ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССОНАЛЬНОГО ОБРАЗОВАНИЯ
«ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ – УЧЕБНО-НАУЧНО-ПРОИЗВОДСТВЕННЫЙ КОМПЛЕКС»
УЧЕБНО-НАУЧНО-ИССЛЕДОВАТЕЛЬСКИЙ ИНСТИТУТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

В.Т. Еременко, А.П. Фисун

ПРОГРАММНО-АППАРАТНЫЕ СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

Рекомендовано ФГБОУ ВПО «Госуниверситет-УНПК»
для использования в учебном процессе в качестве учебного пособия
для высшего профессионального образования

Орел 2015

УДК 681.3.067
ББК 32.973
Е70

Рецензенты:
к.т.н., доцент, зав. кафедрой «Системы информационной безопасности»
ФГБОУ ВПО «Брянский государственный технический университет»
Рытов М.Ю.

д.т.н., профессор кафедры «Информационные системы» ФГБОУ ВПО
«Госуниверситет-УНПК» Раков В.И.

Еременко В.Т.
Е70 Программно-аппаратные средства защиты информации: учебное пособие для высшего профессионального образования / В.Т. Еременко, А.П. Фисун. – Орел: ФГБОУ ВПО «Госуниверситет - УНПК», 2015. – 173 с.

Пособие состоит из введения, семи разделов, заключения, списка используемой литературы.

Первый раздел посвящен средствам и методам защиты информации в операционных системах, второй - моделям безопасности баз данных, третий - защите информации от разрушающих программных воздействий, четвертый посвящен защите систем излучения, анализ программных средств рассмотрен в пятом разделе, запутывающие преобразования программ - в шестом, в седьмом - защита информации систем на этапах жизненного цикла.

Учебное пособие может быть эффективно использовано для самостоятельной работы студентов, обучающихся направлению 10.03.01 (090900) «Информационная безопасность» и по специальностям, связанным с информационной безопасностью.

УДК 681.3.067
ББК 32.973

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Защита информации – область науки и техники, охватывающая совокупность криптографических, программно-аппаратных, технических, правовых и организационных методов и средств обеспечения безопасности информации при ее обработке, хранении и передаче с использованием современных информационных технологий.

Основная цель защиты информации в информационных системах – предотвратить незаконное овладение ею или ее порчу. Эффективной может быть только комплексная защита, сочетающая в себе правовые, организационные и инженерно-технические аспекты.

Что касается подходов к реализации защитных мероприятий по обеспечению безопасности информационных систем, то сложилась трехэтапная разработка таких мер.

Первая стадия – выработка требований – включает:

- определение состава средств информационной системы (ИС);
- анализ уязвимых элементов ИС;
- оценка угроз (выявление проблем, которые могут возникнуть из-за наличия уязвимых элементов);
- анализ риска (прогнозирование возможных последствий, которые могу вызвать эти проблемы).

Вторая стадия – определение способов защиты – включает ответы на следующие вопросы:

- какие угрозы должны быть устранены и в какой мере?
- какие ресурсы системы должны быть защищаемы и в какой степени?
- с помощью каких средств должна быть реализована защита?
- какова должна быть полная стоимость реализации защиты и затраты на эксплуатацию с учетом потенциальных угроз?

Третья стадия – определение функций, процедур и средств безопасности, реализуемых в виде некоторых механизмов защиты.

В данном пособии описаны подходы к обеспечению защищенности ИС на второй и третьей стадиях применяемых для защиты ИС мер.

1 Защита информации в операционных системах

1.1 Классификация угроз безопасности операционной системе

Угрозы безопасности операционной системы существенно зависят от условий эксплуатации системы, от того, какая информация хранится и обрабатывается в системе и т. д.

Например, если операционная система (ОС) используется главным образом для организации электронного документооборота, наиболее опасными являются угрозы, связанные с несанкционированным доступом к файлам. Если же операционная система используется как платформа для провайдера Internet-услуг, наиболее опасны атаки на сетевое программное обеспечение (ПО) ОС. Организация эффективной и надежной защиты ОС невозможна без предварительного анализа возможных угроз ее безопасности.

Единой и общепринятой классификации угроз безопасности ОС пока не существует. Однако можно классифицировать эти угрозы по различным аспектам их реализации.

Классификация угроз по цели:

- несанкционированное чтение информации;
- несанкционированное изменение информации;
- несанкционированное уничтожение информации;
- полное или частичное разрушение операционной системы (под разрушением ОС понимается целый комплекс разрушающих воздействий от кратковременного вывода из строя («завешивания») отдельных программных модулей системы до физического стирания с диска системных файлов).

Классификация угроз по принципу воздействия на ОС:

- использование известных (легальных) каналов получения информации, например, угроза несанкционированного чтения файла, доступ пользователей к которому определен некорректно – разрешен доступ пользователю, которому согласно адекватной политике безопасности доступ должен быть запрещен;

- использование скрытых каналов получения информации, например, угроза использования злоумышленником недокументированных возможностей ОС;
- создание новых каналов получения информации с помощью программных закладок.

Классификация угроз по характеру воздействия на ОС:

- активное воздействие: несанкционированные действия злоумышленника в системе;
- пассивное воздействие: несанкционированное наблюдение злоумышленника за процессами, происходящими в системе.

Классификация угроз по типу используемой злоумышленником слабости защиты:

- неадекватная политика безопасности, в том числе и ошибки администратора системы;
- ошибки и недокументированные возможности ПО ОС, в том числе и так называемые люки – случайно или преднамеренно встроенные в систему «служебные входы», позволяющие обходить систему защиты; обычно люки создаются разработчиками ПО для тестирования и отладки, и иногда разработчики забывают их удалить или оставляют специально;
- ранее внедренная программная закладка.

Классификация угроз по способу воздействия на объект атаки:

- непосредственное воздействие;
- превышение пользователем своих полномочий;
- работа от имени другого пользователя;
- использование результатов работы другого пользователя (например, несанкционированный перехват информационных потоков, инициированных другим пользователем).

Классификация угроз по способу действий злоумышленника:

- в интерактивном режиме (вручную);
- в пакетном режиме (с помощью специально написанной программы, которая выполняет негативные воздействия на ОС без непосредственного участия пользователя-нарушителя).

Классификация угроз по объекту атаки:

- ОС в целом;
- объекты ОС (файлы, устройства и т. д.);

– субъекты ОС (пользователи, системные процессы);
– каналы передачи данных.

Классификация угроз по используемым средствам атаки:

– штатные средства ОС без использования дополнительного ПО;
– ПО третьих фирм (к этому классу ПО относятся как компьютерные вирусы и другие вредоносные программы (exploits), которые можно легко найти в Интернет, так и ПО, изначально разработанное для других целей: отладчики, сетевые мониторы, сканеры и т.д.);
– специально разработанное ПО.

Классификация угроз по состоянию атакуемого объекта ОС на момент атаки:

– хранение;
– передача;
– обработка;

Приведенная классификация не претендует ни на строгость, ни на полноту.

Система называется *защищенной*, если она предусматривает средства защиты от основных классов угроз.

Защищенная ОС обязательно должна содержать средства разграничения доступа пользователей к своим ресурсам, а также средства проверки подлинности пользователя, начинающего работу с ОС. Кроме того, защищенная ОС должна содержать средства противодействия случайному или преднамеренному выводу ее из строя.

Если ОС предусматривает защиту не от всех основных классов угроз, а только от некоторых, такую сиситему называют частично защищенной. Например, ОС MS-DOS с установленным антивирусным пакетом является частично защищенной системой – она защищена от компьютерных вирусов.

Политика безопасности – это набор норм, правил и практических приемов, регулирующих порядок хранения и обработки ценной информации. В применении к ОС политика безопасности определяет то, какие пользователи могут работать с ОС, какие пользователи имеют доступ к каким объектам ОС, какие события должны регистрироваться в системных журналах и т. д.

Адекватной политикой безопасности называется такая политика безопасности, которая обеспечивает достаточный уровень

защищенности ОС. Следует отметить, адекватная политика безопасности – это не обязательно та политика безопасности, при которой достигается максимально возможная защищенность системы.

1.2 Основные средства и методы защиты информации в ОС

Под *аппаратным обеспечением средств защиты* ОС понимается совокупность средств и методов, используемых для решения следующих задач:

1) Управление оперативной и виртуальной памятью компьютера.

Основная угроза, защита от которой реализуется с помощью средств управления оперативной памятью, заключается в том, что один процесс, выполняющийся в многозадачной ОС, несанкционированно получает доступ к области памяти другого процесса, выполняющегося параллельно. Данная угроза представляет опасность не только с точки зрения обеспечения надежности ОС, но и с точки зрения обеспечения ее безопасности.

Существует два основных подхода к обеспечению защиты оперативной памяти процесса от несанкционированного доступа со стороны других процессов.

Первый подход заключается в том, что при каждом обращении процессора к оперативной памяти осуществляется поверка корректности доступа. Теоретически этот подход позволяет создать абсолютно надежную защиту от несанкционированного доступа процесса к «чужой» памяти. То есть, если выделить каждому процессу отдельную область памяти и блокировать все обращения за ее пределы, доступ процесса к чужой памяти становится невозможным. Однако при этом станет практически невозможным взаимодействие процессов. В применяемых на практике ОС значительная часть оперативной памяти, выделенной процессу, является разделяемой.

Альтернативный подход к обеспечению защиты оперативной памяти заключается в выделении каждому процессу индивидуального адресного пространства, аппаратно изолированного от других процессов. При этом по какому бы адресу оперативной памяти ни обратился процесс, он не сможет обратиться к памяти, выделенной другому процессу, поскольку одному и тому

же значению адресу в разных адресных пространствах соответствуют разные физические адреса оперативной памяти. На практике центральный процессор использует логическую адресацию оперативной памяти, то есть позволяет обращаться к ячейкам памяти не по физическим адресам, а по неким виртуальным адресам, автоматически преобразуемых в физические незаметно для выполняющегося процесса.

Данный подход надежно защищает от случайных, обусловленных ошибками в программном обеспечении, обращений процессов к чужой оперативной памяти, но не всегда позволяет защититься от подобных обращений, предпринимаемых преднамеренно с целью несанкционированного доступа к чужому адресному пространству. При данном подходе, так же как и при предыдущем, значительная часть оперативной памяти процесса должна быть сделана разделяемой, то есть значительная часть физической оперативной памяти должна проецироваться сразу в несколько различных адресных пространств. Иначе требования ОС к оперативной памяти значительно возрастут, а ее производительность значительно снизится. При этом по-прежнему сохраняется опасность преднамеренного несанкционированного воздействия одного процесса на другой путем несанкционированного обращения к разделяемой памяти.

Кроме того, при любом подходе к обеспечению защиты памяти ОС должна предусматривать средства отладки программ. А отладка программы невозможна без доступа процесса-отладчика к области памяти, принадлежащей отлаживаемому процессу. Несанкционированное использование отладчиков представляет собой серьезную угрозу защищенности ОМ, поэтому обычно выдвигается требование, согласно которому политика безопасности, принятая в защищенной ОС, не должна допускать запуск любых отладчиков.

Описанные подходы к решению рассматриваемой проблемы не являются взаимоисключающими и могут применяться в совокупности.

2) Планирование задач.

Планирование задач в многозадачной операционной системе заключается в распределении ОС времени центрального процессора (или процессоров) между параллельно выполняющимися задачами. В роли задач могут выступать либо процессы, либо потоки или нити –

потоки машинных команд, последовательно выполняющихся на процессоре. В многопоточных ОС один процесс может иметь два или более потоков, что позволяет распараллеливать вычисления не только между процессами, но и в пределах одного процесса. Неэффективное планирование задач негативно сказывается в основном на эффективности и надежности функционирования ОС, но так же может негативно повлиять и на ее защищенность. Например, если пользователь-злоумышленник может приостанавливать выполнение системных процессов, тем самым он может блокировать различные функции ОС, включая функции, связанные с защитой информации.

Планирование задач может быть реализовано без вытеснения прерванных задач (*невытесняющее* планирование задач) или с вытеснением прерванных задач (*вытесняющее* планирование задач). В первом случае выполнение задачи может быть прервано только по инициативе самой задачи, то есть задача, выполнив все необходимые действия, должна самостоятельно прервать свое выполнение. При невытесняющем планировании задача получает управление при получении сообщения от пользователя, внешнего устройства, операционной системы или другой задачи и отдает управление по завершении обработки сообщения. До тех пор, пока задача не завершила обработку сообщения, другие задачи не могут получить управление, и операционная система «зависает». Вывести из строя многозадачную ОС, в которой планирование задач производится без вытеснения, очень легко – для этого необходимо ввести в бесконечный цикл обычную прикладную программу. Обычно ОС с вытесняющим планированием задач содержат специальные средства аварийного завершения задач в экстренных случаях, но эти средства никогда не работают достаточно надежно. Поэтому планирование задач без вытеснения неприемлемо для защищенных ОС.

Основная угроза подсистеме планирования задач (планировщику задач) заключается в том, что злоумышленник сможет приостановить или прекратить выполнение задач, критичных для обеспечения безопасности ОС. Для нейтрализации этой угрозы она должна обладать следующими свойствами:

- поддерживать вытеснение задач;
- позволять создание высокоприоритетных задач только привилегированным пользователям;

- критичные для обеспечения безопасности системы задачи защищать от несанкционированного вмешательства в ход их выполнения (например, от несанкционированного снижения приоритета);
- фатальный сбой в процессе функционирования одной из задач, критичных для обеспечения безопасности, должен вызывать крах ОС.

3) Синхронизация параллельных задач.

В многозадачных ОС с вытеснением задач часто возникает необходимость синхронизации параллельно выполняющихся задач. Пусть, например, две задачи совместно выполняют сложную операцию над некоторыми данными, причем эта операция разбивается на три простые операции, две из которых могут выполняться параллельно, а для реализации третьей необходимо, чтобы первые две операции уже были выполнены. Эту ситуацию иллюстрирует рисунок 1.

Данный метод синхронизации задач очень прост и универсален и может применяться в любой ОС. Однако этот метод имеет существенный недостаток. Дело в том, что задача 1 затрачивает часть процессорного времени на поиск знака, выставленного задачей 2. Если время ожидания между двумя последовательными попытками обнаружения знака выбрано недостаточно большим, то доля процессорного времени, получаемого задачей 1, будет достаточно велика, что приведет к заметному снижению общей производительности ОС. С другой стороны, если время ожидания между попытками обнаружения знака велико, это вызовет неоправданное снижение быстродействия задачи 1. Чтобы избежать этой ситуации, во многих ОС предусматриваются специальные функции, позволяющие задачам ждать появления знаков, выставленных другими задачами, не затрачивая ресурсов ОС.

4) Обеспечение корректности совместного доступа к объектам.

В процессе функционирования многозадачной ОС часто возникает ситуация, когда две или более задач одновременно обращаются к одному и тому же объекту ОС. Если при этом режим доступа хотя бы одной задачи допускает изменение данных

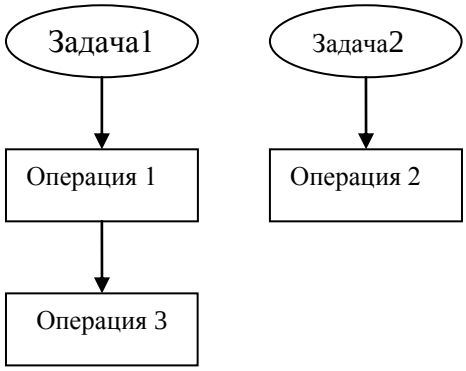

Рисунок 1 – Синхронизация параллельных задач

объекта, не исключено, что обращения других задач к данному объекту будут выполнены некорректно, а если две или более задач одновременно изменяют данные объекта, весьма вероятно, что эти данные окажутся испорчены.

Если несколько задач одновременно читают некоторые данные, это не таит в себе никакой опасности. Если же задача изменяет какие-то данные, доступ к этим данным должен быть запрещен другим задачам. Единственное исключение из этого правила допускается в том случае, когда все задачи, изменяющие один и тот же объект, открывают его в режиме, допускающем совместное изменение данных. Открывая объект в таком режиме, задача тем самым заявляет ОС, что она полностью берет на себя ответственность за все действия, связанные с обеспечением корректности совместного доступа к данным. Если в ОС для некоторого класса объектов поддерживается открытие в режиме совестного изменения, ОС должна предусматривать специальные средства, с помощью которых программы могут обеспечивать корректный совместный доступ к объектам данного типа.

Не всегда некорректный совместный доступ данных происходит при обращениях задач к объектам ОС. Если в ОС допускается выполнение нескольких параллельных потоков в составе одного процесса, некорректный совместный доступ может иметь место при одновременном изменении двумя или более потоками процесса одних и тех же глобальных данных.

Так же как и при синхронизации задач, такой подход приводит к неоправданным расходам процессорного времени. Поэтому в большинстве ОС предусматриваются специальные средства

поддержки корректности совместного доступа нескольких задач к одним и тем же данным.

5) Предотвращение тупиковых ситуаций.

Тупиковая ситуация (другие названия – тупик, клинч, дедлок) может возникнуть, когда несколько программ одновременно пытаются открыть несколько одних и тех же объектов в режиме монопольного доступа. Если одна программа открыла одну часть рассматриваемого множества объектов, а другая программа – другую часть, ни одна из программ не сможет открыть остальные объекты до тех пор, пока другая программа их не закроет. Если функции закрытия объектов в подобной ситуации не предусмотрены ни в одной из программ, ситуация становится тупиковой – каждая из программ ждет, когда другая программа закроет открытые ею объекты.

Рассмотрим пример. ОС используется для ведения банковских операций. В системе имеется два банковских счета, на одном из которых лежит $1000, на другом $2000. В системе имеется два пользователя, каждый из которых имеет полный доступ к обоим счетам. Эти два пользователя одновременно обращаются к данным счетам, причем первый из них хочет перевести $300 с первого счета на второй, а второй – перевести $1500 со второго счета на первый. События могут развиваться следующим образом.

1. Задача, запущенная первым пользователем, открывает первый банковский счет в режиме монопольного доступа.
2. Задача, запущенная вторым пользователем, открывает второй банковский счет в режиме монопольного доступа.
3. Задача, запущенная первым пользователем, пытается открыть второй банковский счет в режиме монопольного доступа, но не может этого сделать, поскольку второй банковский счет уже открыт в режиме монопольного доступа задачей, запущенной вторым пользователем. Задача ждет, когда второй банковский счет будет освобожден.
4. Задача, запущенная вторым пользователем, пытается открыть первый банковский счет в режиме монопольного доступа, но не может этого сделать, поскольку первый банковский счет уже открыт в режиме монопольного доступа задачей, запущенной первым пользователем. Задача ждет, когда первый банковский счет будет освобожден.

В результате выполнение обеих задач приостанавливается на неопределенное время (рисунок 2).

Существует ряд методов борьбы с тупиковыми ситуациями. Самый простой из них заключается в следующем. Если программа для выполнения некоторой операции должна открыть в монопольном режиме несколько объектов ОС, но не смогла этого сделать, поскольку некоторые из этих объектов уже открыты в монопольном режиме другой программой, программа должна закрыть все уже открытые объекты, подождать некоторое время, и повторить всю операцию сначала. Наилучшие результаты достигаются, если время ожидания берется случайным. При использовании этого метода можно гарантировать, что в процессе функционирования данной программы тупиковые ситуации не возникнут.

Как правило, ОС гарантируют невозможность возникновения тупиковых ситуаций при выполнении прикладными программами кода ОС на основе решения задач «обедающих философов», «читателей и писателей» и других задач, решающих проблемы тупиков. Обеспечить предотвращение тупиковых ситуаций при выполнении кода прикладной программы должна сама программа.

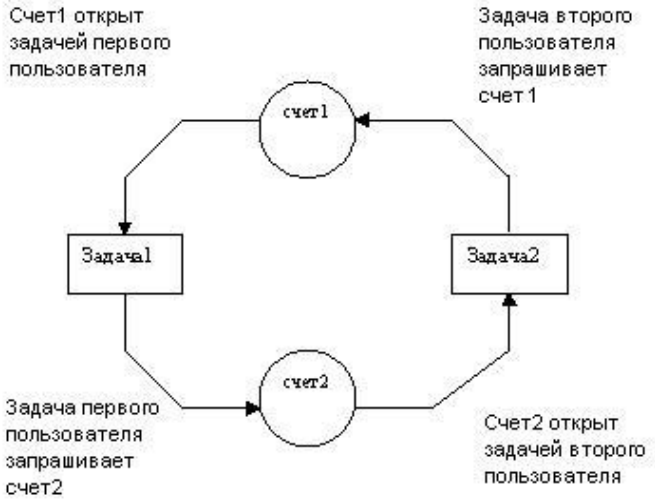

Рисунок 2 – Тупиковая ситуация в банковской системе

1.3 Разграничение доступа к объектам ОС

Объектом доступа называется любой элемент ОС, доступ которого пользователей и других субъектов доступа может быть произвольно ограничен. Ключевым словом в данном определении

является слово «произвольно». Так как возможность доступа к объектам ОС определяется не только ее архитектурой, но и текущей политикой безопасности.

Методом доступа к объекту называется операция, определенная для некоторого объекта.

Субъектом доступа называется любая сущность, способная инициировать выполнение операций над объектами. Иногда к субъектам относят процессы, выполняющиеся в системе. Но, поскольку процессы в ОС выполняются не сами по себе, а от имени и под управлением пользователей, логичнее считать субъектом доступа именно пользователя, от имени которого выполняется процесс.

Под *правом доступа* понимается право на выполнение доступа к объекту по некоторому методу или группе методов. Говорят, что субъект имеет некоторую привилегию, если он имеет право на доступ по некоторому методу или группе методов ко всем объектам ОС, поддерживающим данный метод доступа.

Разграничением доступа субъектов к объектам является совокупность правил, определяющая для каждой тройки субъект-объект-метод, разрешен ли доступ данного субъекта к данному объекту по данному методу. Субъект называется суперпользователем, если он имеет возможность игнорировать правила разграничения доступа к объектам.

Правила разграничения доступа, действующие в ОС, устанавливаются администраторами системы при определении текущей политики безопасности. За соблюдением этих правил субъектами доступа следит *монитор ссылок* – часть подсистемы защиты ОС.

Рассмотрим типичные модели разграничения доступа.

1. Избирательное разграничение доступа.

Система правил избирательного или дискреционного разграничения доступа формулируется следующим образом.

1) Для любого объекта операционной системы существует владелец.
2) Владелец объекта может произвольно ограничивать доступ других субъектов к данному объекту.
3) Для каждой тройки субъект-объект-метод возможность доступа определена однозначно.

4) Существует хотя бы один привилегированный пользователь (администратор), имеющий возможность обратиться к любому объекту по любому методу доступа. Это не означает, что этот пользователь может игнорировать разграничение доступа к объектам и поэтому является суперпользователем. Не всегда для реализации возможности доступа к объекту ОС администратору достаточно просто обратиться к объекту.

Последнее требование введено для реализации механизма удаления потенциально недоступных объектов.

Для определения прав доступа субъектов к объектам при избирательном разграничении доступа используется матрица доступа. Строки этой матрицы представляют собой объекты, столбцы – субъекты. В каждой ячейке матрицы хранится совокупность прав доступа, предоставленных данному субъекту на данный объект.

Поскольку матрица доступа очень велика, она никогда не хранится в системе в явном виде. Для сокращения объема матрицы доступа используется объединение субъектов доступа в группы. Права, предоставленные группе субъектов для доступа к данному объекту, предоставляются каждому субъекту группы.

Вместе с каждым объектом доступа хранятся его *атрибуты защиты*, описывающие, кто является владельцем объекта и каковы права доступа к данному объекту различных субъектов. Атрибуты защиты фактически представляют собой совокупность идентификатора владельца объекта и строку матрицы доступа в кодированном виде.

2. Изолированная программная среда.

Изолированная или замкнутая программная среда представляет собой расширение модели избирательного разграничения доступа. Здесь правила разграничения доступа формулируются следующим образом:

- для любого объекта ОС существует владелец;
- владелец объекта может произвольно ограничивать доступ других субъектов к данному объекту;
- для каждой четверки субъект-объект-метод-процесс возможность доступа определена однозначно;
- существует хотя бы один привилегированный пользователь (администратор), имеющий возможность обратиться к любому объекту по любому методу доступа;

– для каждого субъекта определен список программ, которые этот субъект может запускать.

При использовании изолированной программной среды права субъекта на доступ к объекту определяется не только правами и привилегиями субъекта, но и процессом, с помощью которого субъект обращается к объекту.

