается внутрь *f1,* далее *f1* проходится и режиме Step до вызова *f2,* управление передается внутрь *f2,* внутри *f2* ищется вызов функции, вызываемой из *f2,* в котором выполняются интересующие нас действия. Когда такая функция найдена, программа опять перезагружается и т.д.

В процессе выполнения этих операций полезно записывать на бумаге последовательность вызовов функций, ведущих к функции *х.*

При использовании данного метода не следует использовать точки останова (или, как минимум, использовать их с осторожностью), поскольку это может привести к тому, что будет анализироваться последовательность вложенных вызовов функций, заведомо не ведущая к функции *х.*

Метод Step-Trace очень эффективен, если в качестве функции *х* выбрана функция, принимающая решение разрешить или запретить пользователю выполнение некоторого действия.

По окончании первого этапа анализа программы динамическим методом мы знаем либо расположение в коде программы одной из интересующих нас функций, либо расположение в памяти буферов, содержащих интересующие данные. Задачей второго этапа анализа программы является выявление всех функций анализируемой программы, реализующих интересующие нас алгоритмы.

Второй этап заключается в анализе потоков данных внутри программы и выяснении путей преобразования этих данных. На втором этапе анализа программы эффективно работают два метода: метод аппаратной точки останова и метод Step-Trace второго этапа.

**Метод аппаратной точки останова**

Данный метод целесообразно использовать в качестве продолжения метода маяков. По окончании первого этапа нам уже известно расположение в памяти буферов с интересующими данными в определенные моменты выполнения программы. Требуется выявить в программе функции, заполняющие или анализирующие эти буфера.

Суть метода заключается в установке на интересующие аналитика данные аппаратной точки останова, срабатывающей при чтении данных из буфера или записи данных в буфер в зависимости от того, что происходит с буфером в процессе работы анализируемой программы.

Как только данные изменены или прочитаны процессором, отладчик останавливается на команде, следующей за той, которая изменила или использовала интересующие данные. Остается выяснить характер работы программы с интересующими данными и, если это неинтересно, продолжить отслеживание преобразований этих данных.

Данный метод весьма эффективен, если анализируемые данные являются глобальной переменной анализируемой программы. Если эти данные лежат в стеке, метод, как правило, не работает из-за рекурсивного зацикливания. Если исследуемые данные лежат в динамически распределяемой памяти (в «куче»), эффективность метода может варьироваться в широких пределах в зависимости от того, насколько интенсивно происходит выделение и освобождение памяти в анализируемой области, т. е. от случайных факторов.

**Метод Step-Trace второго этапа**

Метод Step-Trace может применяться и на втором этапе анализа программы. В этом случае искомая функция *х* должна удовлетворять всего одному условию — функция *х* изменяет интересующие нас данные.

При использовании данного метода на втором этапе анализа программы трассировка начинается не с начала программы, а с момента либо прохода маяка, либо завершения трассировки на первом этапе. Тот факт, что функция *х* была вызвана внутри некоторого вызова функции, устанавливается не по внешнему виду экрана компьютера, а по состоянию интересующих нас буферов памяти. Если в момент прохода вызова некоторой функции в режиме Step содержимое буферов изменилось, это означает, что функция *х* вызывается внутри данного вызова.

В остальном применение метода Step-Trace на втором этапе ничем не отличается от его применения на первом этапе.

К началу третьего этапа анализа машинного кода все интересующие функции программы уже выявлены. Осталось только выяснить, в чем, собственно, заключаются алгоритмы, реализуемые этими функциями. Обычно для этого достаточно внимательно просмотреть дизассемблированные листинги интересующих функций. Если листинги выглядят слишком сложными для понимания, можно попробовать вручную или автоматизированно перевести машинный код функции на один из языков программирования высокого уровня.

После того как выдвинута гипотеза о сути интересующих алгоритмов, эту гипотезу нужно обязательно проверить. Для этого можно, например, запрограммировать интересующий нас алгоритм на языке высокого уровня, вставить в тестовую программу, запустить и посмотреть, совпадает ли ее поведение в части анализируемых алгоритмов с поведением изучаемой программы. Как правило, на этапе проверки полученных результатов выявляются ошибки в понимании анализируемого алгоритма. В среднем для полного восстановления алгоритма работы программы требуется три-четыре итерации анализа/проверки.