Изолированная программная среда существенно повышает защищенность ОС от разрушающих программных воздействий, включая программные закладки и компьютерные вирусы. Кроме того, при использовании данной модели повышается защищенность целостности данных, хранящихся в системе. В тоже время изолированная программная среда создает определенные сложности в администрировании ОС.

Изолированная программная среда не защищает от утечки конфиденциальной информации.

3. Полномочное разграничение доступа без контроля информационных потоков.

Полномочное или мандатное разграничение доступа обычно применяется в совокупности с избирательным. При этом правила разграничения доступа формулируются следующим образом.

– для любого объекта ОС существует владелец;
– владелец объекта может произвольно ограничивать доступ других субъектов к данному объекту;
– для каждой тройки субъект-объект-метод возможность доступа определена однозначно;
– существует хотя бы один привилегированный пользователь (администратор), имеющий возможность удалить любой объект;
– во множестве объектов доступа операционной системы выделяется подмножество *объектов полномочного разграничения доступа*. Каждый объект полномочного разграничения доступа имеет *гриф секретности*. Чем выше числовое значение грифа секретности, тем секретнее объект. Нулевое значение грифа секретности означает, что объект не секретен. Если объект не является объектом полномочного разграничения доступа или если объект не секретен, администратор может обратиться к нему по любому методу, как и в предыдущей модели разграничения доступа;
– каждый субъект доступа имеет *уровень допуска*. Чем выше числовое значение уровня допуска, тем больший допуск имеет

субъект. Нулевое значение уровня допуска означает, что субъект не имеет допуска. Обычно ненулевое значение допуска назначается только субъектам-пользователям и не назначается субъектам, от имени которых выполняются системные процессы;

– если:
- объект является объектом полномочного разграничения доступа,
- гриф секретности объекта строго выше уровня допуска субъекта, обращающегося к нему,
- субъект открывает объект в режиме, допускающем чтение информации,

то доступ субъекта к объекту запрещен независимо от состояния матрицы доступа.

К объектам полномочного разграничения доступа обычно относят только файлы. Часто множество объектов полномочного разграничения доступа лежит во множестве всех файлов, но не совпадает с ними. В идеале к объектам полномочного разграничения доступа следует относить файлы, в которых может храниться секретная информация, и не относить файлы, в которых секретная информация храниться не может (например, файлы программ).

В данной модели разграничения доступа администраторам ОС, как правило, назначается нулевой уровень допуска.

Не трудно заметить, что, за исключением правила 4, данная модель сводится к предыдущей.

Поскольку данная модель не дает ощутимых преимуществ по сравнению с предыдущей и в то же время существенно сложнее ее в технической реализации, на практике данная модель используется крайне редко.

4. Полномочное разграничение доступа с контролем информационных потоков.

Правила разграничения доступа в данной модели формулируются следующим образом.

– для любого объекта операционной системы существует владелец;
– владелец объекта может произвольно ограничивать доступ других субъектов к данному объекту;
– для каждой четверки субъект-объект-метод-процесс возможность доступа определена однозначно в каждый момент

времени. При изменении состояния процесса со временем возможность предоставления доступа также может измениться, то есть если в некоторый момент времени к некоторому объекту разрешен доступ некоторого субъекта посредством некоторого процесса, это не означает, что в другой момент времени доступ тоже будет разрешен. Вместе с тем в каждый момент времени возможность доступа определена однозначно – никаких случайных величин здесь нет. Поскольку права процесса на доступ к объекту меняются с течением времени, они должны проверяться не только при открытии объекта, но и перед выполнением над объектом таких операций, как чтение и запись;

- существует хотя бы один привилегированный пользователь (администратор), имеющий возможность удалить любой объект;
- во множестве объектов выделяется подмножество *объектов полномочного разграничения доступа*. Каждый объект полномочного разграничения доступа имеет *гриф секретности*. Чем выше числовое значение грифа секретности, тем секретнее объект. Нулевое значение грифа секретности означает, что объект не секретен. Если объект не является объектом полномочного разграничения доступа или если объект несекретен, администратор может обратиться к нему по любому методу, как и в предыдущей модели разграничения доступа;
- каждый субъект доступа имеет *уровень допуска*. Чем выше числовое значение уровня допуска, тем больший допуск имеет субъект. Нулевое значение уровня допуска означает, что субъект не имеет допуска. Обычно ненулевое значение допуска назначается только субъектам-пользователям и не назначается субъектам, от имени которых выполняются системные процессы;
- если:
  - объект является объектом полномочного разграничения доступа,
  - гриф секретности объекта строго выше уровня допуска субъекта, обращающегося к нему,
  - субъект открывает объект в режиме, допускающем чтение информации, то доступ субъекта к объекту должен быть запрещен независимо от состояния матрицы доступа. Это – так называемое правило NRU (Not read Up – не читать выше).

- каждый процесс операционной системы имеет *уровень конфиденциальности*, равный максимуму из грифов секретности объектов, открытых процессом на протяжении своего существования. Уровень конфиденциальности фактически представляет собой гриф секретности информации, хранящейся в оперативной памяти процесса;
- если:
  - объект является объектом полномочного разграничения доступа,
  - гриф секретности объекта строго ниже уровня конфиденциальности процесса, обращающегося к нему,
  - субъект собирается записывать в объект информацию, то доступ субъекта к объекту должен быть запрещен независимо от состояния матрицы доступа. Это правило разграничения доступа предотвращает утечку секретной информации. Так называемое *правило NWD* (Not Write Down – не записывать ниже).
- понизить гриф секретности объекта полномочного разграничения доступа может только субъект, который:
  - имеет доступ к объекту согласно правилу 7;
  - обладает специальной привилегией, позволяющей ему понижать грифы секретности объектов.

При использовании данной модели разграничения доступа существенно страдает производительность операционной системы, поскольку права доступа к объекту должны проверяться не только при открытии объекта, но и при каждой операции чтения/записи.

Кроме того, данная модель разграничения доступа создает пользователям определенные неудобства, связанные с тем, что если уровень конфиденциальности процесса строго выше нуля, то вся информация в памяти процесса фактически является секретной и не может быть записана в несекретный объект. Если процесс одновременно работает с двумя объектами, только один из которых является секретным, процесс не может записывать информацию из памяти во второй объект. Эта проблема решается посредством использования специального программного интерфейса (API) для работы с памятью. Области памяти, выделяемые процессам, могут быть описаны как объекты полномочного разграничения доступа, после чего им могут назначаться грифы секретности. При чтении секретного файла процесс должен считать содержимое такого файла

в секретную область памяти, используя для этого функции операционной системы, гарантирующие невозможность утечки информации. Для работы с секретной областью памяти процесс должен использовать специальные функции. Поскольку утечка информации из секретных областей памяти в память процесса невозможна, считывание процессом секретной информации в секретные области памяти не отражается на уровне конфиденциальности процесса. Если же процесс считывает секретную информацию в область памяти, не описанную как объект полномочного разграничения доступа, повышается уровень конфиденциальности процесса.

Из сказанного следует, что пользователи ОС, реализующих данную модель разграничения доступа, вынуждены использовать программное обеспечение, разработанное с учетом этой модели. В противном случае пользователи будут испытывать серьезные трудности в процессе работы с объектами операционной системы, имеющими ненулевой гриф секретности.

Также вызывает определенные проблемы вопрос о назначении грифов секретности создаваемым объектам. Если пользователь создает новый объект с помощью процесса, имеющего ненулевой уровень конфиденциальности, пользователь вынужден присвоить новому объекту гриф секретности не ниже уровня конфиденциальности процесса. Во многих случаях это неудобно.

Каждая из приведенных моделей имеет свои недостатки и достоинства. С помощью таблицы можно провести их сравнительный анализ.

Из таблицы 1 видно, что модель полномочного разграничения доступа без контроля информационных потоков уступает по всем параметрам модели избирательного разграничения доступа. Поэтому применять полномочное разграничение доступа без контроля информационных потоков нецелесообразно ни в каких ситуациях.

Если для организации чрезвычайно важно обеспечение защищенности системы от несанкционированной утечки информации, без полномочного разграничения доступа с контролем информационных потоков просто не обойтись. В остальных ситуациях применение этой модели нецелесообразно из-за резкого ухудшения эксплуатационных качеств ОС. Что касается изолированной программной среды, то ее целесообразно

использовать в случаях, когда очень важно обеспечивать целостность программ и данных ОС. В остальных ситуациях простое избирательное разграничение доступа наиболее эффективно.

Таблица 1 – Модели разграничения доступа

свойства модели	избирательное разграничение доступа	изолированная программная среда	полномочное разграничение доступа	
			без контроля потоков	с контролем потоков
защита от утечки информации	отсутствует	отсутствует	отсутствует	имеется
защищенность от разрушающих воздействий	низкая	высокая	низкая	низкая
сложность реализации	низкая	средняя	средняя	высокая
сложность администрирования	низкая	средняя	низкая	высокая
затраты ресурсов компьютера	низкие	низкие	низкие	высокие
использования программного обеспечения, разработанного для других систем	возможно	возможно	возможно	проблематично

1.4 Стандарты защищенности операционных систем

Специальных стандартов защищенности ОС не существует. Для оценки защищенности операционных систем используются стандарты, разработанные для компьютерных систем вообще. Ниже мы подробно рассмотрим две наиболее часто применяемые в России системы стандартов такого рода. *«Оранжевая книга» (TCSEC[1]).*

Наиболее известным стандартом безопасности компьютерных систем является документ под названием «Критерии безопасности компьютерных систем» (Trusted computer system evaluation criteria), разработанный Министерством обороны США в 1983 году. Этот документ более известен под неформальным названием «Оранжевая книга». Согласно «Оранжевой книге» все защищенные

[1] http://www.boran.com/security/tcsec.html

компьютерные системы делятся на семь классов от D1 (минимальная защита, фактически отсутствие всякой защиты) до A1 (максимальная защита). Основные требования «Оранжевой книги» к операционным системам можно сформулировать следующим образом (очень упрощенно).

Класс D1. Никаких требований. К этому классу относятся все операционные системы, не удовлетворяющие требованиям высших классов.

*Класс C1***.** В операционной системе поддерживается избирательное (дискреционное) разграничение доступа. Пользователь, начинающий работать с системой, должен подтвердить свою подлинность (аутентифицироваться).

Класс C2. Выполняются все требования класса C1. Все субъекты и объекты операционной системы имеют уникальные идентификаторы. Все действия всех субъектов доступа, не разрешенные явно, запрещены События, потенциально опасные для поддержания защищенности операционной системы, регистрируются в специальном журнале (журнале аудита), работать с которым могут только привилегированные пользователи. Вся информация, удаляемая из оперативной памяти компьютер или с внешних носителей информации, удаляется физически и не может быть в дальнейшем доступна ни одному субъекту доступа.

Класс B1. Выполняются все требования класса C2. Поддерживается полномочное (мандатное) разграничение доступа к объектам операционной системы. Поддерживается маркировка экспортируемой информации.

Класс B2. Выполняются все требования класса B1. Подсистема защиты операционной системы реализует формально определенную и четко документированную модель безопасности. Осуществляется контроль скрытых каналов утечки информации. Интерфейс подсистемы защиты четко и формально определен, его архитектура и реализация полностью документированы. Выдвигаются более жесткие требования к идентификации, аутентификации и разграничению доступа.

Рассмотрим ОС, удовлетворяющие требованиям класса C2. Более низкие классы вообще нет необходимости рассматривать. Требованиям класса C2 удовлетворяют многие версии UNIX, Windows NT, OS/400, VAX/VMS и IBM MVS с пакетом RACF. Операционные системы, удовлетворяющие требованиям более

высоких классов защиты, практически недоступны для рядового пользователя. Это объясняется, с одной стороны, большой ресурсоемкостью подсистем защиты, удовлетворяющих требованиям класса B1 и выше, и, с другой стороны, трудностями обеспечения нормального функционирования распространенного программного обеспечения в таких операционных системах. Если требования класса позволяют применять в защищенной операционной системе программное обеспечение, разработанное для других программных сред (например, в Windows NT можно запускать Microsoft Office для Windows 95), то требования более высоких классов защиты настолько жестки, что заметно мешают функционированию прикладных программ, разработанных без учета этих требований. Например, текстовый редактор Microsoft Word, будучи запущен в ОС, удовлетворяющей требованиям класса B1, будет некорректно функционировать при одновременном открытии документов с различным грифом секретности.

В настоящее время существует более 20 операционных систем для компьютеров класса «мэйнфрейм» и «супер ЭВМ», удовлетворяющих требованиям класса B1, и, по крайней мере, одна ОС (Bull Multics), удовлетворяющая требованиям класса B2.

За время, прошедшее со времени разработки требований «Оранжевой книги», многие из них уже устарели. С другой стороны, появился целый ряд новых требований к безопасности компьютерных систем, не отраженных в «Оранжевой книге». Это связано с тем, что за это время был открыт целый ряд ранее неизвестных угроз безопасности компьютерных систем.

К основным недостаткам «Оранжевой книги» относятся следующие:

- совершенно не рассматриваются криптографические средства защиты информации;
- практически не рассматриваются вопросы, связанные с обеспечением защиты системы от атак, направленных на временный вывод системы из строя (атаки типа «отказ в обслуживании»);
- не уделяется должного внимания вопросам защиты от негативных воздействий программных закладок и компьютерных вирусов;

- недостаточно подробно рассматриваются вопросы взаимодействия нескольких экземпляров защищенных систем в локальной или глобальной вычислительной сети;
- требования к средствам защиты от утечки конфиденциальной информации из защищенной системы ориентированы на хранение конфиденциальной информации в базах данных и не пригодны для защиты электронного документооборота.

В 1992 году Гостехкомиссия при Президенте Российской Федерации опубликовала пять руководящих документов, посвященных вопросам защиты компьютерных систем. Рассмотрим важнейшие из них: «Средства вычислительной техники. Защита от несанкционированного доступа к информации. Показатели защищенности от несанкционированного доступа к информации» и «Автоматизированные системы. Защита от несанкционированного доступа к информации. Классификация автоматизированных систем и требования по защите информации».

В первом документе рассматриваются требования к обеспечению защищенности отдельных программно-аппаратных элементов защищенных компьютерных систем. Под средствами вычислительной техники понимаются не только аппаратные средства, но и «совокупность программных и технических элементов систем обработки данных, способных функционировать самостоятельно или в составе других систем». Установлено семь классов защищенности средств вычислительной техники. Самые низкие требования предъявляются к классу 7, самые высокие – к классу 1. Требования этих классов в основном соответствуют аналогичным требованиям «Оранжевой книги», при этом класс 7 соответствует классу D1 «Оранжевой книги», класс 6 – классу C1,…, класс 1 – классу A1. Из наиболее существенных отличий следует отметить следующие:

- начиная с класса 5 вводятся требования, связанные с поддержанием целостности комплекса средств защиты защищаемой системы. Эти требования усиливаются в классах 4 и 3;
- в классе 5 требования к процедурам идентификации и аутентификации пользователей несколько менее сильны, чем в соответствующем ему классе C2 «Оранжевой книги»;
- начиная с класса 4 требования к запрету повторного использования удаленной информации несколько более сильны, чем в «Оранжевой книге».

Второй документ, касающийся автоматизированных систем, вводит стандарты защищенности не отдельных программно-аппаратных средств защиты, а всей защищенной компьютерной системы в целом. Система стандартов этого документа более существенно отличается от аналогичных стандартов Оранжевой книги. Все автоматизированные системы разделяются на три группы, в каждой из которых вводится своя иерархия классов защиты. Всего вводится девять классов защиты, требования которых в применении к операционным системам излагаются ниже (очень упрощенно).

ГРУППА 3. Однопользовательские системы.

Класс 3Б. Проверка подлинности пользователя при входе в систему. Регистрация входа и выхода пользователей из системы. Учет используемых внешних носителей информации.

Класс 3А. Выполняются все требования класса 3Б. Регистрация распечатки документов. Физическая очистка очищаемых областей оперативной и внешней памяти.

ГРУППА 2. Многопользовательские системы, в которых пользователи имеют одинаковые полномочия доступа ко всей информации.

Класс 2Б. Проверка подлинности пользователя при входе в систему. Регистрация входа и выход» пользователей из системы. Учет используемых внешних носителей информации.

Класс 2А. Выполняются все требования класса 2Б. Избирательное разграничение доступа. Регистрация событий, потенциально опасных для поддержания защищенности системы. Физическая очистка очищаемых областей оперативной и внешней памяти. Наличие подсистемы шифрования конфиденциальной информации, использующей сертифицированные алгоритмы.

ГРУППА 1. Многопользовательские системы, в которых пользователи имеют различные полномочия доступа к информации.

Класс 1Д. Проверка подлинности пользователя при входе в систему. Регистрация входа и выхода пользователей из системы. Учет используемых внешних носителей информации.

Класс 1Г. Выполняются все требования класса 1Д. Избирательное разграничение доступа. Регистрация событий, потенциально опасных для поддержания защищенности системы. Физическая очистка очищаемых областей оперативной и внешней памяти.

Класс 1В. Выполняются все требования класса 1Г. Полномочное разграничение доступа. Усилены требования к подсистеме регистрации событий, потенциально опасных для поддержания защищенности системы. Интерактивное оповещение администраторов системы о попытках несанкционированного доступа.

Класс 1Б. Выполняются все требования класса 1В. Наличие подсистемы шифрования конфиденциальной информации, использующей сертифицированные алгоритмы.

Класс 1А. Выполняются все требования класса 1Б. Различные субъекты доступа имеют различные ключи, используемые для шифрования конфиденциальной формации. Нетрудно видеть, что требования; к защищенности систем классов 3Б, 2Б и 1Д (минимальная защита в каждой группе) совпадают друг с другом и представляют собой несколько усиленную версию класса С1 «Оранжевой книги» (и соответствующего ему класса 6 для средств вычислительной техники). Требования класса 1Г в основном совпадают с требованиями класса С2 «Оранжевой книги», а класс 1В из рассматриваемого документа очень похож на класс В1 «Оранжевой книги». Таким образом, отличия требований Гостехкомиссии от требований «Оранжевой книги» в наиболее распространенном диапазоне защищенности операционных систем (С2-В1 по Оранжевой книге и 1Г-1В по документам Гостехкомиссии), незначительны.

Если ОС сертифицирована по некоторому классу защищенности некоторой системы стандартов, это вовсе не означает, что информация, хранящаяся и обрабатывающаяся в этой системе, защищена согласно соответствующему классу. Защищенность операционной системы определяется не только ее архитектурой, но и текущей политикой безопасности. Если, например, все объекты ОС доступны всем пользователям, эта система не защищена вообще, даже если она сертифицирована по В2.

Как правило, сертификация операционной системы по некоторому классу защиты сопровождается составлением требований к адекватной политике безопасности, при неукоснительном выполнении которой защищенность конкретного экземпляра операционной системы будет соответствовать требованиям соответствующего класса защиты.

Для примера рассмотрим некоторые (далеко не все) требования к конфигурации ОС Windows NT, которые должны выполняться для соответствия защищенности операционной системы классу С2 «Оранжевой книги»:

- загрузка компьютера невозможна без ввода пароля;
- на жестких дисках используется только файловая система NTFS;
- запрещено использование паролей короче 6 символов;
- запрещена эмуляция OS/2 и POSIX;
- запрещен анонимный и гостевой доступ;
- запрещен запуск любых отладчиков;
- тумблер питания и кнопка RESET недоступны пользователям;
- запрещено завершение работы операционной системы без входа пользователя в систему;
- политика безопасности в отношении аудита построена таким образом, что при переполнении журнала безопасности операционная система прекращает работу (зависает). После этого восстановление работоспособности системы может быть произведено только администратором;
- запрещено разделение между пользователями ресурсов сменных носителей информации (флоппи-дисков, CD-ROM и т. д.);
- запись в системную директорию и файлы инициализации операционной системы разрешена только администраторам и системным процессам.

Определяя адекватную политику безопасности, администратор ОС должен в первую очередь ориентироваться на защиту операционной системы от конкретных угроз ее безопасности. К сожалению, в настоящее время часто встречается ситуация, когда администратор формирует требования к политике безопасности исходя не из комплекса угроз, защиту от которых нужно реализовать, а из некоторых абстрактных рекомендаций по поддержанию безопасности той или иной операционной системы. Наиболее часто в качестве таких рекомендаций берут требования различных стандартов безопасности компьютерных систем, наиболее «популярными» являются стандарты «Оранжевой книги».

В результате не исключена ситуация, когда ОС, сертифицированная по очень высокому классу защиты, оказывается уязвимой для некоторых угроз даже при соответствии политики безопасности требованиям соответствующего класса защиты. Например, операционная система, защищенная по классу С2

«Оранжевой книги», уязвима для несанкционированных действий администратора.

Рассмотрим далее, каким образом обеспечивается безопасность в наиболее популярных ОС, таких как Windows NT и Linux. Порядок функционирования системы безопасности ОС Microsoft Windows NT (речь идет о версии 4.0 с произвольным Service Pack) определяется большим числом настроек параметров и элементов ее структуры, которые при инсталляции ОС по умолчанию устанавливаются таким образом, что решению задачи защиты информации в ОС отводится второстепенная роль. Данное обстоятельство вынуждает пользователя ОС самостоятельно выбрать тот или иной путь решения задачи защиты информации. Важную роль при этом играет принятая пользователем политика безопасности, а если посмотреть на проблему шире, концепция повышения эффективности защиты ОС Microsoft Windows NT.

Следует отметить, что в большинстве случаев особое внимание уделяется многочисленным настройкам параметров ОС с целью приведения порядка ее функционирования в соответствие неким формальным требованиям, а также проверке корректности работы отдельных защитных механизмов. В то же время вопросам определения ролей администратора и других привилегированных пользователей ОС, правил безопасного администрирования, порядка взаимодействия распределенных компонентов системы защиты уделяется недостаточно внимания. Таким образом, не учитывается тот факт, что даже некоторые штатные действия пользователей могут внести угрозу безопасности всей ОС, не говоря уже об ошибках администрирования или злонамеренных действиях.

1.5 Защита информации в ОС Windows

Целью защиты информации в WINDOWS можно рассмотреть на примере защиты от атак, цель которых – реализация угроз конфиденциальности или целостности информации. Их можно разделить на несколько групп.

1) Атаки, реализуемые через воздействие на подсистему аутентификации с использованием следующих возможностей:

– возможность получения прямого доступа или доступа через загрузку на компьютере иной ОС (например, ОС MS-DOS) к разделам SAM или SECURITY реестра ресурсов с целью последующей

модификации хранящихся в них аутентификационных данных пользователей;

– возможность получения прямого доступа или доступа через загрузку на компьютере иной ОС к разделам SAM или SECURITY реестра ресурсов с целью последующего подбора хранящихся в их аутентификационных данных пользователей;

– возможность модификации системного ПО, с целью подмены процедуры аутентификации;

– возможность перехвата и анализа пакетов сетевого информационного обмена с целью подбора переданных по каналам ЛВС аутентификационных данных пользователей.

2) Атаки, реализуемые путем незаконного захвата привилегий. По используемой бреши в системе защиты их можно разделить на две группы:

– с использованием отсутствия проверки наличия привилегии отлаживать системные процессы в некоторых функциях ОС. Несмотря на то, что в ОС Windows NT 4.0 Service Pack 4 данная проблема решена, как показывает опыт, нет гарантий отсутствия аналогичных брешей.

– с использованием возможности подмены нарушителем системных именованных коммуникационных каналов (pipe) и получения за счет этого привилегий пользователей к ним обратившихся. Захват привилегий происходит при удаленном редактировании реестра ресурсов ОС, журнала аудита, администрировании сетевого принтера и в некоторых других случаях.

3) Атаки, реализуемые путем внедрения в ОС программных закладок или троянских коней. Для внедрения закладок в большинстве случаев необходимы получение прав администратора ОС или загрузка на компьютере отличной от Microsoft Windows NT операционной системы. По уровню внедрения закладки в ОС их можно разделить на две группы:

– закладки, внедряемые на уровне ядра ОС (kernel mode). Эти закладки позволяют динамически модифицировать в памяти компьютера код ядра ОС, осуществлять доступ к объектам (файлам) без учета требований системы разграничения доступа;

– закладки, внедряемые на пользовательском уровне ОС (user mode). Данные закладки позволяют модифицировать процедуру аутентификации пользователя или осуществлять доступ к объектам

(файлам) от имени пользователя с максимальными правами (правами пользователя SYSTEM).

Так как в дальнейшем предполагается рассматривать доменную архитектуру ОС как базовую при построении системы защиты информации и с учетом описанных выше атак, можно выделить следующие наиболее уязвимые элементы и данные системы защиты в домене ОС Microsoft Windows NT:

- аутентификационные данные пользователей рабочих станций, хранящиеся в их реестрах ресурсов;
- аутентификационные данные пользователей домена, сохраняемые в реестрах ресурсов рабочих станций, с которых они осуществляли вход в домен;
- системное ПО рабочих станций;
- аутентификационные данные пользователей рабочих станций, передаваемые по каналам ЛВС;
- некоторые штатные действия администратора домена по непосредственному или удаленному администрированию рабочих станций;
- некоторые обращения друг к другу распределенных компонентов ОС.

Первой в 1996 году концепцию повышения эффективности защиты предложила сама корпорация Microsoft. В качестве основного ориентира ею были выбраны требования класса защиты C2 TCSEC. Однако в дополнение к этим требованиям предлагалось наложить существенное ограничение на порядок конфигурирования ОС, а именно: компьютеры, на которых функционирует ОС Microsoft Windows NT, должны быть изолированными, то есть отключенными от локальных или глобальных вычислительных сетей.

Сертификация ОС Microsoft Windows NT Workstation, Server version 3.5 U.S. Service Park 3 по классу защиты C2 TCSEC была успешно проведена в 1996 году. При этом были представлены необходимые настройки ОС и требования к ее конфигурации. Анализируя указанные настройки, можно сделать вывод, корпорацией Microsoft была использована следующая концепция повышения эффективности защиты ОС Microsoft Windows NT. Обеспечение безопасности ОС Microsoft Windows NT в соответствии с требованиями класса C2 TCSEC возможно:

- без установки дополнительного программного или аппаратного обеспечения,

- с помощью соответствующих настроек параметров и порядка конфигурирования ОС,только в случае, если компьютер, на котором функционирует ОС, является изолированным.

При реализации этой концепции основной проблемой является обеспечение защиты компьютера от загрузки иной ОС, отличной от Microsoft Windows NT. Для решения этой проблемы предлагается использовать парольную защиту программы модификации параметров работы компьютера (пароль на BIOS Setup) с целью не допустить возможности загрузки с флоппи-диска. Данная концепция может быть реализована на практике, но требуемая при этом изоляция компьютеров от вычислительных сетей существенно сужает область ее практического применения. Поэтому сертификация ОС Microsoft Windows NT 3.5 SP3 по классу защиты C2 TCSEC хотя и имела важное значение для дальнейшего совершенствования системы защиты данной ОС, но в основном носила рекламный характер.

Изолированность компьютеров от ЛВС, необходимая для приведения настроек и конфигурации ОС Microsoft Windows NT 3.5 SP3 в соответствие с требованиями класса защиты C2 TCSEC, не могла устроить большинство пользователей данной ОС. В связи с этим следующим шагом корпорации Microsoft была разработка требований к настройке параметров и конфигурации ОС в соответствии с требованиями класса F-C2 и уровня адекватного E3 ITSEC[2] («Европейская Оранжевая книга»), при этом ОС может функционировать на компьютерах, соединенных между собой в единую ЛВС. Сертификация по данному классу защиты ОС Microsoft Windows NT Workstation, Server Version 4.0 (build 1381) Service Part была осуществлена в 1999 году.

Обеспечение безопасности ОС Microsoft Windows NT в соответствии с требованиями класса F-C2 и уровня адекватности E3 ITSEC возможно:

- без установки дополнительного программного или аппаратного обеспечения,
- с помощью соответствующих настроек параметров и порядка конфигурирования ОС

Данная концепция позволяет повысить эффективность защиты ОС, но в то же время содержит ряд недостатков.

[2] http://www.boran.com/security/itsec.htm

1. Не в полном объеме указаны необходимые настройки ОС, связанные с обеспечением безопасной работы на компьютерах, подключенных к ЛВС:
   - отключение возможности удаленной аутентификации по протоколу LANManager;
   - запрет выдачи списка сетевых ресурсов для анонимного пользователя;
   - разрешение удаленного доступа к реестру ресурсов только специально на это уполномоченным пользователям;
   - запрет использования службы удаленного запуска задач (Shedule).
2. Защита данных, передаваемых по коммуникационным каналам ЛВС, от их перехвата без использования дополнительных (в первую очередь криптографических) модулей, как правило, является сложной и труднореализуемой на практике задачей.
3. Не представлен порядок действий администратора домена по предотвращению или сокращению ущерба описанных выше атак, реализующих незаконный захват привилегий.
4. Структура системы защиты домена представлена в концепции (в виде одного рубежа, в ней не приведены требования к настройке и конфигурированию ОС, цель которой сократить ущерб общей безопасности домена при преодолении нарушителем защиты одной из рабочих станций, компрометации аутентификационных данных одного из пользователей и т. д.

Для устранения недостатков группы можно воспользоваться специальными настройками ОС. При устранении трех последующих недостатков целесообразно иметь в виду следующие обстоятельства:

- Основная причина перечисленных недостатков указанной концепции состоит в том, что в качестве ориентира при ее разработке были выбраны требования F-C2 и уровня адекватности E3 ITSEC, которые являются достаточно общими, так как обозначены для оценки безопасности широкого класса систем защиты. Поэтому за рамками рассмотрения при разработке остались указанные в недостатках необходимые для обеспечения безопасности настройки параметров, элементы структуры, требования к порядку функционирования и администрирования ОС. Таким образом, необходим выбор

новой концепции, учитывающей специфику ОС Microsoft Windows NT как сетевой многопользовательской ОС.

- Новую концепцию невозможно построить на основе требований каких-либо других функциональных классов защиты ITSEC, так как требования следующего за F-C2 функционального класса F-B1 невыполнимы, поскольку в них содержится условие реализации мандатной политики безопасности, что практически невозможно без полного изучения ОС, а реализация требований функциональных классов F-AV, F-DI, F-DX не устраняет недостатков 3 и 4 группы.
- При разработке новой концепции необходимо иметь в виду факт недоступности для независимой экспертизы исходных текстов исполняемых модулей и полной документации ОС Microsoft Windows NT, а следовательно, не может быть получено достаточных гарантий корректности работы всех защитных механизмов ОС.
- Не приведет к желаемому результату использование в качестве ориентира требований классов защиты автоматизированных систем, представленных в руководящих документах Гостехкомиссии России. В силу наличия требования реализации мандатной политики безопасности в классах 1 и 2 групп, возможными для использования остаются только классы 3 группы, требования которых в свою очередь не устраняют перечисленных выше недостатков.
- В концепции безопасного администрирования ОС Microsoft Windows NT нельзя не учесть последние достижения в области разработки критериев оценки защищенных компьютерных систем, изложенных в «Единых критериях безопасности информационных технологий». Целесообразно учесть профили защиты, описывающие, например, требование точного определения роли администраторов безопасности и требование выделения необходимого числа доменов в соответствии с политикой безопасности.

Таким образом, устранение недостатков описанной выше концепции приведения настроек системы защиты ОС Microsoft Windows NT в соответствии с требованиями класса F-C2 уровня адекватности ЕЗ ITSEC возможно через разработку и реализацию положений новой концепции безопасного администрирования. Эта новая концепция должна учитывать перечисленные выше

обстоятельства и быть специфичной для ОС иностранного производства, недоступной для полного анализа и изучения.

1.6 Защита информации в ОС Linux

Защита данных. Для контроля целостности данных, которая может быть нарушена в результате как локальных, так и сетевых атак, в Linux используется пакет Tripwire. При запуске он вычисляет контрольные суммы всех основных двоичных и конфигурационных файлов, после чего сравнивает их с эталонными значениями, хранящимися в специальной базе данных. В результате администратор имеет возможность контролировать любые изменения в системе. Целесообразно размещать Tripwire на закрытом от записи гибком магнитном диске и ежедневно запускать.

Безусловно, что для повышения конфиденциальности полезно хранить данные на дисках в зашифрованном виде. Для обеспечения сквозного шифрования всей файловой системы в Linux используются криптографические файловые системы CFS (Cryptographic File System) и TCFS (Transparent Cryptographic File System).

Защита дисплеев. Защита графического дисплея – важный момент в обеспечении безопасности системы. Она направлена на исключение возможности перехвата пароля, ознакомления с информацией, выводимой на экран, и т. п. Для организации этой защиты в Linux предусмотрены следующие средства:

- программа xhost (позволяет указать, каким узлам разрешен доступ к вашему дисплею);
- регистрация с использованием xdm (x display manager) – для каждого пользователя генерируется 128-битный ключ (cookie);
- регистрация с использованием xdm (x display manager) – для каждого пользователя генерируется 128-битный ключ (cookie);

Дополнительно к этому для организации контроля доступа к видеоподсистеме компьютера в рамках Linux разработан проект GGI (Generic Graphics Interface). Идея GGI состоит в переносе части кода, обслуживающего видеоадаптеры, в ядро Linux. С помощью GGI практически исключается возможность запуска на вашей консоли фальшивых программ регистрации.

Сетевая защита. По мере развития сетевых технологий вопросы безопасности при работе в сети становятся все более актуальными. Практика показывает, что зачастую именно сетевые

атаки проходят наиболее успешно. Поэтому в современных ОС сетевой защите уделяется очень серьезное внимание. В Linux для обеспечения сетевой безопасности тоже применяется несколько эффективных средств:

- защищенная оболочка ssh для предотвращения атак, в которых для получения паролей используются анализаторы протоколов;
- программы tcp_wrapper для ограничения доступа к различным службам вашего компьютера;
- сетевые сканеры для выявления уязвимых мест компьютера;
- демон tcpd для обнаружения попыток сканирования портов со стороны злоумышленников (в дополнение к этому средству полезно регулярно просматривать файлы системного журнала);
- система шифрования PGP (Pret-ty Good Privacy);
- программа stelnet (защищенная версия хорошо известной программы telnet);
- программа qmail (защищенная доставка электронной почты);
- программа ipfwadm для настройки межсетевых экранов (firewall);
- режим проверки паролей входных соединений для систем, разрешающих подключение по внешним коммутируемым линиям связи или локальной сети.

Свойство модульной архитектуры Linux – ограничения по умолчанию. Web-навигаторы, работающие в ОС Linux, не поддерживают такие небезопасные по своей природе объекты, как элементы управления ActiveX, но даже если бы они поддерживались, вредоносный элемент ActiveX смог бы запуститься только с полномочиями того пользователя, который запустил web-навигатор. И в этом случае самый большой вред, который он смог бы принести – это заразить или удалить собственные файлы пользователя.

Даже сервисы, например, web-серверы, обычно запускаются как пользователи с ограниченными полномочиями. Так, Debian GNU/Linux запускает web-сервер Apache как пользователя «www-data», принадлежащего к группе с тем же именем «www-data». Если злоумышленник на компьютере с Debian получит полный контроль над web-сервером Apache, он сможет воздействовать только на файлы, принадлежащие пользователю «www-data», то есть на web-страницы. В свою очередь, MySQL, сервер базы данных SQL-типа, часто используемый вместе с Apache, запускается с полномочиями пользователя «mysql». Даже если Apache и MySQL вместе

обслуживают web-страницы, злоумышленник, получивший контроль над Apache, не будет иметь полномочий, позволяющих использовать уязвимость в Apache для получения контроля над сервером базы данных, потому что сервер базы данных «принадлежит» другому пользователю.

Кроме того, пользователи, ассоциированные с такими сервисами, как Apache, MySQL и т.д., часто устанавливаются с учетными записями, не имеющими доступа к командной строке. Поэтому, если злоумышленник сможет получить права учетной записи пользователя MySQL, он не сможет воспользоваться этой уязвимостью для того, чтобы дать произвольные команды на сервер Linux, поскольку данная учетная запись не может вызывать команды.

Linux — это операционная система, сконструированная, в основном, по модульному принципу, от ядра (центрального «мозга» Linux) до приложений. В Linux практически нет нерасторжимых связей между какими-либо компонентами. Не существует единственного процессора web-навигатора, используемого справочными системами или программами электронной почты. В самом деле, нетрудно сконфигурировать большинство программ электронной почты так, чтобы использовать встроенный web-навигатор для отображения HTML-сообщений либо запускать любой нужный web-навигатор для просмотра HTML-документов или перехода по ссылкам, приведенным в тексте сообщения. Следовательно, брешь в одном процессоре web-навигатора необязательно представляет опасность для каких-либо других приложений на данном компьютере, так как почти никакие другие приложения, кроме самого web-навигатора, не зависят от единственного процессора web-навигатора.

Не все в Linux является модульным. Две наиболее популярные графические среды, KDE и GNOME, в каком-то смысле монолитны по своей архитектуре. По крайней мере, монолитны настолько, что в принципе обновление одной части GNOME или KDE может нарушить работу других частей GNOME или KDE. Но и GNOME, и KDE не до такой степени монолитны, чтобы требовалось использование приложений, разработанных специально для GNOME или KDE. Приложения GNOME или любые другие приложения можно запускать под KDE, а KDE или любые другие приложения – под GNOME.

Ядро Linux поддерживает модульные драйверы, но в значительной мере является монолитным ядром, потому что сервисы в этом ядре взаимозависимы. Все отрицательные последствия монолитности минимизируются тем, что ядро Linux, насколько это возможно, разработано как наименьшая часть системы. Linux придерживается следующего принципа: «Если задача может быть выполнена вне ядра, она должна быть выполнена вне ядра». Это означает, что в Linux почти каждая полезная функция («полезная» означает «воспринимаемая конечным пользователем») не имеет доступа к уязвимым частям системы Linux.

2 Защита информации в базах данных

2.1 Классификация угроз безопасности БД

Защита базы данных – это обеспечение защищенности базы данных против любых предумышленных или непредумышленных угроз с помощью различных технических и организационных средств. Далее рассмотрены методы обеспечения безопасности баз данных

Как правило, рассматривают три составляющие защиты данных: обеспечение секретности, целостности и доступности.

Так, например, если ценность информации теряется при ее раскрытии, то говорят, что имеется опасность нарушения секретности информации. Если ценность информации теряется при изменении или уничтожении информации, то говорят, что имеется опасность для целостности информации. Если ценность информации в ее оперативном использовании, то говорят, что имеется опасность нарушения доступности информации. Если ценность информации теряется при сбоях в системе, то говорят, что есть опасность потери устойчивости к ошибкам.

Под целостностью понимается гарантированная сохранность информации в базе данных при отказах аппаратуры или сбоях программ (защита от потери данных), а также обеспечение согласованности информации при попытках ее искажения (защита от искажения данных).

В связи с делением защиты в БД на составляющие, выделяют следующие основные виды угроз безопасности в базах данных:

1) Правовая угроза

В отличие от программы для ЭВМ, компьютерного алгоритма, языка программирования, которые так или иначе предназначены для обеспечения функционирования ЭВМ, база данных служит прежде всего для поиска, восприятия и использования организованных по определенным критериям сведений. От каталогов и сборников на бумажных носителях базу данных в принципе отличает только электронная форма представления и, соответственно, связанные с ней особенности хранения, обработки и поиска находящихся в базе данных сведений. Именно электронная форма базы данных послужила основанием объединения в одном законе положений об охране программ для ЭВМ и баз данных.

В законе Российской Федерации от 23 сентября 1992 г. «О правовой охране программ для электронных вычислительных машин и баз данных» под «базой данных» понимается объективная форма представления и организации совокупности данных (например, статей, расчетов), систематизированных таким образом, чтобы эти данные могли быть найдены и обработаны с помощью ЭВМ.

Предусмотренная Законом Российской Федерации от 23 сентября 1992 г. «О правовой охране программ для электронных вычислительных машин и баз данных» охрана базы данных в качестве сборника характеризуется рядом особенностей. В частности, установлено, что базы данных охраняются независимо от того, являются ли данные, на которых они основаны или которые они включают, объектами авторского права. Авторское право на базу данных, состоящую из материалов, не являющихся объектами авторского права, принадлежит лицам, создавшим базу данных. Если же база данных состоит из охраняемых произведений, то авторское право на нее признается лишь при соблюдении авторского права на каждое из входящих в ее состав произведений, т.е. установление факта нарушения авторского права на любое из произведений, включенных в состав базы данных, является основанием для отнесения такой базы данных к числу неохраноспособных объектов. Кроме того, авторское право на каждое из произведений, включенных в базу данных, сохраняется, и их использование может осуществляться независимо от такой базы данных.

Таким образом, составителю базы данных для включения в ее состав любого охраняемого произведения требуется предварительно получить согласие автора или иного правообладателя такого произведения. В то же время сами они, дав такое разрешение, могут продолжать использование своих произведений по собственному усмотрению (без нарушения положений соответствующего соглашения). Вместе с тем авторское право на базу данных не препятствует другим лицам осуществлять самостоятельный подбор и организацию произведений и материалов, входящих в эту базу данных. Иными словами, это положение допускает создание баз данных какими-либо лицами с использованием тех же произведений и иных материалов, включенных в ранее созданную другими лицами базу данных. На такую вновь созданную базу данных будет распространяться авторское право, если она явилась результатом творческой деятельности.

Данное положение является недостатком в охране базы данных как сборника, поскольку в отличие от первоначального составителя (или автора) базы данных, третьи лица не несут, столь же существенных (качественно или количественно) расходов по сбору, сверке, компоновке и представлению их в соответствующей форме. Это значит, что такие третьи лица могут выходить на рынок со своими базами данных, имеющими, по существу, то же содержание, что и базы данных первоначального составителя, но по цене, меньшей чем у него, по крайней мере, на величину разницы в упомянутых расходов.

2) Несанкционированный доступ

При доступе в информационную систему, основой которой является база данных, пользователь должен идентифицировать себя, а система – проверить подлинность идентификации (произвести аутентификацию).

Идентификация — это присвоение какому-либо объекту или субъекту, реализующему доступ к БД, уникального имени (логина), образа или числового значения. Установление подлинности *(аутентификация)* заключается в проверке, является ли данный объект (субъект) в самом деле тем, за кого себя выдает. Конечная цель идентификации и установления подлинности объекта в вычислительной системе – его допуск к информации ограниченного пользования в случае положительного результата проверки или отказ в допуске при отрицательном результате.

Объектами идентификации и установления подлинности в информационной системе могут быть:

• человек (оператор, пользователь, должностное лицо);

• техническое средство (терминал, дисплей, ЭВМ);

• документы;

• носители информации (диски, магнитные ленты и т. д.);

• информация на дисплее, табло и т. д.

Установление подлинности может производиться человеком, аппаратным устройством, программой, вычислительной системой и т. д.

Как правило, любая процедура идентификации предполагает ввод пользователем своего логина (login) и пароля (password). В зависимости от особенностей функционирования системы пароль выбирается самим пользователем либо назначается администратором (или же иногда его генерирует сама система). Пароль должен быть

таким, чтобы его нельзя было легко раскрыть. Для этого при выборе и использовании пароля рекомендуется руководствоваться следующими правилами:

– пароль не должен содержать личных данных пользователя (таких, как фамилия, имя, серия или номер паспорта либо другого документа, удостоверяющего личность, дата рождения, адрес и т. п.);
– пароль не должен быть словом из какого-либо словаря (входить в какой-либо тезаурус), так как перебор слов заданного словаря — технически достаточно простая задача;
– пароль не должен быть слишком коротким (подобрать сочетание символов в этом случае также не представляет сложности);
– пароль не должен состоять из повторяющихся букв или фрагментов текста;
– пароль не должен состоять из символов, соответствующих подряд идущим клавишам на клавиатуре (например, «QWERTY» — образец недопустимого пароля);
– желательно включать в пароль символы в разных регистрах (прописные и строчные буквы, кириллицу и латиницу, знаки препинания, цифры и др.); чтобы пароль хорошо запоминался, его можно составить из отдельных частей слов, входящих в какую-либо фразу (например, так в связи со знаменательными событиями для некоторых детей когда-то в нашей стране выбирали имена: женское имя Даздраперма — от фразы «Да здравствует первое мая!» или мужское Ювкосур — от фразы «Юрий в космосе. Ура»)

В качестве примера приведем пароль «Деуо,са», составленный из фразы «Доброе утро, страна». При этом соблюдались следующие правила:

• из каждого слова взяты первый и последний символы;
• в пароль включен знак препинания, содержащийся внутри фразы.

Несоблюдение этих и ряда других правил ведет к раскрытию пароля и к возможности несанкционированного доступа к данным. Исследуя этот вопрос, специалисты установили процент раскрываемости пароля в зависимости от тематической группы, в которую он входит (табл. 2).

Таблица 2 – Частота выбора паролей и их раскрываемость

	Тематическая группа	Частота выбора пароля человеком, %	Раскрываемость пароля, %
1	Номера документов (паспорт, пропуск, удостоверение личности, зачетная книжка, страховой полис и пр.)	3,5	99
2	Последовательность клавиш ПК, повторяющиеся символы	14,1	72,3
3	Номера телефонов	3,5	66,6
4	Адрес места жительства (или часть адреса — индекс, город, улица и пр.), место рождения	4,7	55,0
5	Имена, фамилии и производные от них	22,2	54,5
6	Дата рождения или знак зодиака пользователя либо его родственников (возможно, в сочетании с именем, фамилией и производными от них)	11,8	54,5
7	Интересы (спорт, музыка, хобби)	9,5	29,2
8	Прочее	30,7	5,7

Среднее время безопасности пароля определяется по формуле

$$T = (d + \frac{m}{n}) \cdot \frac{S}{2},$$

где d – промежуток времени между двумя неудачными попытками несанкционированного входа в систему, m – количество символов в пароле, n – скорость набора пароля (количество символов, набираемых в единицу времени), S – количество всевозможных паролей указанной длины.

Таким образом, среднее время безопасности пароля фактически равно времени, за которое можно ввести (перебрать) половину всевозможных паролей заданной длины. Однако большинство

информационных систем предусматривают возможность ввода идентифицирующих данных не более заданного количества раз (как правило не более трех раз за один сеанс работы). В такой ситуации задача нарушителя значительно усложняется. Очевидно, что нарушитель прежде всего постарается войти в систему, введя пароль из той тематической группы, где процент раскрываемости достаточно высок.

3) Внедрение через SQL

Многие разработчики Web-приложений считают запросы SQL не стоящими внимания, не зная о том, что их может использовать злоумышленник. Это означает, что запросы SQL могут быть использованы для обхода систем защиты, аутентификации и авторизации, а также иногда могут быть использованы для получения доступа к командам уровня операционной системы.

Внедрение в команды SQL – техника, при которой злоумышленник создает или изменяет команды SQL для получения доступа к скрытым данным, для изменения существующих и даже для выполнения команд уровня операционной системы. Это достигается в том случае, если программа использует введенные данные в комбинации со статическими параметрами для создания запроса SQL.

При недостаточной проверке вводимых данных и соединении с базой данных на правах суперпользователя злоумышленник может создать нового суперпользователя в базе данных.

Существует путь получения паролей через страницы поиска. Все, что нужно злоумышленнику – это одна не обработанная должным образом переменная, используемая в SQL-запросе. Использоваться могут команды WHERE, ORDER BY, LIMIT и OFFSET запроса SELECT. Если база данных поддерживает конструкцию UNION, злоумышленник может добавить к исходному запросу еще один – для получения паролей. В этом случае поможет хранение зашифрованных паролей.

```
$query  = «SELECT id, name, inserted, size FROM products
                  WHERE size = '$size'
                  ORDER BY $order LIMIT $limit, $offset;»;
$result = odbc_exec($conn, $query);
```

Статическая часть запроса может быть совмещена с другим запросом SELECT, который выведет все пароли:

```
union select '1', concat(uname||'-'||passwd) as name, '1971-01-
01', '0' from usertable;
```

Если подобный запрос (использующий ' и -) будет задан в одной из переменных, используемых $query, то атака будет успешной.

Запросы SQL «UPDATE» также могут быть использованы для атаки на базу данных. Эти запросы также подвержены опасности «обрезки» и добавления новых запросов. Но здесь злоумышленник работает с командой SET. В этом случае необходимо знание некоторой информации о структуре базы данных для удачной модификации запроса. Такая информация может быть получена путем изучения названий переменных форм или просто подбором. В конце концов, не так уж и много имен придумано для полей пользователей и паролей.

```
$query = «UPDATE usertable SET pwd='$pwd' WHERE uid='$uid';»;
```

Злоумышленник посылает значение ' or uid like'%admin%'; --, в переменную $uid для изменения пароля администратора или просто устанавливает $pwd в «hehehe', admin='yes', trusted=100 « (с завершающим пробелом) для получения прав. Запрос будет искажен так:

```
// $uid == ' or uid like'%admin%'; --
$query = «UPDATE usertable SET pwd='...' WHERE uid='' or uid like
'%admin%'; --»;
// $pwd == «hehehe', admin='yes', trusted=100 «
$query = «UPDATE usertable SET pwd='hehehe', admin='yes',
trusted=100 WHERE ...;»
```

Пример того, как на некоторых серверах баз данных могут быть выполнены команды уровня ОС (атака на операционную систему сервера баз данных MSSQL):

```
$query  = «SELECT * FROM products WHERE id LIKE '%$prod%'«;
$result = mssql_query($query);
```

Если злоумышленник пошлет значение

```
a%' exec master..xp_cmdshell 'net user test testpass /ADD' –
```

в $prod, то $query будет выглядеть так:

```
$query  = «SELECT * FROM products
                    WHERE id LIKE '%a%'
                    exec master..xp_cmdshell 'net user test
testpass /ADD'--»;
```

```
$result = mssql_query($query);
```

Сервер MSSQL выполняет все команды SQL, включая команду добавления нового пользователя в локальную базу данных пользователей. Если это приложение было запущено, как sa и служба MSSQLSERVER имеет достаточно прав, злоумышленник будет иметь учетную запись для доступа к этой машине.