**5. Особенности анализа некоторых видов программ**

**5.1. Особенности анализа оверлейных программ**

Обычно весь код и все данные запущенной программы постоянно находятся в оперативной памяти. Однако существуют так называемые *оверлейные программы,* которые размещают в оперативной памяти только те фрагменты кода, которые выполняются в данный момент (это, собственно, и есть определение оверлейной программы). При переполнении оперативной памяти ненужные более функции программы удаляются из памяти, и занимаемая ими память освобождается. При необходимости эти функции могут быть повторно считаны с диска.

При анализе оверлейной программы возникает проблема, выражающаяся в потере точек останова, установленных внутри программы. Когда программа удаляет из оперативной памяти участок кода, в котором установлена точка останова, из памяти удаляется и точка останова (байт СС). Когда данный участок кода повторно считывается с диска в оперативную память, вместо байта СС в память считывается байт, который был на этом месте до установки точки останова, и точка останова не восстанавливается. Таким образом, если изучаемая программа является оверлейной, нельзя исключать, что установленная точка останова будет потеряна до того момента, когда на нее будет передано управление.

При использовании метода Step-Trace для изучения оверлейных программ также возникают трудности. Дело в том, что для оверлейных программ довольно частой является ситуация, когда после вызова функции командой call управление никогда не возвращается на команду, следующую за командой call. Если в процессе выполнения вызванной функции произошла подкачка (считывание) кода с диска и расположение кода в оперативной памяти изменилось, возврат из функции будет осуществлен не туда, где была команда call в момент вызова функции, а туда, где эта команда находится в момент возврата. Если такая ситуация возникла в процессе анализа программы с использованием метода Step-Trace, происходит «уход» программы из-под отладчика.

Преодолеть описанные трудности не так сложно, как кажется на первый взгляд. Любая оверлейная программа содержит *диспетчер оверлеев* — код, отвечающий за подкачку фрагментов программы с диска и за удаление ненужных в данный момент фрагментов программы из оперативной памяти. Диспетчер оверлеев никогда не удаляется из оперативной памяти. Если поставить внутри диспетчера оверлеев точку останова, которая срабатывает при каждом изменении расположения кода программы в памяти, для большинства отладчиков все перечисленные проблемы исчезают.

Как правило, отладчики при каждом срабатывании точки останова проверяют наличие в памяти других точек останова. Если какая-то точка останова потеряна, отладчик ее восстанавливает. При наличии и диспетчере оверлеев описанной точки останова отладчик при каждом ее срабатывании восстанавливает все остальные точки останова.

При наличии в диспетчере описанной точки останова исчезает и проблема, связанная с «уходом» трассируемой программы из-под отладчика. Когда расположение кода в памяти изменяется, управление передается диспетчеру оверлеев, срабатывает точка останова и «уход» программы из-под отладчика не происходит.

**5.2. Особенности анализа графических программ Windows**

Консольные программы имеют всего одну точку входа. С момента запуска программы и до момента ее завершения управление никогда не передается другим программам (за исключением системных вызовов и параллельных потоков). Программа возвращает управление операционной системе только после завершения.

Графическая программа Windows возвращает управление в Windows сразу же по окончании инициализации и создания главного окна. В дальнейшем программа получает управление лишь тогда, когда в одно из ее окон приходит сообщение. При этом управление передается в ту точку входа программы, которая связана с данным окном, — в оконную или диалоговую функцию.

С помощью классической схемы метода Step-Trace, описанной выше, можно трассировать только главную точку входа в программу (функцию WinMain) и функции, непосредственно вызываемые из нее. Для того чтобы применить этот метод к оконным и диалоговым функциям, а также функциям программы, вызываемым из оконных или диалоговых функций, метод Step-Trace необходимо модифицировать.

При использовании модифицированного метода Step-Trace для анализа оконных функций и функций, вызываемых из них, следует вначале выбрать точку входа в программу, из которой имеется прямая передача управления в интересующую нас функцию *х.* Другими словами, нужно получить адрес оконной функции окна, в котором пользователь дает команду выполнить действие, ведущее к вызову функции *х.* Проще всего сделать это, воспользовавшись утилитой Spy++, поставляемой в составе пакета Microsoft Visual Studio. Данная утилита позволяют, указав мышью окно, получить практически полную информацию об этом окне, в том числе и адрес его оконной функции. Использование утилиты Spy++ иллюстрирует рис. 2. В данном случае утилита Spy++ применена для получения адреса оконной функции главного окна калькулятора Windows. Этот адрес указан в строке Window Ргос и равен 01006118.