2.2 Простейшая модель безопасности баз данных

На самом элементарном уровне концепции обеспечения безопасности баз данных подразумевает поддержку двух фундаментальных принципов: проверку полномочий и проверку подлинности.

Для работы с базой данных каждый пользователь получает определенный набор полномочий, т.е., как было сказано выше, определенный набор операций, которые он может выполнять над объектами базы данных. Базовая модель проверки полномочий представлена на рисунке 3.

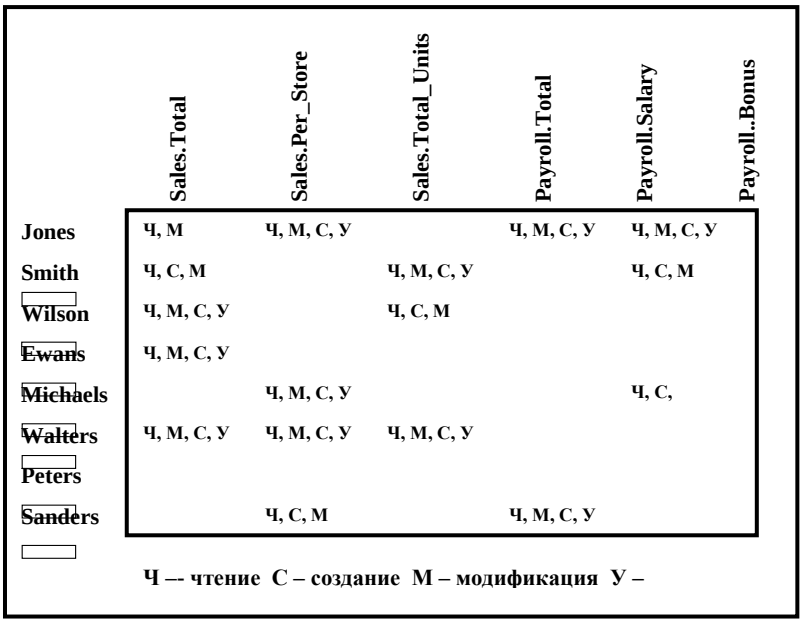

Рисунок 3 – Простая модель проверки полномочий

То есть теоретически возможно поддерживать матрицу безопасности, устанавливающую отношения между всеми пользователями и процессами системы с одной стороны и всеми объектами с другой. В каждой клетке матрицы хранится список, содержащий от 0 до N операций, которые данный пользователь или

процесс может выполнять по отношению к данному объекту. Система управления безопасностью, которая поддерживала бы такую матрицу и обеспечивала бы ее применение при выполнении любой операции в данной среде, могла бы гарантировать высокий уровень защиты информации. Однако в реальности все намного сложнее. На рисунке 4 показано, почему такая базовая модель недостаточна для обеспечения хотя бы минимального уровня безопасности.

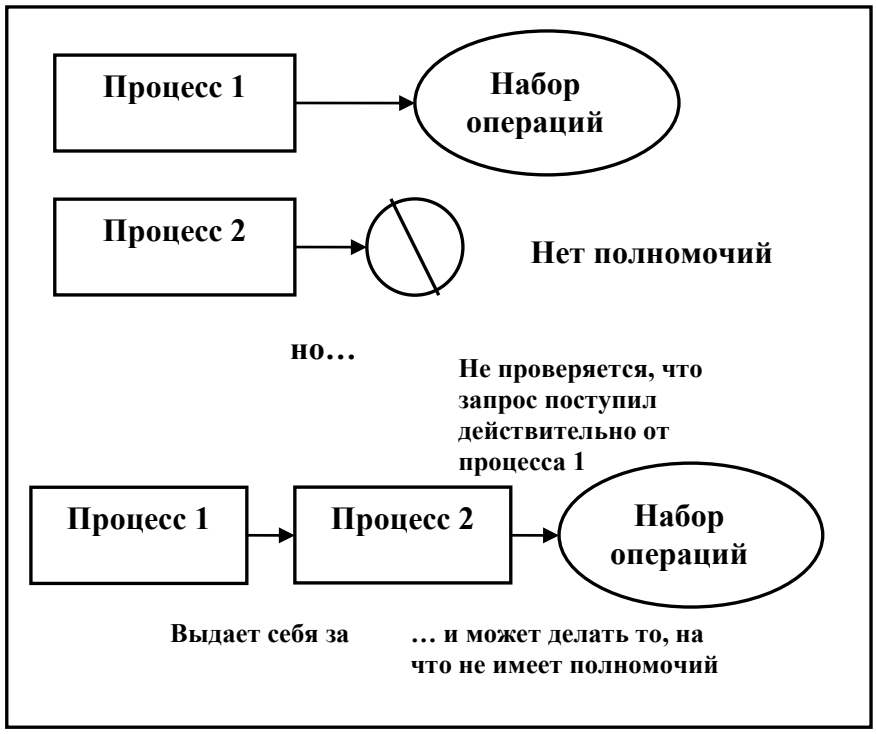

Рисунок 4 – Проблемы, возникающие при использовании базовой модели проверки полномочий

Если Процессу 2 удастся выдать себя за Процесс 1, то он сможет выполнять действия, разрешенные Процессу 1 (но не Процессу 2). Поэтому очевидно, что необходимы какие-то дополнительные меры обеспечения безопасности. Такими мерами являются идентификация пользователей и проверка их подлинности.

Проверка полномочий приобрела еще большее значение в условиях массового распространения распределенных вычислений. При существующем высоком уровне связности вычислительных систем необходимо контролировать все обращения к системе.

Для аутентификации может быть затребована информация различного характера, прежде чем подлинность будет признана установленной. Обычно подлинность устанавливается один раз, однако в системах, обеспечивающих высокую степень безопасности,

может потребоваться периодическая перепроверка или перепроверка в определенных ситуациях. Повторное установление подлинности может быть желательно, например, после всех системных сбоев.

Для установления подлинности пользователей используются пароли, известные только системе и законному пользователю с указанным номером и другие методы диалога. Если пользователь в состоянии правильно представить требуемую информацию, признается его подлинность.

Процесс подтверждения подлинности включает предоставление информации, известной только пользователю. Этого можно достичь указанием пароля (что было рассмотрено выше), или путем ответа на некоторый вопрос системы. Одна из таких процедур, описана автором книги «Современные методы защиты информации» Хаффманом Л.Дж., и состоит в следующем. Сначала система передает пользователю выбранное случайным образом число x. Пользователь в уме осуществляет некоторое простое преобразование T этого числа x и передает результат y=T(x) обратно системе. Система выполняет такое же преобразование исходного числа x и сравнивает результат с y. Любой сторонний наблюдатель, увидит только значения x и y, из которых практически невозможно точно установить вид преобразования, т.к. сама функция преобразования никогда не передается по линиям связи, по которым передаются только x и T(x).

Эта схема удобна для определения подлинности, поскольку необходимые вычисления довольно просты, и никакие пароли не нужно помещать на хранение. Функция преобразования может быть различной для каждого пользователя. Функцию преобразования при желании можно изменить в зависимости от некоторого внешнего события.

2.3 Многоуровневая модель безопасности баз данных

Сочетание средств проверки полномочий и проверки подлинности – мощное орудие борьбы за безопасность информационных систем и баз данных. Если все пользователи, работающие в интерактивном режиме или запускающие пакетные приложения, достаточно надежны и имеют доступ к максимально закрытой информации, хранимой в системе, то упомянутый подход к обеспечению безопасности может быть вполне достаточным.

Например, если в системе хранится информация, классифицированная по уровням от полностью открытой до совершенно секретной, но все пользователи системы имеют доступ к самым секретным данным, то в такой системе достаточно иметь надежные механизмы проверки полномочий и проверки подлинности. Такая модель называется «работой на высшем уровне (секретности)» (running at system high).

Однако она оказывается неудовлетворительной, если в учреждении необходимо организовать действительно многоуровневую среду защиты информации. Многоуровневая защита означает:

1. В вычислительной системе хранится информация, относящаяся к разным классам секретности;
2. Часть пользователей не имеют доступа к максимально секретному классу информации.

Классический пример подобной среды – вычислительная система военного учреждения, где в логически единой базе данных (централизованной или распределенной) может содержаться информация от полностью открытой до совершенно секретной, при этом степень благонадежности пользователей также варьируется от допуска только к несекретной информации до допуска к совершенно секретным данным. Таким образом, пользователь, имеющий низший статус благонадежности, может выполнять свою работу в системе, содержащей сверхсекретную информацию, но ни при каких обстоятельствах не должен быть допущен к ней.

Многоуровневая зашита баз данных строится обычно на основе модели Белла-ЛаПадула, которая предназначена для управления субъектами, т. е. активными процессами, запрашивающими доступ к информации, и объектами, т. е. файлами, представлениями, записями, полями или другими сущностями данной информационной модели [1].

Объекты подвергаются классификации, а субъекты причисляются к одному из уровней благонадежности. Классы и уровни благонадежности называются классами или уровнями доступа.

Класс доступа характеризуется двумя компонентами. Первый компонент определяет иерархическое положение класса. Второй компонент представляет собой множество элементов из неиерархического набора категорий, которые могут относиться к

любому уровню иерархии. Например, возможна следующая иерархия классов (сверху вниз):

1. Совершенно секретно;
2. Секретно;
3. Конфиденциально;
4. Несекретно.

Второй компонент может относиться, например, к набору категорий:

1. Ядерное оружие;
2. Недоступно для иностранных правительств;
3. Недоступно для служащих по контракту.

Очевидно, что можно определить матрицу соотношений между иерархическими и неиерархическими компонентами. Например, если некоторый объект классифицирован как совершенно секретный, но ему не приписана ни одна из категорий неиерархического набора, то он может предоставляться иностранным правительствам, в то время как менее секретный объект может иметь категорию «недоступно для иностранных правительств» и, следовательно, не должен им предоставляться. Однако в модели Белла-ЛаПадула создается «решетка», где неиерархические компоненты каждого уровня иерархии автоматически приписываются всем более высоким уровням (так называемое «обратное наследование»). Эта модель отображена на рисунке 5.

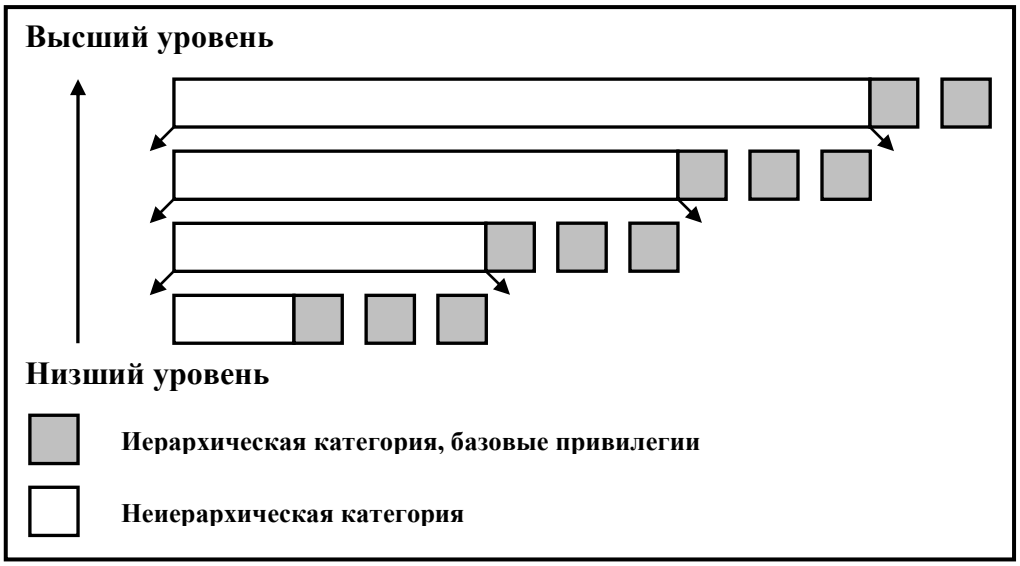

Рисунок 5 – Решетка классов доступа в модели Белла-ЛаПадулла

В модели Белла-ЛаПадула существует два основных принципа. Первый известен под названием простого свойства секретности

(simple security property). Он состоит в следующем: субъект класса CL1 имеет доступ к объектам класса CL2, если CL2 не выше CL1. Например, пользователь класса «совершенно секретно», имеет доступ к информации классов «совершенно секретно», «секретно», «конфиденциально», «несекретно».

Менее очевидно другое свойство, обозначаемое символом *, которое позволяет субъекту иметь право на запись в объект только в том случае, если класс субъекта такой же или ниже, чем класс записываемого объекта. Это означает, что информация, принадлежащая какому-либо уровню секретности, никогда не может быть записана в какой-либо объект, имеющий более низкий уровень секретности, поскольку это могло бы привести по неосторожности к деградации защиты классифицированной информации.

Как и в большинстве областей вычислительных средств и информационного управления и, в особенности, в области безопасности, приведенная выше простая модель в реальности представляет не слишком большую ценность, подобно упрощенным моделям проверки полномочий и подлинности. Хотя принципы Белл-ЛаПадула достаточно неплохи, но требование поддержки нескольких уровней защиты в пределах одной таблицы (или в одной базе данных) сопряжено с рядом серьезных проблем. Возможно, например, возникновение ситуации, когда один и тот же человек может иметь «настоящую» миссию и миссию «прикрытие» (например, в военных учреждениях) (Рисунок 6). При этом в базе данных должна быть представлена и реальная информация, и прикрытие. Такую проблему можно решить, используя, например, многозначные кортежи.

Фамилия	Класс	Звание	Класс	Специальность	Класс	Класс кортежа
Джонс	Н	Сержант	Н	Программист	Н	Н
Джонс	Н	Капитан	С	Программист	Н	С
Мартин	Н	Майор	Н	Оркестрант	Н	Н
Мартин	Н	Майор	Н	Разведчик	С	С

Н – несекретно С – секретно

Рисунок 6– Пример многозначного отношения

2.4 Многозначность

Идея многозначности заключается в том, что в рамках одного отношения может существовать множество кортежей с одним и тем же значением первичного ключа.

Многозначность впервые была применена в 1988 г. в модели безопасной базы данных SeaView (Secure Data Views). За прошедшие с тех пор годы в этом направлении были проведены значительные исследования и написано немало работ.

В отсутствие многозначности отношение с многоуровневой защитой, рассмотренное в конце предыдущего раздела, следовало бы обрабатывать следующим образом (рисунок 7). База данных может маскировать значения, недоступные пользователю или процессу по соображениям безопасности, и такое маскирование обычно осуществляется при помощи пустых значений (null values).

Рисунок 7 – Маскирование представления данных с целью обеспечения секретности

Каждый кортеж существует в единственном экземпляре, поэтому непривилегированный процесс может попытаться записать неклассифицированные данные в столбцы кортежа, содержащие как бы пустые значения (рисунок 8). В связи с этим в СУБД с

многоуровневой защитой возникает проблема обработки таких операций, поскольку на самом деле значение столбца непустое, и замещать его нельзя. Естественно, что подобные попытки должны отвергаться средствами системы безопасности.

Однако неудача при выполнении операции модификации открывает определенные возможности, получившие название косвенных каналов (для получения закрытой информации, которые будут рассмотрены ниже), т. е. пути для снижения классификации данных (преднамеренного или случайного). Отказ в выполнении вполне законной транзакции на модификацию пустых значений в базе данных с многоуровневой защитой выдает информацию о том, что эти элементы на самом деле не пусты, а содержат классифицированные данные. Уведомляя не наделенного соответствующими полномочиями пользователя или процесс о наличии в том или ином кортеже классифицированных данных, тем самым открывается косвенный канал.

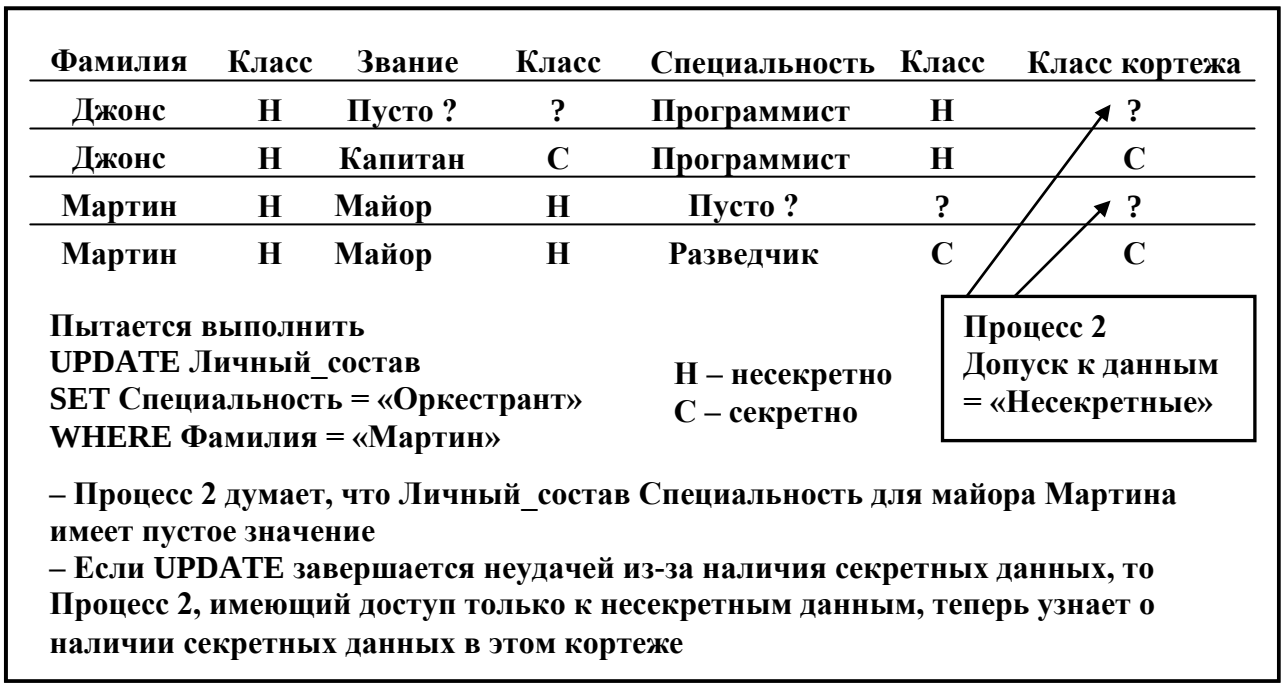

Рисунок 8 – Попытка модификации маскированных объектов

Возможно, СУБД с многоуровневой защитой симулирует выполнение подобных модификаций, чтобы не раскрывать скрытый канал. Но, если пользователь, только что изменивший некоторые данные, попытается выполнить запрос или приложение, которое

обращается к этим данным, а ему ответят, что они отсутствуют, значит, опять создается косвенный канал.

Таким образом, многозначность – это существенное свойство данных с многоуровневой защитой, а ее реализация – это вопрос второстепенный, не имеющий отношения к представлению информации для пользователей разных уровней благонадежности. Возможно различные способы реализации многозначности, например, хранить два экземпляра кортежей. Способы реализации многозначности остаются важным направлением для исследовательской работы.

Итак, многозначность, примененная впервые в проекте SeaView, стала общепринятой моделью реализации баз данных с многоуровневой защитой, в особенности реляционных. Многозначность активно исследуется и в качестве механизма обеспечения безопасности также и в объектно-ориентированных базах данных, развитие средств защиты которых в настоящее время еще значительно отстает по сравнению с реляционными СУБД.

3 Защита информации от разрушающих программных воздействий

3.1 Понятие разрушающего программного воздействия

Существующая на сегодняшний день концепция построения защищенных компьютерных систем (КС) подразумевает использование в едином комплексе программных средств различного назначения. Так, система автоматизированного документооборота банка может использовать на одних и тех же аппаратных средствах (например, компьютере, аппаратуре передачи данных (модемах) и т.д.) взаимосвязанный комплекс программных средств: операционную среду, программные средства управления базой данных (СУБД), телекоммуникационные средства, средства обработки текстов (редакторы, текстовые процессоры), возможно, также средства антивирусного контроля, разграничения доступа к программам и данным, средства криптографической защиты передаваемой и хранимой информации, средства криптографической идентификации и аутентификации (электронная цифровая подпись) и многое другое.

Важным моментом при работе прикладных программ, в особенности средств защиты информации, является необходимость потенциального невмешательства иных присутствующих в КС прикладных или системных программ в процесс обработки информации.

Под несанкционированным доступом (НСД) к ресурсам защищенной КС понимаются действия по использованию, изменению и уничтожению исполняемых модулей и массивов данных, принадлежащих указанной системе, производимые субъектом, не имеющим права на такие действия. Данного субъекта будем называть злоумышленником (нарушителем). Остальные субъекты именуются легальными пользователями. Данное априорное деление предполагает несколько существенно важных моментов:

- система имеет механизм различения злоумышленников и легальных пользователей;
- в системе имеются пассивные и активные компоненты (исполняемые модули и данные), пользование которыми злоумышленником нежелательно;
- в системе имеется механизм установления соответствия субъекта и информации, к которой он имеет доступ.

Как уже отмечалось, в настоящее время для интегрального обозначения процедур обеспечения безопасности информации употребляют «политика безопасности», а всевозможные ситуации нарушения априорно предписанных правил называют нарушениями безопасности.

Считая, что злоумышленник в совершенстве владеет всем программным и аппаратным обеспечением системы, можно предполагать, что несанкционированный доступ может быть вызван следующими причинами:

- отключением или видоизменением защитных механизмов злоумышленником (сюда же можно отнести процедуры доступа «мимо» средств контроля, например, при нарушении уровня иерархии информационных объектов – доступ к объектам типа «файл» как к последовательности секторов и др.);
- входом злоумышленника в систему под именем и с полномочиями легального пользователя.

В первом случае злоумышленник должен видоизменить защитные механизмы в системе (например, отключить программу запросов паролей пользователей). Во втором – каким-либо образом выяснить или подделать идентификатор реального пользователя (например, «подсмотреть» пароль, вводимый с клавиатуры). В обоих случаях НСД можно представить моделью опосредованного несанкционированного доступа – когда проникновение в систему осуществляется на основе некоторого воздействия, произведенного предварительно внедренной в систему программой (программами).

Например, злоумышленник пользуется информацией, которая извлечена из некоторого массива данных, созданного при совместной работе программного средства злоумышленника и системы проверки прав доступа (т.е. предварительно внедренная в систему программа при доступе легального пользователя перехватит его пароль и сохранит в заранее известном и доступном злоумышленнику месте, а затем злоумышленник воспользуется данным паролем для входа в систему). Либо злоумышленник изменит часть системы защиты так, чтобы она перестала выполнять свои функции. Например, модифицирует систему проверки прав доступа так, чтобы она пропускала любого пользователя, или изменит программу шифрования вручную (или при помощи некоторой другой

программы) таким образом, чтобы она перестала шифровать или изменила алгоритм шифрования на более простой.

3.2 Программа с потенциально опасными последствиями

Программой с потенциально опасными последствиями (badware -»вредные программы») назовем некоторую программу (осмысленный набор инструкций для какого-либо процессора), которая способна выполнить любое непустое подмножество перечисленных функций:

1. скрыть признаки своего присутствия в программной среде КС;
2. реализовать самодублирование, ассоциирование себя с другими программами и/или перенос своих фрагментов в иные (не занимаемые изначально указанной программой) области оперативной или внешней памяти;
3. разрушить (исказить произвольным образом) код программ (от
личных от нее) в оперативной памяти КС;
4. перенести (сохранить) фрагменты информации из оперативной
памяти в некоторые области оперативной или внешней памяти прямого
доступа (локальных или удаленных);
5. имеет потенциальную возможность исказить произвольным образом, заблокировать и/или подменить выводимый во внешнюю память или в канал связи массив информации, образовавшийся в результате работы прикладных программ или уже находящийся во внешней памяти, либо изменить его параметры.

Программы с потенциально опасными последствиями можно (весьма условно) разделить на три класса.

1. Классические программы – «вирусы» (термин применен в 1984 г.
Ф.Коэном). Особенность данного класса вредных программ заключается в
не направленности их воздействия на конкретные типы прикладных программ, а также в том, что во главу угла ставится самодублирование вируса. Разрушение информации вирусом не направлено на конкретные программы и встречается у 10...20 %

вирусов.

2. Программы типа «программный червь» или «троянский конь» и
фрагменты программ типа «логический люк». В данном случае имеет место обратная ситуация – самодублирование не всегда присуще такого рода программам или фрагментам программ, но они обладают возможностью перехвата конфиденциальной информации, или извлечения информации из сегментов систем безопасности, или разграничения доступа.

3. Программные закладки или разрушающие программные воздействия (РПВ) – обобщенный класс программ (в смысле отсутствия конкретных признаков) с потенциально опасными последствиями, обязательно реализующие хотя бы один из пп. 3-5 определения программы с потенциально опасными последствиями.

Далее наряду с термином «программа с потенциально опасными последствиями» будут использованы термины «закладка», «программная закладка» либо сокращение РПВ.

Кроме того, программные закладки можно классифицировать по методу и месту их внедрения и применения (т.е. по «способу доставки» в компьютерную систему):

- закладки, ассоциированные с программно-аппаратной средой компьютерной системы (основная BIOS или расширенные BIOS);
- закладки, ассоциированные с программами первичной загрузки (находящиеся в Master Boot Record или BOOT-секторах активных разделов),
- загрузочные закладки;
- закладки, ассоциированные с загрузкой драйверов DOS, драйверов внешних устройств других ОС, командного интерпретатора, сетевых драйверов, т.е. с загрузкой операционной среды;
- закладки, ассоциированные с прикладным программным обеспечением
  общего назначения (встроенные в клавиатурные и экранные драйверы,
  программы тестирования компьютеров, утилиты);
- исполняемые модули, содержащие только код закладки (как правило,
  внедряемые в файлы пакетной обработки типа. BAT);

- модули-имитаторы, совпадающие по внешнему виду с некоторыми программами, требующими ввода конфиденциальной информации – наиболее характерны для Unix-систем
- закладки, маскируемые под программные средства оптимизационного назначения (архиваторы, ускорители обмена с диском и т.д.);
- закладки, маскируемые под программные средства игрового и развлекательного назначения (как правило, используются для первичного внедрения закладок).

Как видно, программные закладки имеют много общего с классическими вирусами, особенно в части ассоциирования себя с исполняемым кодом (загрузочные вирусы, вирусы-драйверы, файловые вирусы). Кроме того, программные закладки, как и многие известные вирусы классического типа, имеют развитые средства борьбы с отладчиками и дизассемблерами.

Для того чтобы закладка смогла выполнить какие-либо действия по отношению к прикладной программе или данным, она должна получить управление, т.е. процессор должен начать выполнять инструкции (команды), относящиеся к коду закладки. Это возможно только при одновременном выполнении двух условий:

1. закладка должна находиться в оперативной памяти до начала работы программы, которая является целью воздействия закладки, следовательно, она должна быть загружена раньше или одновременно с этой программой;
2. закладка должна активизироваться по некоторому общему, как для закладки, так и для программы событию, т.е. при выполнении ряда условий в программно-аппаратной среде управление должно быть передано программе-закладке. Данное событие далее будем называть активизирующим.

Обычно выполнение указанных условий достигается путем анализа и обработки закладкой общих относительно закладки и прикладной программы воздействий (как правило, прерываний) либо событий (в зависимости от типа и архитектуры операционной среды). Причем прерывания должны сопровождать работу 'прикладной программы или работу всего компьютера. Данные условия являются необходимыми (но недостаточными), т.е. если они не выполнены, то

активизация кода закладки не произойдет и код не сможет оказать какого-либо воздействия на работу прикладной программы.

Кроме того, возможен случай, когда при запуске программы (активизирующим событием является запуск программы) закладка разрушает некоторую часть кода программы, уже загруженной в оперативную память (ОП), и, возможно, систему контроля целостности кода или контроля иных событий и на этом заканчивает свою работу. Данный случай не противоречит необходимым условиям.

С учетом замечания о том, что закладка должна быть загружена в ОП раньше, чем цель ее воздействий, можно выделить закладки двух типов.

1. Закладки резидентного типа – находятся в памяти постоянно с некоторого момента времени до окончания сеанса работы компьютера (выключения питания или перезагрузки). Закладка может быть загружена в память при начальной загрузке компьютера, загрузке операционной среды или запуске некоторой программы (которая по традиции называется вирусоносителем или просто носителем), а также запущена отдельно.

2. Закладки нерезидентного типа – начинают работу, как и закладки резидентного типа, но заканчивают ее самостоятельно через некоторый промежуток времени или по некоторому событию, при этом выгружая себя из памяти целиком.

3.3 Модели взаимодействия прикладной программы и программной закладки

Итак, мы сформулировали общее понятие программной закладки или РПВ. Теперь попытаемся обозначить области их воздействия на компьютерные системы. Для этого рассмотрим следующие модели закладок (в смысле описания некоторого класса возможных действий злоумышленника), исходящие из образа мыслей и возможностей злоумышленника или, иначе говоря, из модели злоумышленника.

1. Модель «перехват». Программная закладка встраивается (внедряется) в ПЗУ, ОС или прикладное программное обеспечение и сохраняет все или избранные фрагменты вводимой (с внешних устройств) или выводимой (на жесткий диск, принтер и др.) информации в скрытой области локальной или удаленной внешней

памяти прямого доступа. Объектом сохранения может быть клавиатурный ввод (либо целиком, либо избранные последовательности, например повторяемые два раза), документы, выводимые на принтер, или уничтожаемые файлы-документы. Данная модель может быть двухэтапной: на первом этапе сохраняются только атрибутивные признаки (например, имена или начала файлов), затем накопленная информация снимается и злоумышленник принимает решение о конкретных объектах дальнейшей атаки. Для данной модели существенно наличие во внешней памяти места хранения информации, которое должно быть организовано таким образом, чтобы обеспечить ее сохранность на протяжении заданного промежутка времени и возможность последующего съема, Важно также, чтобы сохраняемая информация была каким-либо образом замаскирована от просмотра легальными пользователями.

2. Модель «троянский конь". Закладка встраивается в постоянно используемое программное обеспечение и по некоторому активизирующему событию моделирует сбойную ситуацию на средствах хранения информации или в оборудовании компьютера (сети). Тем самым могут быть достигнуты две различные цели: во-первых, парализована нормальная работа компьютерной системы и, во-вторых, злоумышленник (под видом ремонта) может ознакомиться с имеющейся в системе или накопленной по модели «перехват» информацией. Событием, активизирующим закладку, может быть некоторый момент времени, либо сигнал из канала модемной, связи (явный или замаскированный), либо состояние некоторых счетчиков (например, число запусков программ).

3. Модель «наблюдатель». Закладка встраивается в сетевое или телекоммуникационное программное обеспечение. Пользуясь тем, что данное ПО, как правило, всегда активно, программная закладка осуществляет контроль за процессами обработки информации на данном компьютере, установку и удаление закладок, а также съем накопленной (первая модель, «сохранение») информации. Закладка может инициировать события ранее внедренных закладок, действующих по модели «троянский конь»

4. Модель «компрометация». Закладка либо передает заданную злоумышленником информацию (например, клавиатурный ввод) в канал связи или сохраняет ее, не полагаясь на гарантированную возможность последующего приема или снятия, иногда закладка

инициирует постоянное обращение к информации, приводящее к росту отношения сигнал/шум при перехвате побочных излучений.

5. Модель «искажение или инициатор ошибок». Программная закладка искажает потоки данных, возникающие при работе прикладных программ (выходные потоки), либо искажает входные потоки информации, либо инициирует (или подавляет) возникающие при работе прикладных программ ошибки.

6. Модель «уборка мусора». В данном случае прямого воздействия РПВ может и не быть; изучаются «остатки» информации. В случае применения программной закладки навязывается такой порядок работы, чтобы максимизировать количество остающихся фрагментов ценной информации. Злоумышленник получает либо данные фрагменты, используя закладки моделей 2 и 3, или непосредственный доступ к компьютеру под видом ремонта или профилактики.

У рассмотренных различных по целям воздействия моделей закладок имеется важная общая черта – наличие операции записи, производимой закладкой (в оперативную или внешнюю память). При отсутствии данной операции никакое негативное влияние невозможно. Вполне понятно, что для направленного воздействия закладка должна также выполнять и операции чтения (иначе можно реализовать лишь функцию разрушения – например, целенаправленная модификация данных в каком-либо секторе жесткого диска возможна только после их прочтения).

Несанкционированная запись закладкой может происходить:

- в массив данных, не совпадающий с пользовательской информацией, сохранение информации;
- в массив данных, совпадающий с пользовательской информацией или ее подмножеством, – искажение, уничтожение или навязывание информации закладкой.

Следовательно, можно рассматривать три основные группы деструктивных функций, которые могут выполняться закладками:

1. сохранение фрагментов информации, возникающей при работе

пользователя, прикладных программ, вводе выводе данных, во внешней памяти (локальной или удаленной) сети или выделенном компьютере, в том числе различных паролей, ключей и кодов доступа, собственно конфиденциальных документов в электронном виде, либо безадресная компрометация фрагментов ценной

информации (модели «перехват», «компрометация»);

2. изменение алгоритмов функционирования прикладных программ (т.е. целенаправленное воздействие во внешней или оперативной памяти) – происходит изменение собственно исходных алгоритмов работы программ, например программа разграничения доступа станет пропускать пользователей по любому паролю (модели «искажение», «троянский конь»);

3. навязывание некоторого режима работы (например, при уничтожении информации – блокирование записи на диск, при этом информация, естественно, не уничтожается) либо замена записываемой информации, навязанной закладкой.

Итак, можно выделить следующие компоненты программной среды, в которой существует закладка: множество фрагментов кода прикладных программ, множество фрагментов кода закладки (закладок) и множество событий (как последовательностей передачи управления от одного фрагмента кода к другому) в программной среде. В последнем множестве выделяются события, по которым управление передается подмножеству фрагментов кода закладки – далее будем называть их активизирующими событиями.

3.4 Методы перехвата и навязывания информации

В компьютерной системе в любом случае происходит ввод и вывод информации (иначе в обработке информации нет никакого смысла: компьютерная система либо представляет собой систему без входа, либо систему без выхода). Для злоумышленника зачастую весьма важна и вводимая, и выводимая информация.

Например, в случае идентификации пользователя он вводит в компьютер (непосредственно с клавиатуры либо через какое-либо устройство типа считывателя кредитной карты) свою оригинальную, индивидуальную информацию. Злоумышленник, определив ее, сможет работать в дальнейшем за данного пользователя. Основной путь ввода документов в компьютере – клавиатура, реже – сканер, для вывода – принтер.

Чтобы оценить, что в системе интересно для перехвата ввода вывода, надо первоначально понять технологию работы компьютерной системы, а именно: как образуется, изменяется, циркулирует, уничтожается информация в ней и каким образом этот процесс управляется.

Прикладная программа выделяет себе в оперативной памяти некоторую информативную область (как правило, доступную для непосредственного восприятия: область экрана, клавиатурный буфер и т.д.), куда помещается информация для обработки. Закладка определяет адрес информативной области программы (иногда этот адрес фиксирован). Далее необходимо анализировать события, связанные с работой прикладной программы или операционной среды, причем интерес представляют лишь события, результатом которых может стать появление интересующей злоумышленника информации в информативной области. Установив факт наступления интересующего события, закладка переносит часть информативной области либо всю информативную область в свою область сохранения (непосредственно на диск или в выделенную область оперативной памяти). Сохранение фрагментов вводимой и выводимой информации можно представить схемой, показанной на рисунке 9.

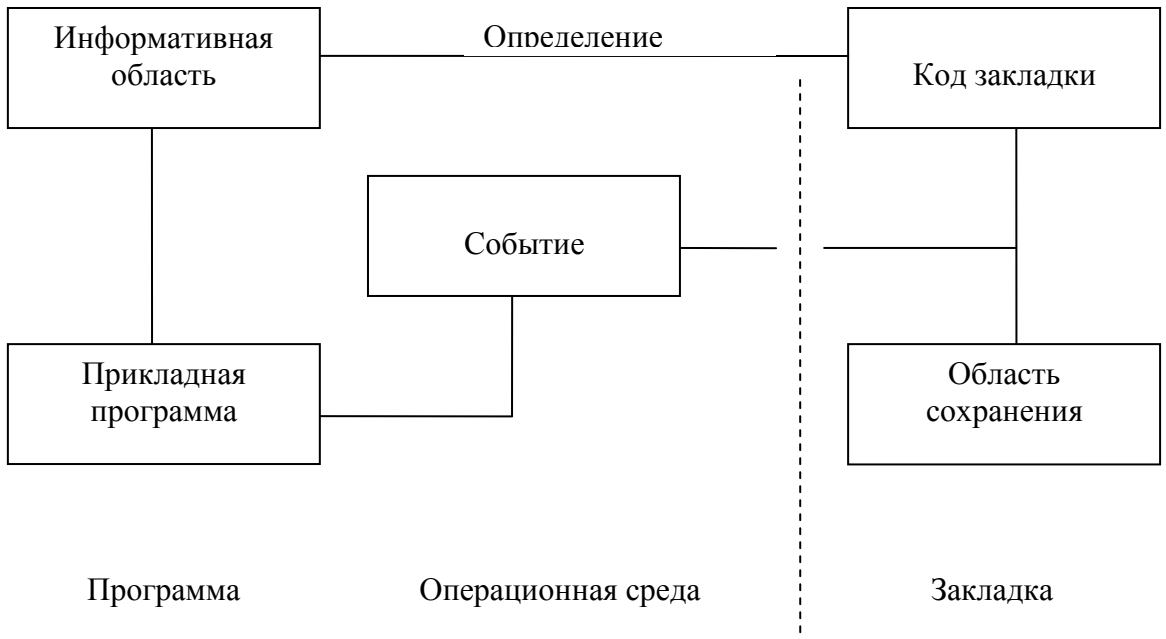

Рисунок 9 – Сохранение фрагмента информации

Итак, существенно следующее. Закладка должна иметь доступ к информативной области, где накапливается вводимая или выводимая информация (как минимум, знать ее адрес), и иметь место для сохранения информации (причем это место должно быть скрытым).

Закладки, перехватывающие клавиатурный ввод, очень опасны и встречаются чаще всего, поскольку клавиатура является основным устройством управления и ввода информации. Через клавиатурный ввод можно получить информацию о вводимых конфиденциальных сообщениях (текстах), паролях и фактически обо всех действиях оператора.

Перехват может происходить в основном следующими способами:

- встраиванием в цепочку прерывания int 9h;
- анализом содержания клавиатурного порта или буфера клавиш по таймерному прерыванию.

Перехвату может подвергаться и собственно содержимое файлов, к которым обращается легальный пользователь. Как правило, перехват имеет смысл тогда, когда после обращения легального пользователя файл изменился так, что его содержимое стало недоступно для непосредственного восприятия (например, файл был зашифрован пользователем). Тогда злоумышленник модифицирует операцию «открыть файл», совмещая ее с созданием файла-области сохранения, записью из исходного файла в область сохранения и с закрытием файла-области сохранения.

Часто закладки-перехватчики файловых операций нацелены на операцию уничтожения файла. Операция уничтожения совмещается с копированием файла в область сохранения.

Рассмотрим перехват файловых операций применительно к DOS. Предположим, что прикладная программа производит некоторые файловые операции. Для этого файл открывается, часть его или весь файл считывается в буфер оперативной памяти, обрабатывается прикладной программой, а затем прикладная программа формирует файл с прежним или новым именем (файл-результат).

Активизирующими событиями в данном случае являются, как правило – открытие файла (int 21 h, функция 3Dh), его закрытие либо уничтожение.

Выделим две основные функции: преобразование и сохранение. Сохранение может касаться исходного файла, файла-результата (если рассматривать обратные криптографические преобразования – расшифровка) вспомогательного файла, который может содержать дополнительную (ключевую) информацию. Преобразование изменяет

информативные атрибуты исходного файла, файла-результата, вспомогательного файла либо буфера оперативной памяти.

Закладки, перехватывающие уничтожаемые файлы, достаточно часто встречались в различных компьютерных системах. Как правило, закладки внедрялись в локальную сеть. Место сохранения «спасенных» файлов – сервер.

Закладки-сборщики уничтоженной информации, как правило, выявлялись на простом косвенном признаке – частое переполнение дисков, где размещалась область сохранения закладки (например, сервер).

Весьма интересна с точки зрения модели «перехват» технология аутентификации пользователей в защищенной сетевой операционной среде Windows NT. В момент старта системы появляется приглашение нажать сочетание клавиш Ctrl-Alt-Del. При нажатии клавиш системный процесс WinLogon, отвечающий за запрос паролей пользователей, загружает динамическую библиотеку MSGINA.DLL Данная библиотека отвечает за диалог с пользователем, ввод пароля, проверку его правильности (аутентификация) и передачу результата опознания пользователя и его данных (имени, имени домена) обратно в WinLogon. Функции данной библиотеки описаны в файле Winwlx.h, и существует штатный механизм замены исходной библиотеки на пользовательскую. Для этого необходимо добавить ключ-строку (string) GinaDLL в реестр \Registry\Machine\Software\Micro – softWVindows NT\CurrentVersion\Winlogon GinaDII = <путь к модифицированной библиотеке>.

Понятно, что присутствующая в системе закладка может искажать параметры каких-либо операций (мы не говорим здесь об ошибочных ситуациях). Можно выделить статическое и динамическое искажение. Статическое искажение заключается в изменении параметров программной среды, например редактирование файла AUTOEXEC.BAT так, чтобы первой запускалась заданная программа; изменение исполняемых модулей (редактирование кода или данных) с целью последующего выполнения нужных злоумышленнику действий. Динамическое искажение заключается в изменении каких-либо параметров процессов при помощи ранее активизированной закладки.

Статическое искажение, как правило, проводится один раз. Вообще говоря, разделить эти процессы трудно – так, внедрение

закладки по вирусному механизму в исполняемый файл сопряжено со статическим искажением исполняемого кода программы, а ее дальнейшая работа может быть связана с тем, что код закладки будет резидентным и станет влиять на файловые операции уже динамически.

Динамическое искажение условно можно разделить на искажение на входе (на обработку попадает уже искаженный документ) и искажение на выходе (искажается отображаемая для человека и прикладных программ информация).

Рассмотрим конкретную модель воздействия закладки для DOS. Закладка встраивается в цепочки прерывания int 21 для следующих функций и управляется следующими событиями.

- «Открыть файл». Закладка отфильтровывает нужные имена или указатели файлов.
- «Считать из файла». Закладка выполняет прерывание по старому адресу, затем сохраняет считанный буфер в собственном (обычно скрытом) файле либо исправляет в буфере некоторые байты; кроме того, возможно влияние на результаты операции чтения (например, на число прочитанных байтов). Данные действия особенно опасны для программ подтверждения подлинности электронных документов («электронная подпись»). При считывании приготовленного для подписи файла-документа может произойти изменение имени автора, даты, времени, цифровых данных, заголовка назначения документа (например, изменена сумма платежа в платежных поручениях и др.).
- «Записать в файл». Закладка редактирует нужным образом буфер в оперативной памяти или сохраняет файл или часть его в скрытой области, а затем выполняет старое прерывание – в результате записывается файл либо с измененным содержанием, либо каким-то образом дублированный в области сохранения. Закладки такого типа могут, на пример, навязывать истинность электронной подписи, даже если файл был изменен (если пометка об истинности подписи хранится в файле, что характерно для систем автоматизированного финансового документооборота).

Рассмотрим, например, процесс «электронного подписывания», реализованный в программе PGP. Программа считывает файл для вычисления хеш-функции блоками по 512 байт, причем завершением

процесса чтения является считывание блока меньшей длины. Работа закладки основана на навязывании длины файла. Закладка позволяет считать только первый 512-байтовый блок и вычисляет подпись только на его основе. Такая же схема действует и при проверке подписи. Следовательно, остальная часть файла может быть произвольным образом искажена.

Интересный пример воздействия на конечный результат проверки цифровой подписи – закладка, обнаруженная в финансовой системе с применением электронной цифровой подписи (ЭЦП) «Блиц». Работа закладки заключается в изменении на экране и в файле-журнале слов «НЕ ПОДТВЕРЖДАЕТСЯ» (при ошибочной подписи) на слово «ПОДТВЕРЖДАЕТСЯ» Активизирующим событием для искажения файла-журнала служит запись в него, а редактирование содержимого экрана выполнялось по таймерному прерыванию.

Практика применения ЭЦП в системах автоматизированного финансового документооборота показала, что именно программная реализация ЭЦП наиболее сильно подвержена влиянию со стороны программных закладок, которые позволяют осуществлять проводки заведомо фальшивых финансовых документов и вмешиваться в порядок разрешения споров по факту применения ЭЦП. Отметим четыре основных универсальных способа воздействия программных закладок на ЭЦП.

1. Способ искажения входной информации, связанный с искажением поступающего на подпись файла.

2. Способ искажения результата проверки, связанный с влиянием закладки на признак правильности подписи независимо от результата работы.

3. Способ изменения длины сообщения – предъявление программе ЭЦП электронного документа меньшей длины; следовательно, производится подпись только части документа.

4. Способ искажения программы ЭЦП, связанный с изменением исполняемого кода самой программы ЭЦП (изменение алгоритма хеширования и т.д.).

Остановимся на последнем подробнее, распространяя алгоритм искажения кода программы на произвольную прикладную программу. Предположим, что злоумышленнику известна интересующая его программа с точностью до команд реализации на конкретном процессоре. Следовательно, возможно смоделировать

процесс ее загрузки и выяснить относительные адреса частей программы относительно сегмента оперативной памяти, в который она загружается. Это означает, что возможно произвольное изменение кода программы и соответственно отклонение (может быть, негативного характера) в работе прикладных программ.

Тогда алгоритм действия закладки может быть таким:

- закладка загружается в память каким-либо образом;
- закладка осуществляет перехват (редактирование цепочки) одного или нескольких прерываний:
- прерывания (события) «запуск программ и загрузка оверлеев (библиотек)»;
- прерывания BIOS «считать сектор»;
- прерывание таймера.

По одному из трех событий закладка получает управление на свой код и далее выполняется:

- проверка принадлежности запущенной программы или уже работающей (для таймерного прерывания) к интересующим программам;
- определение сегмента, в который загружена программа;
- запись относительно определенного сегмента загрузки некоторых значений в ОП так, чтобы отключить процедуры контроля и/или исправить программу нужным образом.

Однако все же основным способом активизации разрушающих закладок является запуск ассоциированных с ними программ. При этом закладка получает управление первой и выполняет какие-либо действия (изменение адресов прерывания на собственные обработчики, исправление в коде прикладных программ и т.д.). При этом борьба с воздействием закладок может быть произведена только путем контроля целостности исполняемых файлов непосредственно перед их исполнением. В данном случае закладка ближе всего к классическому «вирусу».

Достаточно важно использование ошибок при влиянии закладок на работу прикладных программ. Инициирование ошибок в компьютерной системе является частным случаем искажения. Кратко рассмотрим этот вопрос. Ошибкой будем называть ситуацию, при которой завершение какого-либо события в программной среде происходит отличным от нормального (предписанного на этапе проектирования или программирования) образом. Будем рассматривать два потенциально возможных управляемых

злоумышленником процесса, связанных с ошибками, а именно инициирование и подавление ошибок. Возможны следующие четыре ситуации.

1. Инициирование статической ошибки – на устройствах хранения информации создается область, действия в которой (чтение, запись, форматирование и т.д.) приводят к возникновению ошибочной ситуации. Инициирование статической ошибки может затруднять или блокировать некоторые действия прикладных программ, например затрудняет корректное уничтожение ненужной информации.

2. Инициирование динамической ошибки – при выполнении некоторого процесса инициируется ошибочная ситуация из числа возможных ошибок данной операции. При этом заданное действие может не выполняться (в целях разрушительных воздействий – например, постоянно инициируется ситуация «модем занят», что приводит к блокированию приема или передачи) или же выполняться (т.е. реально ошибка не имеет места, а производится маскировка каких-либо действий закладки с тем, чтобы создать впечатление того, что они не произошли). Инициирование может быть более тонким, например в закладке против системы PGP достаточно выставить флаг переноса при прочтении второго блока (в примере он читается верно, но с нулевой длиной), чтобы достигнуть совершенно аналогичного эффекта «подделки» подписи.