После того как адрес оконной функции получен, нужно поставить в нее точку останова. Не следует ставить точку останова в самое начало функции — в этом случае программа будет останавливаться на ней при приходе в окно любых сообщений, в том числе и сообщений, совершенно не интересных с точки зрения анализа программы (например, сообщения о том, что окно необходимо перерисовать). Обычно оконная функция начинает свою работу с анализа идентификатора пришедшего сообщения (это второй параметр оконной функции, он располагается в стеке по адресу [ebp + 0Сh]). В пода- вляющем большинстве случаев аналитика интересует лишь реакция оконной функции на сообщение WM\_COMMAND (111h), приходящего в окно, когда пользователь выбирает пункт меню, нажимает специальную комбинацию клавиш, например [**Ctrl-Ins**], или выполняет какое-то действие с одним из контрольных элементов окна (например, нажимает кнопку ОК). Для того чтобы точка останова срабатывала только при приходе сообщения WM\_COMMAND, нужно просмотреть код оконной функции, найти там команды типа

mov еах, [ebp + ОС]

cmp еах,lllh

jne. . .

и поставить точку останова после них. Многие отладчики позволяют ставить на оконные функции точки останова, срабатывающие лишь при приходе определенных сообщений, но, к сожалению, эти точки останова часто работают некорректно, поэтому данной возможностью лучше не пользоваться.

После того как в нужную оконную функцию установлена точка останова, можно начинать трассировку программы. В качестве исходной точки для трассировки берется эта точка останова. В осталь- ном применение метода Step-Trace в этом случае ничем не отличается от его применения в консольных приложениях.

Все вышеописанное относится только к обычным (не диалоговым) окнам. Попытки применить описанную методику к трассировке функций третьего класса (вызываемых из диалоговых функций) к успеху не приводят. Дело в том, что для всех диалоговых окон, имеющихся в системе, в роли оконной функции выступает одна и та же системная функция DefDlgProc, которая при необходимости вызывает диалоговую функцию, ассоциированную с данным диалоговым окном. Код функции DefDlgProc имеет довольно сложный вид, и обнаружить передачу управления из DefDlgProc в диалоговую функцию затруднительно. Вышеописанная методика позволяет установить точку останова в DefDlgProc, но с точки зрения изучения программы это почти ничего не дает аналитику. Поэтому при трассировке функций программы, вызываемых из диалоговых окон, целесообразно использовать иные подходы.

Один из возможных подходов заключается в следующем. Вначале устанавливается точка останова на системную функцию Windows, с помощью которой создается интересующее нас диалоговое окно. Всего в Windows имеется около десяти системных функций (точное количество зависит от версии операционной системы), с помощью которых можно создать диалоговое окно, и в общем случае надо ста- вить точку останова на каждую из них. Впрочем, обычно по внешнему виду диалогового окна можно быстро определить, с помощью какой функции оно создавалось.

Предположим, что интересующее нас диалоговое окно создано с помощью функции DialogBoxParamW. Мы запускаем анализируемую программу под отладчиком, устанавливаем точку останова на функцию DialogBoxParamW и даем программе команду на создание диалогового окна. Программа останавливается на только что установленной точке останова. Четвертым параметром функции DialogBoxParamW является указатель на диалоговую функцию создаваемого окна, этот адрес размещается в стеке по адресу [ebp + 14h].

После того как адрес диалоговой функции окна стал известен, нам остается лишь поставить точку останова в диалоговую функцию и дальше действовать так же, как в случае, когда функция *х* относится ко второму классу

Особым случаем является анализ диалоговых функций программ, написанных в среде MFC. Для этих программ не только оконная функция диалогового окна, но и диалоговая функция являются стандартными системными функциями, не имеющими никакого отношения к анализируемой программе. Как правило, программа получает данные, введенные пользователем в диалоговое окно, непосредственно из полей объекта C + + , соответствующего диалоговому окну (этот объект принадлежит дочернему классу класса CDialog). Если диалоговое окно является модальным, оно вначале создается как немодальное, а затем для окна вызывается функция CDialog :: DoModal, преобразующая его в модальное диалоговое окно.

Для анализа того, какие данные поступают из диалогового окна в программу, необходимо найти в программе вызовы соответствующих методов класса CDialog, а затем внимательно изучить код, следующий за интересующими аналитика вызовами. Следует иметь в виду, что хотя по возвращении управления из функции CDialog :: DoModal диалоговое окно уже не существует как объект графического интерфейса, но соответствующий объект CDialog все еще существует и именно он обычно используется программой для получения данных из диалогового окна.