3. Подавление статической ошибки – ситуация, обратная инициированию, т.е. на ошибочной области имитируется нормальная операция.

4. Подавление динамической ошибки используется для того, чтобы замаскировать ошибочное действие и «подставить» некоторый результат для прикладной программы. Целью подавления динамической ошибки может стать стремление вызвать эффект «снежного кома» – когда подавление ошибки в одном процессе приводит к блокированию работы всей компьютерной системы через некоторый промежуток времени. Данная ситуация крайне опасна для систем, в которых постоянно происходят какие-либо сходные действия (следовательно, легко прогнозировать результат подавления ошибки в одном звене). Зачастую подавление ошибки происходит «извне», когда в программном обеспечении имеется слабое место.

Для прогнозирования возможного влияния ошибок на прикладные программы используется, как правило, имитационное

или ситуационное моделирования комплекса прикладных средств и операционной среды.

3.5 Классификация и методы внедрения программных закладок

3.5.1 Основные классы программных закладок

Программные закладки, отключающие защитные функции системы. Во многих случаях программная закладка, внедренная в защищенную систему, может модифицировать машинный код или конфигурационные данные системы, тем самым полностью или частично отключая ее защитные функции. В защищенной системе создается «черный ход», позволяющий злоумышленнику работать в системе, обходя ее защитные функции.

Отключение программной закладкой защитных функций системы чаще всего используется для снятия защиты от несанкционированного копирования.

Перехватчики паролей. Перехватчики паролей перехватывают имена и пароли, вводимые пользователями защищенной системы в процессе идентификации и аутентификации. В простейшем случае перехваченные имена и пароли сохраняются в текстовом файле, более сложные программные закладки пересылают эту информацию по сети на компьютер злоумышленника.

Существуют три основные архитектуры построения перехватчиков паролей.

1. Перехватчики паролей первого рода действуют по следующему алгоритму. Злоумышленник запускает программу, которая имитирует приглашение пользователю для входа в систему, и ждет ввода. Когда пользователь вводит имя и пароль, закладка сохраняет их в доступном злоумышленнику месте, после чего завершает работу и осуществляет выход из системы пользователя-злоумышленника (в большинстве операционных систем выход пользователя из системы можно осуществить программно). По окончании работы закладки на экране появляется настоящее приглашение для входа пользователя в систему.

Пользователь, ставший жертвой закладки, видит, что он не вошел в систему и что ему снова предлагается ввести имя и пароль. Пользователь предполагает, что при вводе пароля произошла ошибка, и вводит имя и пароль повторно. После этого пользователь входит в

систему, и дальнейшая его работа протекает нормально. Некоторые закладки, функционирующие по данной схеме, перед завершением работы выдают на экран правдоподобное сообщение об ошибке, например: «Пароль введен неправильно. Попробуйте еще раз».

2. Перехватчики паролей второго рода перехватывают все данные, вводимые пользователем с клавиатуры. Простейшие программные закладки данного типа просто сбрасывают все эти данные на жесткий диск компьютера или в любое другое место, доступное злоумышленнику. Более совершенные закладки анализируют перехваченные данные и отсеивают информацию, заведомо не имеющую отношения к паролям.

Эти закладки представляют собой резидентные программы, перехватывающие одно или несколько прерываний, используемых при работе с клавиатурой. Информация о нажатой клавише и введенном символе, возвращаемая этими прерываниями, используется закладками для своих целей.

3. К перехватчикам паролей третьего рода относятся программные закладки, полностью или частично подменяющие собой подсистему аутентификации защищенной системы.

4. Программные закладки, превышающие полномочия пользователя. Эти закладки применяются для преодоления тех систем защиты, в которых реализовано разграничение доступа пользователей к объектам системы (в основном эти закладки применяются против операционных систем). Закладки данного типа позволяют пользователю-злоумышленнику осуществлять доступ к тем объектам, доступ к которым должен быть ему запрещен согласно текущей политике безопасности. В большинстве систем, поддерживающих разграничение доступа, существуют пользователи-администраторы, которые могут осуществлять доступ ко всем или почти всем объектам системы. Если программная закладка наделяет пользователя-злоумышленника полномочиями администратора, злоумышленник имеет практически неограниченный доступ к ресурсам системы.

Средства и методы, используемые такими закладками для превышения полномочий пользователя, в значительной мере определяются архитектурой атакуемой системы. Чаще всего закладки данного класса используют ошибки в программном обеспечении системы.

5. Логические бомбы. Это программные закладки, оказывающие при определенных условиях разрушающие воздействия на атакованную систему и обычно нацеленные на полное выведение системы из строя. В отличие от вирусов логические бомбы не размножаются или размножаются ограниченно. Логические бомбы всегда предназначены для конкретной компьютерной системы – объект атаки. После того как разрушающее воздействие завершено, логическая бомба уничтожается.

Иногда выделяют особый класс логических бомб – временные бомбы, для которых условием срабатывания является достижение определенного момента времени.

Характерным свойством логических бомб является то, что реализуемые ими негативные воздействия на атакованную систему носят исключительно разрушающий характер. Логические бомбы никогда не используются для организации НСД к ресурсам системы, их единственной задачей является полное или частичное разрушение системы.

6. Мониторы. Это программные закладки, перехватывающие те или иные потоки данных, протекающие в атакованной системе. В частности, к мониторам относятся перехватчики паролей второго рода.

Целевое назначение мониторов может быть самым разным:

полностью или частично сохранять перехваченную информацию в доступном злоумышленнику месте, искажать потоки данных;

- помещать в потоки данных навязанную информацию;
- полностью или частично блокировать потоки данных;
- использовать мониторинг потоков данных для сбора информации об атакованной системе.

Мониторы позволяют перехватывать самые различные информационные потоки атакуемой системы. Наиболее часто перехватываются следующие потоки:

- потоки данных, связанные с чтением, записью и другими операциями над файлами;
- сетевой трафик;
- потоки данных, связанные с удалением информации с дисков или из оперативной памяти компьютера (так называемая «сборка мусора»).

7. Сборщики информации об атакуемой среде. Программные закладки этого класса предназначены для пассивного наблюдения за

программной средой, в которую внедрена закладка. Основная цель применения подобных закладок заключается в первичном сборе информации об атакуемой системе. В дальнейшем эта информация используется при организации атаки системы, возможно, с применением программных закладок других классов.

3.5.2 Методы внедрения программных закладок

Маскировка закладки под «неопасное» программное обеспечение. Данный метод заключается в том, что программная закладка внедряется в систему под видом новой программы, на первый взгляд абсолютно безопасной. Программная закладка может представлять собой простой текстовый или графический редактор, системную утилиту, компьютерную игру, хранитель экрана и т.д. После внедрения закладки ее присутствие в системе не нужно маскировать – даже если администратор заметит факт появления в системе новой программы, он не придаст этому значения, поскольку эта программа внешне совершенно безобидна.

Если подобная программная закладка внедряется в многозадачную или многопользовательскую программную среду, ее возможности по оказанию на среду негативных воздействий сильно ограничены. Это обусловлено тем, что в подобных программных средах программы выполняются изолированно друг от друга и программа, как правило, не может оказывать негативные воздействия на другие программы среды и на всю среду в целом. Поэтому при внедрении закладки в многозадачную или многопользовательскую программную среду данный метод целесообразно применять только для внедрения инсталлирующей части составной программной закладки.

Маскировка закладки под «безопасный» модуль расширения программной среды. Многие программные среды допускают свое расширение дополнительными программными модулями. Например, для операционных систем семейства Microsoft Windows модулями расширения могут выступать динамически подгружаемые библиотеки (DLL) и драйверы устройств. В качестве одного или нескольких модулей расширения может быть внедрена в систему программная закладка. Данный метод фактически является частным случаем предыдущего метода и отличается от него только тем, что

закладка представляет собой не прикладную программу, а модуль расширения программной среды.

При внедрении программной закладки в многопользовательскую программную среду, поддерживающую разграничение доступа к объектам, данный метод более эффективен, чем предыдущий. Это объясняется тем, что модули расширения среды могут быть загружены не только пользователем, но и системными процессами программной среды. В этом случае модуль расширения получает существенно большие полномочия. Например, драйверы устройств операционной системы Microsoft Windows NT выполняются в привилегированном режиме, что позволяет им игнорировать разграничение доступа к объектам. В результате драйвер любого устройства Windows NT может обратиться к любому объекту операционной системы по любому методу доступа, в частности прочитать, изменить или удалить любой файл.

С другой стороны, для внедрения модуля расширения в многопользовательскую программную среду пользователь, выполняющий эти действия, должен обладать большими полномочиями, которые при соблюдении адекватной политики безопасности предоставляются только администраторам. Другими словами, пользователь, внедряющий программную закладку с использованием данного метода, должен иметь в системе полномочия администратора.

Подмена закладкой одного или нескольких программных модулей атакуемой среды. Данный метод внедрения в систему программной закладки заключается в том, что в атакуемой программной среде выбирается один или несколько программных модулей, подмена которых фрагментами программной закладки позволяет оказывать на среду требуемые негативные воздействия. Программная закладка должна полностью реализовывать все функции подменяемых программных модулей.

Основная проблема, возникающая при практической реализации данного метода, заключается в том, что программист, разрабатывающий программную закладку, никогда не может быть уверен, что созданная им закладка точно реализует все функции подменяемого программного модуля. Если подменяемый модуль достаточно велик по объему или недостаточно подробно документирован, точно запрограммировать все его функции практически невозможно. Поэтому описываемый метод целесо-

образно применять только для тех программных модулей атакуемой среды, для которых доступна полная или почти полная документация. Оптимальной является ситуация, когда доступен исходный текст подменяемого модуля.

При внедрении программной закладки в многозадачную или многопользовательскую среду с использованием данного метода возникают и другие проблемы, проявляющиеся на этапе подмены программного модуля закладкой.

Любая многозадачная программная среда обеспечивает корректность совместного доступа программ к данным. Корректность совместного доступа подразумевает, что программа не может открыть для записи объект среды, уже открытый другой программой (за исключением ситуации, когда это явно разрешено программой, открывшей объект первой). Поэтому в многозадачной среде невозможна подмена закладкой уже выполняющегося программного модуля – программный модуль является объектом среды, при загрузке программного модуля происходит открытие объ-екта. Для того чтобы подмена могла произойти, необходимо либо завершить работу подменяемого модуля и перезапустить его после подмены, либо изменить конфигурацию системы таким образом, чтобы после ее перезагрузки вместо подменяемого программного модуля была загружена программная закладка.

Если в многопользовательской среде реализовано разграничение доступа к объектам, то каждая программа запускается от имени того пользователя, который ее запустил. Для того чтобы программная закладка смогла оказать на систему серьезное негативное воздействие, она должна быть запущена либо от имени пользователя, обладающего в системе достаточно большими полномочиями (например, администратора), либо как системный процесс. Но если в системе поддерживается адекватная политика безопасности, исполняемые файлы системных программных модулей доступны для записи только администраторам. Следовательно, для внедрения в систему программной закладки с использованием данного метода необходимы полномочия администратора.

Прямое ассоциирование. Данный метод внедрения в систему программной закладки заключается в ассоциировании закладки с исполняемыми файлами одной или нескольких легальных программ системы. Сложность задачи прямого ассоциирования программной закладки с программой атакуемой среды существенно зависит от

того, является атакуемая среда однозадачной или многозадачной, однопользовательской или многопользовательской. Для однозадачных однопользовательских систем эта задача решается достаточно просто. В то же время при внедрении закладки в многозадачную или многопользовательскую программную среду прямое ассоциирование закладки с легальным программным обеспечением является весьма нетривиальной задачей.

Проблемы, возникающие при прямом ассоциировании программной закладки с программным модулем многозадачной или многопользовательской среды, в основном совпадают с проблемами, возникающими при подмене программного модуля такой среды программной закладкой. Прямое ассоциирование закладки с программным модулем предусматривает открытие закладкой для записи исполняемого файла программы, с которой происходит ассоциирование. Если в момент внедрения закладки программа выполняется или если атакуемая среда поддерживает разграничение доступа к своим объектам, при применении данного метода возникают те же самые проблемы, что и при применении предыдущего.

Поскольку при прямом ассоциировании закладки с атакуемой программой нарушается целостность исполняемого файла этой программы, программная закладка может быть легко выявлена с помощью контроля целостности исполняемых файлов программной среды.

При использовании для внедрения программной закладки метода прямого ассоциирования перед разработчиком закладки не стоит задача реализации всех функций программного модуля атакуемой среды, в который должна быть внедрена закладка. Поэтому этот метод в большинстве случаев более эффективен, чем метод подмены. Единственное исключение имеет место в случае, если атакуемый программный модуль хорошо документирован и задача реализации всех его функций решается просто.

Косвенное ассоциирование. Косвенное ассоциирование закладки с программным модулем атакуемой среды заключается в ассоциировании закладки с кодом программного модуля, загруженным в оперативную память. При косвенном ассоциировании исполняемый файл программного модуля остается неизменным, что затрудняет выявление программной закладки.

Для того чтобы косвенное ассоциирование стало возможным, необходимо, чтобы инсталлирующая часть закладки уже

присутствовала в системе. Другими словами, программная закладка, внедряемая в систему с помощью косвенного ассоциирования, должна быть составной.

При реализации косвенного ассоциирования в многозадачной программной среде инсталлирующая часть закладки должна получить доступ к коду или данным атакуемой программы. Поскольку в многозадачных программных средах программы выполняются изолированно друг от друга, косвенное ассоциирование возможно только в отдельных случаях. Например, в операционной системе Windows NT программная закладка может быть косвенно ассоциирована только с ядром операционной системы, HAL DLL или одним из boot-драйверов. При этом инсталлирующая часть закладки должна выполняться в привилегированном режиме (kernel mode).

Если атакуемая программа является оверлейной или в атакуемой программной среде реализована концепция виртуальной памяти, при косвенном ассоциировании закладки с программой возможна ситуация, когда модифицированные закладкой код или данные программы вытесняются из оперативной памяти. При последующей загрузке с диска этого участка кода или данных программы изменения, внесенные закладкой, будут потеряны. Это сильно ограничивает применение данного метода.

Поскольку при косвенном ассоциировании закладки с одной из программ среды не нарушается ни целостность кода, ни целостность данных программной среды, выявить такую закладку весьма сложно. Основным фактором, демаскирующим присутствие в системе такой закладки, является необходимость присутствия в системе инсталлирующей части закладки.

3.6 Компьютерные вирусы как особый класс разрушающих программных воздействий

3.6.1 Понятие компьютерного вируса

Впервые определение компьютерного вируса было сформулировано американским исследователем Ф. Коэном в 1984 году. Под компьютерным вирусом принято понимать программный код, обладающий следующими необходимыми свойствами:

1. способностью к созданию собственных копий, не обязательно совпадающих с оригиналом, но обладающих свойствами оригинала (самовоспроизведение);

2. наличием механизма, обеспечивающего внедрение создаваемых копий в исполняемые объекты вычислительной системы.

Следует обратить внимание на то, что эти свойства являются необходимыми, но не достаточными. В литературе обычно этот тезис иллюстрируется примером операционной системы, которая, являясь программой, обладает механизмом самораспространения, благодаря чему осуществляется установка этой системы на новые компьютеры.

По всей видимости, к этим свойствам необходимо добавить критерий, позволяющий определять, являются ли действия программы, удовлетворяющей свойствам 1 и 2, правомочными в данной вычислительной системе. Если вернуться к примеру с операционной системой, то следует отметить, что в этом случае самокопирование (т.е. установка) происходит всегда с санкции пользователя и под его контролем. Вирусы же в большинстве своем распространяются скрытно и несанкционированно.

Как правило, изучение множества каких-либо объектов начинается с построения системы классификаций, позволяющих обобщать и дифференцировать их признаки и свойства. Наиболее распространенной, а, по мнению многих, основной классификацией компьютерных вирусов является классификация по типам объектов вычислительной системы, в которые они внедряются. В настоящее время выделяются три типа объектов.

1. Программные файлы операционных систем. Вирусы, поражающие эти объекты, называются файловыми.

2. Системные области компьютеров, в частности области начальной загрузки операционных систем. Соответствующие вирусы получили название загрузочных или бутовых.

3. Макропрограммы и файлы документов современных систем обработки информации, например Microsoft Word. Вирусы, связанные с этим типом объектов, именуются макровирусами.

Существует и комбинированный тип – файлово-загрузочные вирусы. В некоторых работах по вирусной проблематике выделяется еще один класс вирусов, распространяющихся через вычислительные сети.

Анализ же основных этапов «жизненного цикла» этой группы вредоносных программ позволит выделить их различные признаки и

особенности, которые могут быть положены в основу дополнительных классификаций.

3.6.2 Жизненный цикл вирусов

Как и у любой программы, у компьютерных вирусов можно выделить две основные стадии жизненного цикла – хранение и исполнение. Стадия хранения соответствует периоду, когда вирус просто хранится на диске совместно с объектом, в который он внедрен. На этой стадии вирус является наиболее уязвимым со стороны антивирусного программного обеспечения, так как он не активен и не может контролировать работу операционной системы с целью самозащиты.

Некоторые вирусы на этой стадии используют механизмы защиты своего кода от обнаружения. Наиболее распространенным способом защиты является шифрование большей части тела вируса. Его использование совместно с механизмами мутаций кода (об этом идет речь ниже) делает невозможным выделение устойчивых характеристических фрагментов кода вирусов – сигнатур. Это, в свою очередь, приводит к недееспособности антивирусных средств, основанных на методах поиска заранее выявленных сигнатур.

Стадия исполнения компьютерных вирусов, как правило, состоит из пяти этапов:

1. загрузка вируса в память;
2. поиск жертвы;
3. заражение найденной жертвы;
4. выполнение деструктивных функций;
5. передача управления программе-носителю вируса.

Рассмотрим эти этапы подробнее.

1. Загрузка вируса. Загрузка вируса в память осуществляется операционной системой одновременно с загрузкой исполняемого объекта, в который вирус внедрен. Например, если пользователь запустил на исполнение программный файл, содержащий вирус, то, очевидно, вирусный код будет загружен в память как часть этого файла. В простейшем случае процесс загрузки вируса представляет собой не что иное, как копирование с диска в оперативную память, сопровождаемое иногда настройкой адресов, после чего происходит передача управления коду тела вируса. Эти действия выполняются операционной системой, а сам вирус находится в пассивном

состоянии. В более сложных ситуациях вирусы могут после получения управления выполнять дополнительные действия, которые необходимы для его функционирования. В связи с этим рассматриваются два аспекта.

Первый из них связан с тем, что некоторые вирусы, как отмечалось выше, используют для самозащиты в период хранения механизмы криптографической защиты. В этом случае дополнительные действия, которые выполняет вирус на этапе загрузки, состоят в расшифровке основного тела вируса.

Второй аспект связан с так называемыми резидентными вирусами. Так как вирус и объект, в который он внедрен, для операционной системы являются единым целым, то, естественно, после загрузки они располагаются в едином адресном пространстве. Следовательно, после завершения работы объекта он выгружается из оперативной памяти и при этом выгружается вирус, переходя в пассивную стадию хранения. Однако некоторые типы вирусов способны сохраняться в памяти и оставаться активными после окончания работы вирусоносителя. Эти вирусы получили название резидентных.

Такие вирусы на стадии загрузки должны позаботиться о «закреплении» своего кода в оперативной памяти. Это можно реализовать различными способами. Например, операционные системы типа MS-DOS содержат стандартные средства поддержки резидентных модулей. Но, как правило, вирусы не пользуются этими механизмами, а переносят свой код либо в самостоятельно отведенные блоки памяти, либо в зарезервированные под нужды ОС участки памяти. Однако размещение кода в оперативной памяти – необходимое, но не достаточное действие. Помимо этого вирус должен позаботиться о том, чтобы этому коду время от времени передавалось управление. Ведь логика работы программ и операционной системы этого не предполагает. Поэтому вирусы должны изменить код системных функций, которые гарантированно используются прикладными программами, добавив в них команды передачи управления своему коду, либо изменить в системной таблице адреса соответствующих системных функций, подставив адреса своих подпрограмм. Такой перехват определенных функций, например чтения/записи файлов, позволит также вирусу контролировать информационные потоки между операционной системой и прикладными программами.

Итак, резидентность, применительно к MS-DOS предполагает выполнение вирусом на этапе загрузки двух действий – «закрепление» в памяти и перехват системных функций. Для многозадачных систем типа Windows вирусы могут обойтись без перехвата системных событий, если не ставится задача контроля информационных потоков. Для того чтобы вирус мог функционировать независимо от программы-носителя, ему достаточно зарегистрировать в системе новую задачу, в контексте которой он будет выполняться.

Перехват системных функций с целью контроля действий ОС является обязательным для так называемых стелс-вирусов (Stealth). Эти вирусы способны скрывать свое присутствие в системе и избегать обнаружения антивирусными программами. К примеру, подобные вирусы могут перехватывать системные функции чтения файла для того, чтобы в случае обращения к зараженному файлу эмулировать его «чистоту», временно восстанавливая первозданный вид.

В заключение следует отметить, что деление вирусов на резидентные и нерезидентные справедливо в основном для файловых вирусов. Загрузочные вирусы, как правило, являются резидентными. Что касается макровирусов, то их также можно считать резидентными, так как для большинства из них выполняются основные требования – постоянное присутствие в течение всего времени работы управляющей среды и перехват функций, используемых при работе с документами.

2. Поиск жертвы. По способу поиска жертвы вирусы можно разделить на два класса. К первому относятся вирусы, осуществляющие активный поиск, с использованием функций операционной системы. Примером являются файловые вирусы, использующие механизм поиска исполняемых файлов в текущем каталоге. Второй класс составляют вирусы, Реализующие пассивный механизм поиска, т.е. вирусы, расставляющие «ловушки» для программных файлов. Как правило, файловые вирусы устраивают подобные ловушки путем перехвата функции Exec операционной системы, а макровирусы – с помощью перехвата команд типа Save as из меню File.

3. Заражение жертвы. В простейшем случае заражение представляет собой самокопирование кода вируса в выбранный в качестве жертвы объект. Классификация вирусов на этом этапе

связана с анализом особенностей этого копирования, а также способов модификации заражаемых объектов.

Рассмотрим сначала особенности файловых вирусов. По способу инфицирования жертвы вирусы можно разделить на два класса. К первому относятся вирусы, которые не внедряют свой код непосредственно в программный файл, а изменяют имя файла и создают под старым именем новый, содержащий тело вируса. Второй класс составляют вирусы, внедряющиеся непосредственно в файлы-жертвы. Они характеризуются местом внедрения. Возможны следующие варианты:

- Внедрение в начало файла. Этот способ является наиболее удобным для COM-файлов MS-DOS, так как данный формат не предусматривает наличие служебных заголовков. При внедрении данным способом вирусы могут либо производить конкатенацию собственного кода и кода программы-жертвы, либо переписывать начальный фрагмент файла в конец, освобождая место для себя.
- Внедрение в конец файла. Наиболее распространенный тип внедрения. Передача управления коду вирусов обеспечивается модификацией первых команд программы (COM) или заголовка файла (EXE).
- Внедрение в середину файла. Как правило, этот способ используется вирусами применительно к файлам с заранее известной структурой (например, к файлу COMMAND.COM) или же к файлам, содержащим последовательность байтов с одинаковыми значениями, длина которой достаточна для размещения вируса. Во втором случае вирусы архивируют найденную последовательность и замещают собственным кодом. Помимо этого вирусы могут внедряться в середину файла, освобождая себе место путем переноса фрагментов кода программы в конец файла или же «раздвигая» файл. Различные схемы заражения представлены на рисунке 10.