Несмотря на кажущуюся сложность описанной процедуры анализ путей получения данных из диалоговых окон для MFC-программ в среднем выполняется даже быстрее, чем для обычных программ Windows.

**5.4. Особенности анализа параллельного кода**

Практически все современные программные интерфейсы допускают создание программ, имеющих несколько параллельно выполняющихся потоков (threads). При этом возможна ситуация, когда один и тот же фрагмент кода программы одновременно выполняется в нескольких параллельных потоках. При установке точек останова в такие программы возникает следующая проблема.

Когда мы ставим точку останова на функцию, используемую более чем одним потоком, программа будет останавливаться на ней независимо от того, какой поток в данный момент выполняется. Если эта точка останова используется как отправная точка трассировки, рано или поздно возникает ситуация, когда два или более параллельных потоков «попадают под трассировку». Пусть, например, мы дали команду Step в одном потоке программы, и в этот момент другой поток остановился на точке останова. Мы имеем два потока, выполнение которых приостановлено отладчиком. Когда мы дадим команду Step в следующий раз, будет продолжено выполнение второго потока. Что же касается первого потока, здесь ситуация зависит от отладчика. В одних отладчиках его выполнение никогда не возобновляется, что приводит к сбоям в работе анализируемой программы, в других при запуске отлаживаемой программы возобновляется выполнение всех приостановленных отладчиком потоков. При этом следующая остановка программы может произойти в любом из потоков программы, «попавших под трассировку». С точки зрения аналитика это выглядит как хаотичное и, на первый взгляд, абсолютно бессмысленное изменение данных в окнах отладчика. Дальнейшая работа в таком режиме становится практически невозможной.

Следует отметить, что некоторые отладчики (например, WinDbg) позволяют устанавливать точку останова на конкретный поток программы. К сожалению, такие точки останова часто работают некорректно.

Для того чтобы решить описанную проблему, при анализе параллельного кода нужно придерживаться следующего правила: в начале трассировки все точки останова должны быть удалены из программы. Только так можно гарантировать, что трассировке будет подвергнут единственный поток программы. Большинство отладчиков допускают временное удаление точки останова (команда Disable). При этом точка останова удаляется из отлаживаемой программы, но информация о ней сохраняется в памяти отладчика, и эта точка останова может быть восстановлена всего двумя-тремя нажатиями клавиш.

**5.5. Особенности анализа кода в режиме ядра Windows**

В подавляющем большинстве случаев анализ программных реализаций проводится для прикладных или системных программ, работающих в режиме пользователя. Однако иногда приходится проводить анализ кода, выполняющегося в режиме ядра. Перечислим две наиболее типичные ситуации:

1) требуется проанализировать программное или программноаппаратное средство, включающее в себя один или несколько драйверов физических или логических устройств;

2) требуется проследить нетривиальный путь передачи информации от процесса-клиента к процессу-серверу. Например, требуется выяснить, какой процесс обслуживает серверный конец LPC-порта, к которому подключился анализируемый процесс-клиент. Другой пример — требуется выяснить, какой процесс отправил заданное оконное сообщение в окно анализируемого процесса.

Для анализа кода, выполняющегося в режиме ядра, используются специальные отладчики, так называемые *системные отладчики.* На сегодняшний день одним из наиболее популярных системных отладчиков является Syser.

Ядром данного отладчика является драйвер, который перехватывает большинство функций ядра Windows и предоставляет пользователю практически полный контроль над операционной системой. Пользователь при этом использует те же самые клавиатуру и мышь, что и остальные программы отлаживаемой системы, это большое достоинство данного отладчика. На те периоды времени, когда управление передается в Syser, работа операционной системы приостанавливается, системный таймер также приостанавливается и, с точки зрения системы, эти периоды как бы «выпадают из ее восприятия».

Работая с Syser, следует иметь в виду следующее:

* всегда, когда это возможно, следует запускать Syser в виртуальной машине. Это заметно ускоряет перезагрузки операционной системы, которые при анализе кода ядра и драйверов обычно требуются очень часто;
* не следует без веских причин запускать драйвер отладчика в режимах Automatic, System и Boot. В этом случае любая ошибка при загрузке отладчика приведет к полной невозможности загрузки операционной системы и потребует восстановления с использованием «безопасного режима» и других подобных средств. Не исключено, что потребуется полная переустановка операционной системы;
* применять Syser для анализа кода, выполняющегося в режиме пользователя, следует лишь в случае крайней необходимости (например, если нужно выяснить, какой процесс-клиент вызвал ту или иную функцию ядра). Точки останова, поставленные в пользовательскую часть того или иного адресного пространства часто работают некорректно. Если необходимо одновременно анализировать и клиентскую программу, и ядро, обычно более удобно запустить в системе два отладчика: Syser для анализа ядра и обычный отладчик для анализа клиентской программы;
* условные (conditional) точки останова в Syser часто работают некорректно. Если условие накладывается на содержимое регистра, вероятность корректного срабатывания точки останова заметно выше, чем если условие накладывается на содержимое области оперативной памяти;
* содержимое некоторых областей системного адресного пространства (например, области памяти, занимаемой программным модулем win32k.sys) зависит от того, в контексте какого процесса выполняется код ядра. Если отладчик явно некорректно показывает содержимое какого-то участка системного адресного пространства, следует явно переключить отладчик на адресное пространство нужного процесса;
* показывая пользователю текущий стек вызовов функций, Syser иногда ошибается, не замечая некоторые кадры стека;
* почти весь код ядра и драйверов является параллельным. Анализируя код, выполняющийся в режиме ядра, следует придерживаться рекомендаций, приведенных в предыдущем подразделе;
* код ядра и драйверов не всегда выполняется в контексте процесса, инициировавшего обрабатываемый запрос. При обработке аппаратных прерываний, асинхронном вводе-выводе и в некоторых других подобных ситуациях выполнение кода в режиме ядра может происходить в контексте любого процесса, выполняющегося в системе в данный момент;
* если необходимо проанализировать под отладчиком основную точку входа драйвера (DriverEntry), практически невозможно установить точку останова в нужный участок кода, используя лишь штатные средства отладчика. Это связано с тем, что анализируемый код начинает выполняться немедленно после загрузки в память, и «поймать» момент времени, когда интересующий аналитика код уже загружен, но еще не выполняется, практически невозможно. Данную проблему можно решить, вставив команду int 3 (байт СС) непосредственно в секцию кода файла анализируемого программного файла. Если в заголовке этого файла установлена ненулевая контрольная сумма, перед стартом драйвера контрольную сумму следует либо обнулить, либо (лучше) перезаписать с учетом изменений, внесенных в машинный код. Команда int 3, встретившаяся в машинном коде, воспринимается отладчиком Syser как точка останова, при ее выполнении управление передается в отладчик. В этот момент аналитик должен с помощью команды eb восстановить байт, поверх которого им был записан байт СС, далее можно выполнять анализ кода как обычно. Не следует без крайней необходимости применять этот прием к драйверам, загружаемым автоматически, — в этом случае малейшая ошибка при работе с отладчиком может привести к невозможности загрузки операционной системы. Если же имеется крайняя необходимость применить данный прием к автоматически загружаемому драйверу, следует предварительно убедиться, что после перезагрузки операционной системы Syser загрузится раньше анализируемого драйвера. Также следует иметь в виду, что этот прием применим лишь в отношении тех драйверов, для которых операционная система не проверяет цифровую подпись файла;
* при анализе в Syser сетевых драйверов не следует забывать, что Syser останавливает таймер только локального компьютера. Клиентские программы, выполняющиеся на других компьютерах сети, могут воспринять приостановку операционной системы сервера отладчиком как фатальную ошибку в ходе выполнения удаленного запроса;
* как правило, Syser не вполне корректно интерпретирует загруженную в него символьную информацию об анализируемых программных модулях. Работая с Syser, полезно держать перед собой открытое окно «умного» дизассемблера, например IDA.

Помимо Syser, для анализа кода, выполняющегося в режиме ядра, можно использовать отладчик WinDbg, входящий в состав программного пакета Debugging Tools for Windows. Достоинства и недостатки этих двух отладчиков примерно компенсируют друг друга, и выбор между Syser и WinDbg — дело личных предпочтений аналитика.

**Инструментальные средства проведения испытания программ.** В состав инструментальных средств проведения испытания программного обеспечения могут входить:

* • анализаторы исходных текстов программ;
* • анализатор объектных кодов программ;
* • программы модельных испытаний;
* • программы оценки показателей качества ПО;
* • программы оценки показателей безопасности ПО;
* • программы экспертизы выполнения требований к ПО;
* • программы интерфейса эксперта;
* • программы принятия решений;

• программы расчета рисков.

.