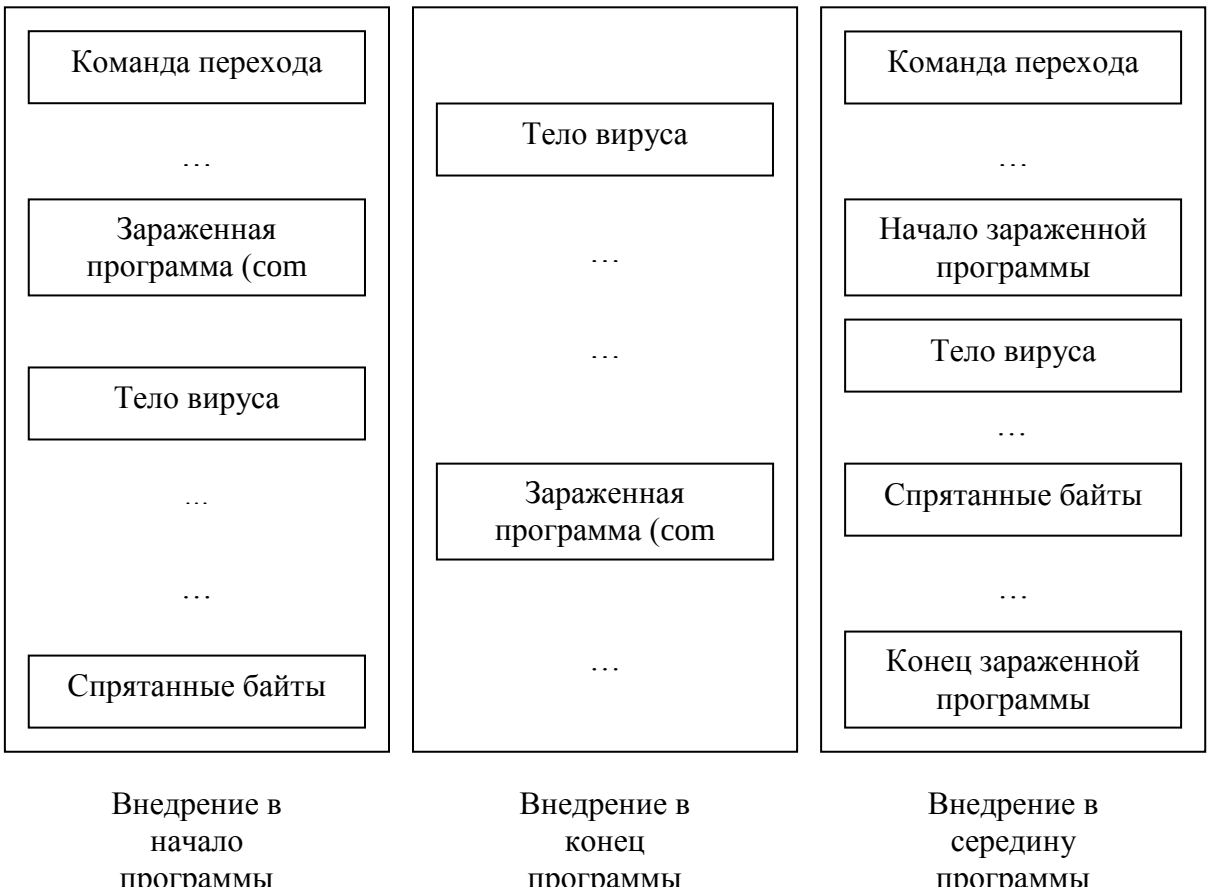

Рисунок 10 – Схема заражения com файла

Особенности этапа заражения для загрузочных вирусов определяются особенностями объектов, в которые они внедряются, – загрузочными секторами гибких и жестких дисков и главной загрузочной записью (MBR) жестких дисков. Основной проблемой является ограниченный размер этих объектов. В связи с этим вирусам необходимо каким-то образом сохранить где-то на диске ту свою часть, которая не уместилась на месте жертвы, а также перенести оригинальный код инфицированного загрузчика. Существуют различные способы решения этой задачи. Ниже приводится классификация, предложенная Е. Касперским.

- используются псевдосбойные сектора. Вирус переносит необходимый код в свободные сектора диска и помечает их как сбойные, защищая тем самым себя и загрузчик от перезаписи;
- используются редко применяемые сектора в конце раздела. Вирус
  переносит необходимый код в эти свободные сектора в конце

диска. С точки зрения операционной системы эти сектора выглядят как свободные;

- используются зарезервированные области разделов. Вирус переносит необходимый код в области диска, зарезервированные под нужды операционной системы, а потому – неиспользуемые;
- короткие вирусы могут уместиться в один сектор загрузчика и полностью взять на себя функции MBR или загрузочного сектора.

Процесс заражения для макровирусов сводится к сохранению вирусного макрокода в выбранном документе-жертве. Для некоторых систем обработки информации это сделать не совсем просто, так как формат файлов документов может не предусматривать возможность сохранения макропрограмм. В качестве примера приведем Microsoft Word. Сохранение макрокода для этой системы возможно только в файлах шаблонов (имеющих по умолчанию расширение .DOT). Поэтому для своего сохранения вирус должен контролировать обработку команды Save as из меню File, которая вызывается всякий раз, когда происходит первое сохранение документа на диск. Этот контроль необходим, чтобы в момент сохранения изменить тип файла-документа (имеющего по умолчанию расширение .DOC) на тип файла-шаблона. В этом случае на диске окажутся и макрокод вируса, и содержимое документа.

Помимо простого копирования кода вируса в заражаемый объект на этом этапе могут использоваться более сложные алгоритмы, обеспечивающие защиту вируса на стадии хранения. К числу таких усложнений можно отнести уже упоминавшееся шифрование основного тела вируса. Однако использование только шифрования является полумерой, так как в открытом виде должна храниться та часть вируса, которая обеспечивает расшифровку вируса на стадии загрузки. Для избежания подобных ситуаций разработчики вирусов могут использовать механизмы «мутаций» кода расшифровщика. Суть этого метода состоит в том, что при внедрении в объект копии вируса часть ее кода, относящаяся к расшифровщику, модифицируется так, чтобы возникли текстуальные различия с оригиналом, но результаты работы остались неизменными. Наиболее распространенными приемами модификации кода являются следующие:

– изменение порядка независимых инструкций;

– замена некоторых инструкций на эквивалентные по результату работы;

– замена используемых в инструкциях регистров на другие;

– внедрение случайным образом зашумляющих инструкций.

Вирусы, использующие подобные механизмы мутации кода, получили название полиморфных. В результате совместного использования механизмов мутации и шифрования внедряемая копия окажется отличной от оригинала, так как одна ее часть будет изменена, а другая окажется зашифрованной на ключе, сгенерированном специально для этой копии вируса. А это существенно осложняет выявление вируса в вычислительной системе.

4. Выполнение деструктивных функций. Вирусы могут выполнять помимо самокопирования деструктивные функции. Согласно классификации Касперского вирусы делятся на «безвредные», «неопасные», «опасные», «очень опасные». Безвредные вирусы – это вирусы, в которых реализован только механизм самораспространения. Они не наносят вред системе, за исключением расхода памяти. Неопасные вирусы – это вирусы, присутствие которых в системе связано с различными эффектами (звуковыми, видео), но которые не наносят вред программам и данным. Опасные вирусы – это вирусы, которые могут стать причиной сбоя системы. Разрушение программ и данных может стать последствием сбоя. Очень опасные вирусы это вирусы, непосредственно приводящие к разрушениям программ и данных.

На «степень опасности» вирусов оказывает существенное влияния та среда, под управлением которой вирусы работают. Так, вирусы, созданные для работы в MS-DOS, обладают практически неограниченными потенциальными возможностями. Распространение вирусов под управлением Windows NT ограничивается развитой системой разграничения доступа.

Возможности макровирусов напрямую определяются возможностями макроязыков, на которых они написаны. К примеру, макроязык справочной системы Windows Help настолько «слаб», с точки зрения разработчиков вирусов, что даже реализация механизмов самокопирования вызывает определенные трудности. С другой стороны, язык Word Basic является благодатной почвой для создания мощных макровирусов, способных доставить серьезные неприятности пользователям.

Дополняя эту классификацию, можно отметить деление вирусов на вирусы, наносящие вред системе вообще, и вирусы, предназначенные для целенаправленных атак на определенные объекты.

5. Передача управления программе-носителю вируса. Здесь следует указать на деление вирусов на разрушающие и неразрушающие. Разрушающие вирусы не заботятся о том, чтобы при инфицировании программ сохранять их работоспособность, поэтому для них этот этап функционирования отсутствует. Для неразрушающих вирусов этот этап связан с восстановлением в памяти программы в том виде, в котором она должна корректно исполняться, и передачей управления программе-носителю вируса.

3.6.3 Общие вопросы борьбы с компьютерными вирусами

Для борьбы с компьютерными вирусами в настоящее время используются различные средства, которые можно разделить на три класса:

- административные;
- юридические;
- технические;

Административные средства, как правило, включают комплекс мер, действующих в рамках предприятий и направленных на снижение ущерба, наносимого компьютерными вирусами. В качестве примеров можно привести программы проведения профилактических мероприятий, планы действия сотрудников в случае, если их компьютер подвергся вирусной атаке, запреты на самостоятельную установку нового программного обеспечения и т.п.

Юридические средства сводятся к привлечению к уголовной (или административной) ответственности лиц, по чьей вине наносится ущерб вычислительным системам. В настоящее, время законодательства многих стран имеют разделы, посвященные компьютерным преступлениям, к числу которых относится распространение компьютерных вирусов. В частности, в Уголовном кодексе Российской Федерации имеется статья 273, в которой предусмотрена ответственность «за создание, использование и распространение вредоносных программ для ЭВМ».

Технические средства разделяются на программные и аппаратные. Под первой группой средств обычно понимают

программы, применяемые для предупреждения заражения и выявления факта заражения. К аппаратным средствам относятся различные устройства, позволяющие контролировать обращения к данным жесткого диска, проверять при загрузке операционной системы состояние загрузочного сектора и т.п.

3.7 Понятие изолированной программной среды

Задача защиты от разрушающих программных воздействий может ставиться в нескольких принципиально различных вариантах. Первый, в какой-то степени классический, вариант – выявление и уничтожение («лечение»). Он берет начало в задачах выявления и ликвидации вирусов. Ослабленная разновидность данного варианта – только выявление вируса. Понятно, что алгоритмы нахождения и «лечения» вирусов отличаются некоторой принципиальной ненадежностью – возможен пропуск вируса во время процедуры проверки при условии, что вирус в компьютерной системе присутствует (ошибка первого рода). С другой стороны, выше мы говорили о множестве путей внедрения РПВ и их разнообразном негативном воздействии. Следовательно, для защиты сколько-нибудь серьезной системы необходимо полностью исключить наличие РПВ (или допустить присутствие РПВ с вероятностью не выше заданной и очень малой).

Итак, формализуем общую задачу борьбы с разрушающими программными воздействиями, рассмотрев следующие начальные условия.

1. Априорно неизвестно наличие в каком-либо множестве программ
фрагментов РПВ. Ставится задача определения факта их наличия или отсутствия; при этом программы не выполняются (статическая задача).

2. При условиях, рассматриваемых в п.1, прикладные программы используются по своему прямому назначению. Также ставится задача выявления закладки, но в данном случае динамическая (по результатам работы).

3. Происходит обмен программным продуктом (либо в пространстве -
передача по каналу связи или пересылка на магнитном носителе, либо во времени – хранение), априорно свободным от потенциально

опасных действий. Программный продукт не исполняется. Задача защиты (статическая) ставится в трех вариантах:

- не допустить внедрение закладки;
- выявить внедренный код закладки;
- удалить внедренный код закладки.
- при условиях п.3 решается динамическая задача – защита от воздействия закладки (закладок) в ходе работы программ.
- при условии потенциальной возможности воздействия закладок

  решается задача борьбы с их итоговым влиянием, т.е. закладки присутствуют в системе, но либо не активны при выполнении критических действий прикладных программ, либо результат их воздействия не конструктивен.

Далее рассмотренные задачи будем упоминать как задачи 1-5.

Методы борьбы с воздействием закладок можно разделить на классы и увязать с общей проблемой защиты программного обеспечения от несанкционированного доступа:

1.Общие методы защиты программного обеспечения, решающие задачи борьбы со случайными сбоями оборудования и несанкционированным доступом.

- Контроль целостности системных областей, запускаемых прикладных программ и используемых данных (решение задачи 3).
- Контроль критических для безопасности системы событий (решение задачи 2).

Данные методы действенны лишь тогда, когда контрольные элементы не подвержены воздействию закладок и разрушающее воздействие либо инициирующее его событие входят в контролируемый класс. Так, система контроля за вызовом прерываний не будет отслеживать обращение к устройствам на уровне портов. С другой стороны, контроль событий может быть обойден путем навязывания конечного результата проверок, влияния на процесс считывания информации, изменения контрольных элементов (хеш-функций), хранящихся в общедоступных файлах или в оперативной памяти.

Важно, что контроль должен быть выполнен до начала влияния закладки либо контроль должен осуществляться полностью аппаратными средствами с программами управления, содержащимися в ПЗУ.

– Создание безопасной и изолированной операционной среды (решение задачи 4).
– Предотвращение результирующего воздействия вируса или закладки (например, запись на диск только в зашифрованном виде на уровне контроллера – тем самым локальное сохранение информации закладкой не имеет смысла – или запрет записи на диск на аппаратном уровне) (решение задачи 5).

2. Специальные методы выявления программ с потенциально опасными последствиями.

– Поиск фрагментов кода по характерным последовательностям (сигнатурам), свойственным закладкам, либо, наоборот, разрешение на выполнение или внедрение в цепочку прерываний только программам с известными сигнатурами (решение задач 1,2).
– Поиск критических участков кода (с точки зрения безопасности компьютерной системы) методом семантического анализа. При этом анализ фрагментов кода на выполняемые ими функции, например выполнение НСЗ, часто сопряжен с дизассемблированием или эмуляцией выполнения (решение задач 1,2).

Рассмотрим процесс создания защищенного фрагмента компьютерной системы в применении к проблеме защиты от РПВ. Первоначально необходимо убедиться, что в программном обеспечении ПЗУ вычислительных средств системы (например, в BIOS ПЭВМ) не имеется РПВ. Данная задача может решаться в статическом варианте (задача 1) и динамическом варианте (задача 2). С точки зрения экономико-временных параметров целесообразнее решение задачи 1, поскольку в противном случае требуется длительная работа в аппаратной среде в различных режимах. На практике желательно комплексно решать как первую, так и вторую задачу. Описанный этап назовем **шагом 1.**

Далее следует определить состав программных средств базовой вычислительной среды, т.е. определить конкретную операционную среду, дополнительные программные средства сервиса (например, программные оболочки или средства телекоммуникации) и программные средства поддержки дополнительного оборудования (программы управления принтером и др.). После этого наступает самый трудоемкий этап (**шаг 2**), на котором необходимо убедиться в отсутствии РПВ в описанном базовом наборе программных средств.

При этом заметим следующее. В составе ПО базовой вычислительной среды не должно быть целого класса возможностей – назовем их инструментальными. Прежде всего, это возможность вмешательства оператора в содержимое оперативной памяти (запись), возможность инициирования и прекращения выполнения процессов нестандартным образом (помимо механизмов операционной среды).

Обобщенно достаточные условия к базовому набору ПО можно сформулировать следующим утверждением. В составе ПО, которое может быть инициировано в компьютерной системе, не должно быть функций порождения и прекращения выполнения процессов, кроме заранее определенных, и не должно быть возможностей влияния на среду выполнения уже активных процессов и на сами эти процессы.

Шаг 3 заключается в проектировании и разработке программных или программно-аппаратных средств защиты компьютерной системы, а затем и в их тестировании.

Шаг 4 состоит в «замыкании» всего комплекса программного обеспечения, включая и средства защиты, в изолированную программную среду. Подробности механизма реализации этой среды будут даны ниже, сейчас отметим, что далее предполагается неизменность состава полученного в ходе выполнения шагов 1-3 программного продукта.

Очевидно, что при пустом множестве активизирующих событий для закладки потенциальные деструктивные действия с ее стороны невозможны.

Положим, что в BIOS и операционной системе отсутствуют закладки. Пусть пользователь работает с программой, процесс написания и отладки которой полностью контролируются, т.е. в ней также исключено наличие закладок или каких-либо скрытых возможностей (проверенная программа).

Откуда потенциально исходит угроза для такой системы?

1. Проверенные программы будут использованы на другом компьютере с другой BIOS, которая может содержать закладки.

2. Проверенные программы будут использованы в аналогичной, но не проверенной операционной среде, в которой могут содержаться закладки.

3. Проверенные программы используются на проверенном компьютере и в проверенной операционной среде, но запускаются еще и не проверенные программы, потенциально несущие в себе

закладки.

Следовательно, деструктивные действия закладок гарантированно невозможны, если:

1. на компьютере с проверенной BIOS установлена проверенная операционная среда;
2. достоверно установлена неизменность операционной среды и BIOS для данного сеанса работы;
3. кроме проверенных программ в данной программно-аппаратной среде не запускалось и не запускается никаких иных программ;
4. исключен запуск проверенных программ в какой-либо иной ситуации, т.е. вне проверенной среды.

При выполнении перечисленных условий программная среда называется изолированной.

Итак, мы видим, что основными элементами поддержания изолированности среды являются контроль целостности и контроль активности процессов. При этом для алгоритмов контроля целостности важно выполнение следующих условий:

1. надежный алгоритм контроля;
2. контроль реальных данных.

Поясним подробнее второй пункт. Контроль целостности всегда сопряжен с чтением данных (по секторам, по файлам и т.д.). Например, закладка в BIOS может навязывать при чтении вместо одного сектора другой или редактировать непосредственно буфер. С другой стороны, даже контроль BIOS может происходить «под наблюдением» какой-либо дополнительной аппаратуры и не показать ее изменение. Аналогичные эффекты могут возникать и при обработке файла. А.Петровым предложена модель «безопасной загрузки» или ступенчатого контроля. Она заключается в по-степенном установлении неизменности компонентов программно-аппаратной среды: сначала проверяется неизменность BIOS, при положительном исходе через проверенную BIOS считываются загрузочный сектор и драйверы операционной среды (по секторам) и их неизменность также проверяется; через проверенные функции операционной среды загружается драйвер контроля вызовов программ.

При рассмотрении проблемы защищенности информации при ее обработке в компьютерной системе необходимо обращать внимание на наличие скрытых возможностей в базовом ПО. Скрытые

возможности сами по себе или в сочетании с другими программами из базового ПО могут привести к опосредованному НСД. Для обеспечения безопасности обрабатываемой информации и всего информационного процесса (технологии) в целом необходимо в базовом ПО предусмотреть:

1. Невозможность запуска никаких иных программ, кроме входящих в состав базового ПО;
2. Невозможность повлиять на среду функционирования и сами программы, уже выполняемые в компьютерной системе;
3. Невозможность изменить любые программы базового ПО. Наиболее просто было бы выполнить данные условия (условия 1-3) в тех случаях, когда все базовое ПО находится в ПЗУ и ПЗУ не содержит более никаких программ и фрагментов кода. Инициирование программ происходит при включении питания. Однако в таком случае система будет представлять собой нечто похожее на микрокалькулятор.

На практике в компьютерной системе работают несколько пользователей (субъектов), каждый из которых использует некоторое подмножество программ базового ПО (эти подмножества могут иметь непустое пересечение). Кроме того, они, как правило, имеют возможность запускать другие программы. Практически в любом компьютере имеется возможность влиять на среду функционирования программ (в частности, текстовый редактор вполне может использоваться для коррекции кода; далее, если не поставить пользователю ограничений, всегда имеется возможность использования отладчиков).

При этом выполнить условия 1-3 становится практически невозможно. Поясним это примером. Текстовый редактор является неотъемлемым инструментом любой прикладной программной системы. Из любого редактора возможен запуск задачи (программы) либо напрямую через операционную среду, либо косвенно, минуя ее (второе встречается реже). Так, для DOS можно использовать команды типа DOS SHELL, подгружая еще одну копию COMMAND.COM. Даже если администратор компьютерной системы исключит из состава ПО все программы, которые кажутся ему подозрительными, то пользователь все равно будет иметь возможность выхода в операционную среду (и выполнения ее команд, в том числе уничтожение информации) и потенциальной активизации принесенных на ГМД программных средств (в том

числе загружая собственную операционную среду). Если же принять во внимание еще и скрытые возможности в ПО, то говорить о защищенности информации в системе невозможно.

Рассмотрим функционирование программ в изолированной программной среде (ИПС). Тогда требования к базовому ПО существенно ослабляются. В самом деле, ИПС контролирует активизацию процессов через операционную среду, контролирует целостность исполняемых модулей перед их запуском и разрешает инициирование процесса только при одновременном выполнении двух условий – принадлежности к разрешенным программам и неизменности программ. В таком случае от базового ПО требуется:

1. невозможность запуска программ в обход контролируемых ИПС событий;

2. отсутствие возможностей влиять на среду функционирования уже запущенных программ (фактически это требование невозможности редактирования оперативной памяти).

Все прочие действия, являющиеся нарушением условий 1-3 в оставшейся их части, будут выявляться и блокироваться. Таким образом, ИПС существенно снижает трудозатраты на анализ ПО на наличие скрытых возможностей.

При включении питания компьютера происходит тестирование ОП, инициализация таблицы прерываний и поиск расширений BIOS. При наличии расширений управление передается им. После отработки расширений BIOS в память считывается первый сектор дискеты или винчестера и управление передается ему, код загрузчика считывает драйверы, далее выполняются файлы конфигурации, подгружается командный интерпретатор и выполняется файл автозапуска.

При реализации ИПС на нее должны быть возложены функции контроля запуска программ и контроля целостности. При описании методологии создания ИПС упоминалась проблема контроля реальных данных. Эта проблема состоит в том, что контролируемая на целостность информация может представляться по-разному на разных уровнях.

Если программный модуль, обслуживающий процесс чтения данных, не содержал РПВ и целостность его зафиксирована, то при его последующей неизменности чтение с использованием его будет чтением реальных данных. Из данного утверждения логически вытекает способ ступенчатого контроля. Предварительно

фиксируется неизменность программ в основном и расширенных BIOS, далее с помощью функции чтения в BIOS (для DOS int 13h) читаются программы обслуживания чтения (драйверы DOS), рассматриваемые как последовательность секторов, и фиксируется их целостность, затем, используя файловые операции, читают необходимые для контроля исполняемые модули (командный интерпретатор, драйверы дополнительных устройств, .EXE и .COM модули и т.д.).

При запуске ИПС таким же образом и в той же последовательности происходит контроль целостности. Этот алгоритм можно обобщить на произвольную операционную среду. Для контроля данных на i-м логическом уровне их представления для чтения требуются предварительно проверенные на целостность процедуры (*i*-1)-го уровня. Самым же первым этапом является контроль целостности программ в ПЗУ (этап 0).

Обратимся теперь к вопросу контроля целостности данных. Предположим, что имеется некоторый файл F – последовательность байтов и некоторый алгоритм A, преобразующий файл F в некоторый файл M (последовательность байтов) меньшей длины. Этот алгоритм таков, что при случайном равновероятном выборе двух файлов из множества возможных соответствующие им числа M с высокой вероятностью различны. Тогда проверка целостности данных строится так: рассматриваем файл F, по известному алгоритму A строим K = A(F) и сравниваем M, заранее вычисленное как M = A(F), с K. При совпадении считаем файл неизменным. Алгоритм A называют, как правило, хеш-функцией или реже – контрольной суммой, а число M – хеш-значением. В данном случае чрезвычайно важно выполнение следующих условий:

– по известному числу M=A(F) очень трудоемким должно быть нахождение другого файла G, не равного F, такого, что M=A(G);

– число M должно быть недоступно для изменения.

Поясним смысл этих условий. Пусть программа злоумышленника изменила файл F (статическое искажение). Тогда, вообще говоря, хеш-значение M для данного файла изменится.

Если программе злоумышленника доступно число M, то она может по известному алгоритму A вычислить новое хеш-значение для измененного файла и заместить им исходное. С другой стороны, пусть хеш-значение недоступно, тогда можно попытаться так построить измененный файл, чтобы хеш-значение его не изменилось

(принципиальная возможность этого имеется, поскольку отображение, задаваемое алгоритмом хеширования, не биективно (неоднозначно). Выбор хорошего хеш-алгоритма, так же как и построение качественного шифра, крайне сложная задача.

4 Защита информации от изучения

4.1 Классификация способов защиты

При проектировании и реализации систем защиты информации (СЗИ) практически всегда возникает задача защиты ее алгоритма. Атака на алгоритм СЗИ может быть пассивной (изучение) – без изменения алгоритма СЗИ и активной – с изменением алгоритма в соответствии с целью злоумышленника. Например, при анализе СЗИ, выполняющей операции криптографической защиты, успешная пассивная атака приводит к тому, что злоумышленнику становится полностью известен алгоритм шифрования и, возможно, часть ключевых элементов, локализованных в программе. При активной атаке на СЗИ от копирования возможно отключение механизмов проверки ключевого признака.

Будем считать, что человек, преследующий цель исследования логики работы СЗИ, является злоумышленником. Для решения своей задачи он располагает средством (субъектом компьютерной системы), которое для краткости будем называть отладчиком.

Задача защиты от изучения (исследования) сводится к решению двух взаимосвязанных задач:

• выявлению отладчика;

• направлению работы защищенной программы или отладчика по «неверному пути».

Атаки злоумышленника на логику работы СЗИ сводятся к декомпозиции программного кода на множество отдельных команд (инструкций), каждая из которых (либо группа инструкций) рассматривается отдельно. Условно такие атаки можно разделить на динамические – разбор по командам, сопряженный с выполнением программы (выполнением каждой рассматриваемой команды), и статические – декомпозиция всего кода программы, возможно, не сопряженная с ее прямым исполнением, основанная на анализе логики исполнения команд и логики вызовов различных функций (дизассемблирование). Не вдаваясь в подробности обоих процессов, отметим лишь существенно важные детали.

Программное средство злоумышленника (отладчик) работает в общей для него и изучаемой программы оперативной памяти, использует те же аппаратные ресурсы (например, процессор), непустое общее подмножество внешних событий программной среды (например, прерываний) – иначе говоря, работа отладчика оставляет в

программно-аппаратной среде, общей с изучаемой программой, некоторые «следы». Кроме того, отладчик управляется человеком, который, как известно, реагирует на все гораздо медленнее, чем ЭВМ. Далее, управление работой отладчика может происходить через клавиатуру, а восприятие результатов – через экран.

Вполне понятно, что изучаемая программа может «мешать» работе отладчика, имея для этого все названные выше признаки. В данном случае представляется уместной аналогия из квантовой механики, когда наблюдатель либо существенно влияет на процесс в мире микрочастиц, либо должен быть так мал, что частицы будут влиять на него. И в том, и в другом случае реальная картина эксперимента искажается.

Злоумышленник должен наблюдать за изучаемой программой как можно «незаметнее», не оставлять следов своей деятельности, и его программное средство должно быть защищено от активного воздействия со стороны изучаемой программы. С другой стороны, разработчик защиты должен так строить ее, чтобы либо обнаруживать присутствие отладчика, либо не давать работать вообще никому, кроме себя, либо мешать восприятию результатов исследования.

4.2 Защита от отладки и дизассемблирования

Способы защиты от исследования можно разделить на четыре класса.

1. Влияние на работу отладочного средства через общие программные или аппаратные ресурсы. В данном случае наиболее известны:

- использование аппаратных особенностей микропроцессора (особенности работы очереди выборки команд, особенности реализации команд и т.д.);
- использование общего программного ресурса (например, общего стека) с отладочным средством и разрушение данных или кода отладчика, принадлежащих общему ресурсу, либо проверка использования общего ресурса только защищаемой программой (например, определение стека в области, критичной для выполнения защищаемой программы);
- переадресация обработчиков отладочных событий (прерываний) от отладочного средства к защищаемой программе.

Выделение трех групп защитных действий в данном классе не случайно, поскольку объективно существуют общие аппаратные ресурсы (и отладчик, и защищаемая программа в случае однопроцессорного вычислителя выполняются на одном и том же процессоре), общие программные ресурсы (поскольку и отладчик, и защищаемая программа выполняются в одной и той же операционной среде), наконец, отладчик создает специфичные ресурсы, существенные для его собственной работы (например, адресует себе отладочные прерывания).

2. Влияние на работу отладочного средства путем использования особенностей его аппаратной или программной среды. Например:

- перемещение фрагментов кода или данных с помощью контроллера прямого доступа к памяти;
- влияние на процесс регенерации оперативной памяти (на некотором участке кода регенерация памяти отключается, а затем опять включается – при нормальной работе никаких изменений нет, при медленном выполнении программы отладчиком она «зависает»);
- переход микропроцессора в защищенный режим.

3. Влияние на работу отладчика через органы управления или/и устройства отображения информации.

Выдаваемая отладочными средствами информации анализируется человеком. Следовательно, дополнительный способ защиты от отладки -это нарушение процесса общения оператора и отладчика, а именно искажение или блокирование вводимой с клавиатуры и выводимой на терминал информации.

4. Использование принципиальных особенностей работы управляемого человеком отладчика. В данном случае защита от исследования состоит в навязывании для анализа избыточно большого объема кода (как правило, за счет циклического исполнения некоторого его участка).

Рассмотрим данный метод подробнее. Пусть имеется некоторое полноцикловое преобразование из N состояний $t_i : t_1 \dots t_N$ (например, обычный двоичный счетчик либо рекуррента). Значение N выбирается не слишком большим. Например, для k-битового счетчика $N = 2^k$. Участок кода, защищаемый от изучения, динамически преобразуется (шифруется) с использованием криптографически стойкого алгоритма на ключе t_i который выбирается случайно равновероятно

из множества состояний T.

Работа механизма защиты от исследования выглядит следующим образом. Программа полноциклового преобразования начинает работу с детерминированного или случайного значения. На установленном значении производится расшифрование зашифрованного участка кода. Правильность расшифрования проверяется подсчетом хеш-значения расшифрованного кода с использованием элементов, связанных с отладкой (стек, отладочные прерывания и др.). Хеш-функция должна с вероятностью 100% определять правильность расшифровки (для этого хеш-значение должно быть не менее k).

Предположим, что полноцикловое преобразование стартует с первого значения. Тогда при нормальном выполнении программы (скорость работы высокая) будет совершено i циклов алгоритма, после чего защищенный участок будет корректно исполнен. При работе отладчика, управляемого человеком, скорость выполнения программы на несколько порядков ниже, поэтому для достижения необходимого значения i будет затрачено значительное время.

Для численной оценки данного метода введем следующие значения. Предположим, что i в среднем равно $M/2$. Пусть w_0 – время выполнения цикла алгоритма (установка текущего значения, расшифровка, проверка правильности расшифровки) в штатном режиме функционирования (без отладки); w_1 – время выполнения того же цикла в режиме отладки; z -предельное время задержки при штатной работе защищенной программы. Тогда $N = z/w_0$. Затраты времени злоумышленника исчисляются средней величиной $T_{зл} = Nw_1/2$. Для приблизительных расчетов $w_1/w_0 = 10000$. (Идея метода принадлежит А. Долгину.)

В ряде способов защиты от отладки идентификация отладчика и направление его по ложному пути происходят одновременно, в одном и том же фрагменте кода (так, при определении стека в области кода защищаемой программы при работе отладчика, использующего тот же стек, код программы будет разрушен). В других случаях ложный путь в работе программы формируется искусственно. Часто для этого используют динамическое преобразование программы (шифрование) во время ее исполнения.

Способ динамического преобразования заключается в следующем:

первоначально в оперативную память загружается фрагмент кода, содержание части команд которого не соответствует тем командам, которые данный фрагмент в действительности выполняет; затем этот фрагмент по некоторому закону преобразуется, превращаясь в исполняемые команды, которые затем и выполняются.

Преобразование кода программы во время ее выполнения может преследовать три основные цели:

- противодействие файловому дизассемблированию программы;
- противодействие работе отладчику;
- противодействие считыванию кода программы в файл из оперативной памяти.

Перечислим основные способы организации преобразования кода программы:

1. Замещение фрагмента кода функцией от находящейся на данном месте команде и некоторых данных.

2. Определение стека в области кода и перемещение фрагментов кода с использованием стековых команд.

3. Преобразование кода в зависимости от содержания предыдущего фрагмента кода или некоторых условий, полученных при работе предыдущего фрагмента.

4. Преобразование кода в зависимости от внешней к программе информации.

5. Преобразование кода, совмещенное с действиями, характерными для работы отладочных средств.

Поясним кратко приведенные способы. Первый способ заключается в том, что по некоторому адресу в коде программы располагается, например, побайтовая разность между реальными командами программы и некоторой хаотической информацией, которая располагается в области данных. Непосредственно перед выполнением данного участка программы происходит суммирование хаотической информации с содержанием области кода и в ней образуются реальные команды.

Второй способ состоит в перемещении фрагментов кода программы в определенное место или наложении их на уже выполненные команды при помощи стековых операций.

Третий способ служит для защиты от модификации кода программы и определения точек останова в программе. Он состоит в том, что преобразование следующего фрагмента кода происходит на основе функции, существенно зависящей от каждого байта или слова

предыдущего фрагмента или нескольких фрагментов кода программы. Такую функцию называют обычно контрольной суммой участка кода программы. Особенностью данного способа является то, что процесс преобразования должен существенно зависеть от посчитанной контрольной суммы (хеш-функции) и не должен содержать в явном виде операций сравнения.

Четвертый способ заключается в преобразовании кода программы на основе некоторой внешней информации, например считанной с ключевой дискеты некопируемой метки, машинно-зависимой информации или ключа пользователя. Это позволит исключить анализ программы, не имеющей ключевого носителя или размещенной на другом компьютере, где машинно-зависимая информация иная.

Пятый способ состоит в том, что вместо адресов отладочных прерываний помещается ссылка на процедуру преобразования кода программы. При этом либо блокируется работа отладчика, либо неверно преобразуется в исполняемые команды код программы.

Часто с отладочными прерываниями бывают совмещены сложные алгоритмы преобразования кода программы, которые при выполнении каждой команды производят преобразование последующей команды в исполняемую и предыдущей в хаотический код. Такой метод получил название «бегущая строка». Как правило, это прерывание int 1h (ONE Step). Для успешного применения данного метода обязательно выполнение следующих условий:

- невозможность переопределения функции, замещающей отладочное прерывание и выполняющей преобразование кода программы, на неиспользуемое прерывание;
- отсутствие операций сравнения для определения факта работы под отладчиком.

Выполнить первое условие возможно, просчитывая контрольные суммы (хеш-функции) пройденного участка кода либо вводя в функциях, замещающих отладочные прерывания, зависимость от положения их в таблице прерываний.

Если нельзя избежать операции сравнения (например, чтобы нелегальная копия программы не «зависала»), то желательно сравнивать не контрольные числовые значения, а некоторые необратимые или сложно-обратимые функции от них.

Важной задачей является защита от трассировки программы по заданному событию (своего рода выборочное исследование). В

качестве защиты от трассировки по заданному событию (прерыванию) можно выделить ряд основных способов.

1. Пассивная защита – запрещение работы при переопределении обработчиков событий относительно заранее известного адреса.
2. Активная защита первого типа – замыкание цепочек обработки событий минуя программы трассировки.
3. Активная защита второго типа – программирование функций, исполняемых обработчиками событий, другими способами, не связанными с вызовом «штатных» обработчиков или обработчиков событий, которые в текущий момент не трассируются.

Например, для защиты от трассировки по дисковым прерываниям для 008 при чтении некопируемой метки с дискеты или винчестера можно использовать следующие приемы:

- работа с ключевой меткой путем прямого программирования контроллера гибкого диска (активная защита второго типа);
- определение одного из неиспользуемых прерываний для работы с диском (активная защита первого типа);
- прямой вызов соответствующих функций в ПЗУ (BIOS) после восстановления различными способами их физического адреса (активная защита первого типа);
- определение факта переопределения адреса прерывания на другую программу и невыполнение в этом случае дисковых операций (пассивная защита).

При операциях с жестким диском, как правило, используется прерывание *int* 13*h*. Для предотвращения трассировки программы по заданному прерыванию (в данном случае прерыванию *int* 13*h*) можно также использовать указанные выше способы, а именно:

- переопределение исходного прерывания в BIOS на неиспользуемый вектор прерывания;
- прямой вызов функций BIOS.

Прерывание *int* 1*h* относится к аппаратным прерываниям, вызываемым особым состоянием процессора. Данное состояние достигается установкой специального флага трассировки в регистре флагов. При этом после завершения выполнения каждой инструкции процессора автоматически вызывается прерывание *int* 1*h* (режим покомандного управления). Для того чтобы включение трассировки не воздействовало на саму программу-обработчик первого прерывания (в режиме обычной работы она состоит из одной

команды *iret*), при вызове прерывания флаг трассировки автоматически сбрасывается. Поскольку при вызове прерывания регистр флагов автоматически сохраняется в стеке, а при окончании работы обработчик его автоматически восстанавливает, то после обработки прерывания управление возвращается к следующей трассируемой команде вновь с установленным флагом трассировки. Функционально это аналогично тому, что после каждой команды прикладной программы находится команда вызова первого прерывания. Выполнив покомандно цепочку *int* 13*h* и проанализировав исполняемые инструкции, можно легко определить точки входа в BIOS.

Часто недооценивается опасность трассировки по событиям операционной среды. В основном таким образом могут быть выделены следующие действия программ:

- определение факта замены обработчиков событий на собственные функции (в частности, для защиты от отладчиков);
- файловые операции, связанные со считываниями различных счетчиков или паролей, вычисление контрольных сумм и хеш-функций файлов;
- файловые операции, связанные со считыванием заголовков и другой существенно важной информации в исполняемых файлах или загружаемых библиотеках.

Кроме того, в защищаемых программах, полученных при помощи распространенных компиляторов, в самом начале идет последовательность предопределенных событий операционной среды (для программ DOS – прерываний 21*h*). Этот факт позволяет в очень многих случаях при остановке на одном из этих прерываний считать код программы из оперативной памяти в незащищенном виде, поскольку к этому моменту антиотладочные средства обычно завершают свою работу.

Для того чтобы СЗИ выполняла предписанные функции, необходимо интегрировать защитные механизмы в исполняемый код защищаемого программного обеспечения. Например, для реализации системы защиты от копирования необходимо принимать решение о том, является ли данная копия программного обеспечения легальной или нет. Это решение принимается на основе анализа некопируемой информации (метки), а процесс приема решения в программе защищается от изучения так, как было описано выше. Встраивание

защитных механизмов можно выполнить следующими основными способами:

вставкой фрагмента проверочного кода (одного или нескольких) в исполняемый файл;

- преобразованием исполняемого файла к неисполняемому виду (шифрование, архивация с неизвестным параметром и т.д.) и применением для загрузки не средств операционной среды, а некоторой программы, в теле которой и осуществляются необходимые проверки;
- вставкой проверочного механизма в исходный код на этапе разработки и отладки программного продукта (для проектируемых и проверяемых СЗИ является основным);
- комбинированием указанных способов.

Применительно к конкретной реализации защитных механизмов для конкретной аппаратно-программной архитектуры можно говорить о защитном фрагменте (в исполняемом или исходном коде). К процессу и результату встраивания защитных механизмов можно предъявить следующие естественные требования:

- высокая трудоемкость обнаружения защитного фрагмента при статическом исследовании (особенно актуальна при встраивании в исходный код программного продукта);
- высокая трудоемкость обнаружения защитного фрагмента при динамическом исследовании (отладке или трассировке по внешним событиям);
- высокая трудоемкость обхода или редуцирования защитного фрагмента.

Возможность встраивания защитных фрагментов в исполняемый код обусловлена типовой архитектурой исполняемых модулей различных операционных сред, содержащих, как правило, адрес точки входа в исполняемый модуль. В этом случае добавление защитного фрагмента происходит следующим образом. Защитный фрагмент добавляется к началу или концу исполняемого файла, точка входа корректируется таким образом, чтобы при загрузке управление предавалось дополненному защитному фрагменту, в составе защитного фрагмента предусматривается процедура возврата к оригинальной точке входа. Достаточно часто оригинальный исполняемый файл подвергается преобразованию. В этом случае перед возвратом управления оригинальной точке входа производится

преобразование образа оперативной памяти загруженного исполняемого файла к исходному виду.

В случае дополнения динамических библиотек возможна коррекция указанным образом отдельных функций.

Существенным недостатком рассмотренного метода является его легкая обнаружимость и в случае отсутствия преобразования оригинального кода исполняемого файла – легкость обхода защитного фрагмента путем восстановления оригинальной точки входа.

5 Анализ программных средств

5.1 Поэтапная схема анализа исполняемого кода

Задачу изучения программы в общем случае можно сформулировать следующим образом. Имеется бинарный код программы (например, exe-файл) и минимальная информация о том, что эта программа делает. Нужно получить более детальную информацию о функционировании этой программы. Другими словами, известно, что делает программа, и необходимо 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 анализ. Но мы будем рассматривать наиболее общий (и наиболее сложный) случай, когда аналитик знает только код программы.

Термин «программа» здесь и далее будем понимать в широком смысле, подразумевая под ней не только исполняемый файл, но и библиотеку функций, драйвер устройства и т.д. Отметим, что изложенная далее методика применима не только к одиночным программам, но и к программным комплексам, содержащим более одной программы.

Работа по анализу программы состоит из трех основных этапов.

1. Подготовительный этап. На этом этапе аналитик знакомится с анализируемой программой, изучает доступную документацию, планирует дальнейшие исследования, подбирает коллектив и организует его работу. Важность этого этапа нельзя недооценивать. Эффективность дальнейшей работы очень сильно зависит от того, насколько полная информация об анализируемой программе была получена на первом этапе. Если аналитик сумел получить исходный текст анализируемой программы, задача анализа программы в большинстве случаев решается тривиально. Наличие у аналитика отладочной информации об анализируемой программе также существенно упрощает дальнейшую работу.

2. Восстановление алгоритмов функционирования программы. Именно на этом этапе производится изучение программы. (Методам, применяемым на этом этапе, посвящена большая часть данной главы.)

3. Проверка полученных результатов. На этом этапе аналитик проверяет результаты проведенных исследований. 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, а если же отличается, это означает, что в анализе программы допущены ошибки, которые необходимо устранить. Чаще всего восстановить анализируемые алгоритмы с первого раза не удается.

В настоящее время сформировались три подхода к восстановлению алгоритмов, реализуемых программой:

1) метод экспериментов;
2) статический метод;
3) динамический метод.

При анализе программ на любом языке обязательно присутствуют три этапа:

1. лексический анализ;
2. синтаксический анализ;
3. семантический анализ.

Первые два из них обычно осуществляются автоматически (например, компиляторами), а семантику отслеживает человек. В этой главе мы попытаемся применить эту схему и к анализу программ в исполняемом коде, хотя точной аналогии между понятиями, существующими в теории компиляторов и вводимыми здесь, провести нельзя.

Каждый из этих этапов также требует предварительной подготовки:

1. выделение чистого кода – удаление кода, отвечающего за защиту этой программы от несанкционированного запуска, копирования и т.п. и преобразование остального кода в стандартный, правильно интерпретируемый дизассемблером. Как частные случаи, это может быть разархивация, расшифровка кода;
2. дизассемблирование – перевод команд на язык ассемблера;
3. перевод в форму, удобную для следующего этапа – выделение основных процедур, циклов и др. управляющих конструкций, основных структур данных, а также, возможно, перевод на язык более высокого уровня.

Таким образом, общая поэтапная схема анализа программ будет выглядеть так (Рисунок 11):

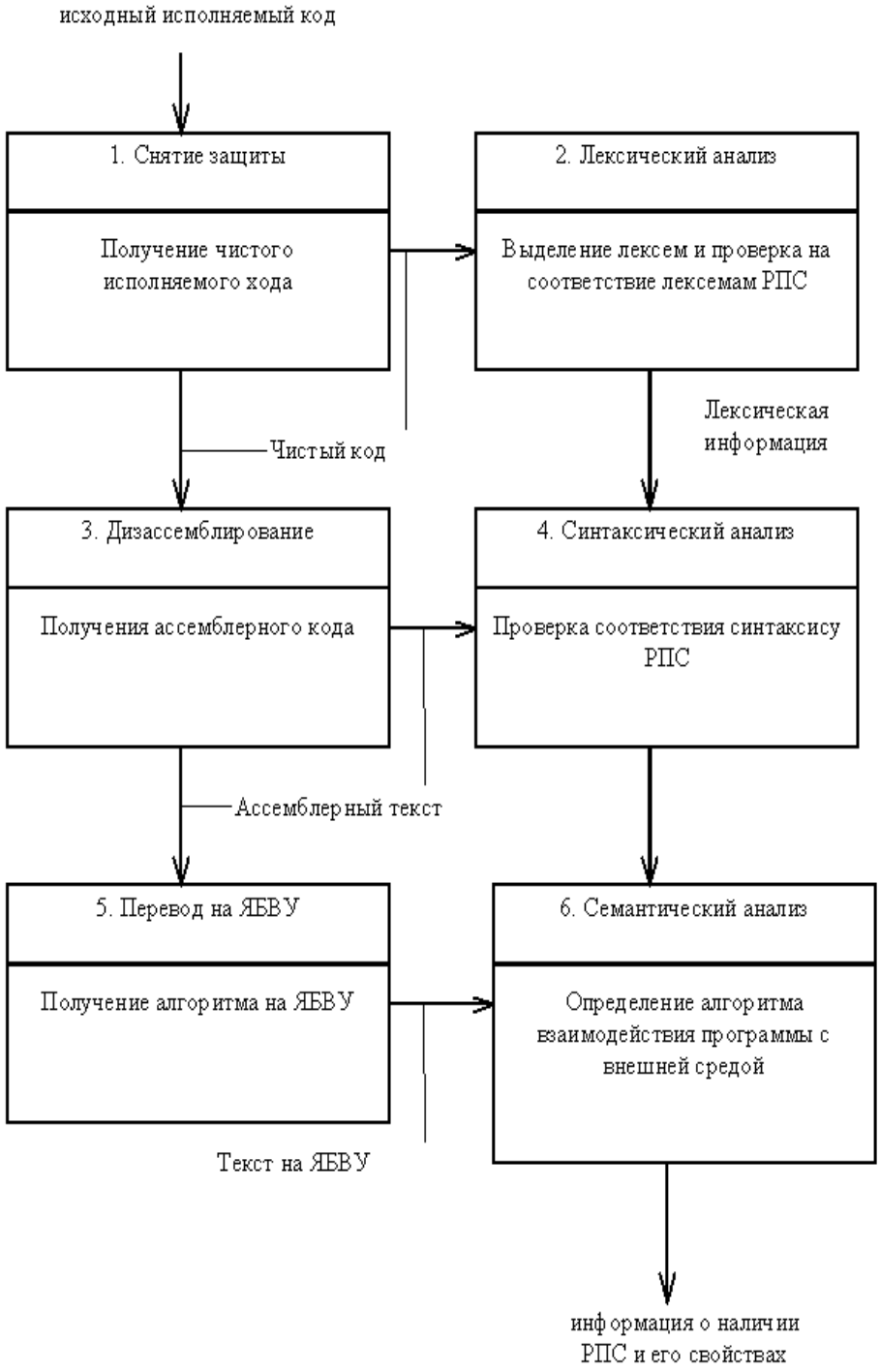

Рисунок 11 – Поэтапная схема анализа исполняемого кода

Задачей первого этапа анализа исполняемого кода программ является подготовка его к этапу лексического анализа и дизассемблирования. Можно назвать две основные причины, по

которым исполняемый код может быть затруднен для дизассемблирования. Это:

1. специально предпринятые меры для противодействия исследованию этого кода. Такой код будем называть защищенным, а алгоритмы, их реализующие – системами собственной защиты программ.
2. меры, направленные на преобразование кода в другую форму и не преследующие задачи противодействия. Чаще всего это архивация или шифрация исходного кода. (Шифрация может применяться и с полезными целями: например, преобразование кода так, чтобы он состоял только из печатаемых символов – это необходимо для пересылки его по каналам электронной почты). Для простоты такие алгоритмы тоже будем называть системами защиты.

Таким образом, преобразованный код будет состоять из процедуры защиты (дешифровки) и защищенной (зашифрованной) части, а алгоритмам, отвечающим за восстановление чистого кода, необходимо проделать обратные действия:

- удалить систему защиты;
- преобразовать оставшийся код в стандартный.

Здесь мы сталкиваемся с той же проблемой: чтобы удалить систему защиты, надо понять, как она работает, т.е. провести семантический анализ ее алгоритма.

Однако, если ввести ограничение (справедливое для большинства защит и упаковщиков программ), что система защиты не включается в программу на этапе ее разработки, а внедряется в уже готовый исполняемый файл (также, как и вирусы), то анализа системы защиты возможно избежать и автоматизировать восстановление исходного кода с помощью т.н. «универсальных распаковщиков». Это следует из того, что в этом случае система защиты отрабатывается сразу, при любом наборе входных данных. И после ее отработки можно снять дамп памяти, содержащий уже расшифрованный код.

Здесь возникает всего две проблемы:

- как определить, что система защиты уже отработала, т.е. найти начало «чистого» кода. Оказывается, что это не так трудно сделать, в частности, начальный код, генерируемый любым компилятором, стандартен и известен.
- определить размер снимаемого дампа (в случае архивации он

может быть даже больше, чем начальный размер программы). Здесь тоже существуют некоторые эвристические методы, но в крайнем случае ничто не мешает взять некоторый максимальный размер.

Для системы MS DOS известно много «универсальных распаковщиков» (UUP, Intruder, AutoHack), позволяющих не просто получить код в виде дампа памяти, а обычный нормально запускающийся EXE-файл. Не вдаваясь в подробности, заметим, что для генерации EXE-файла необходимо еще и определить, какие данные относятся к таблице перемещения и восстановить ее. Это достигается тем, что исходная программа запускается два раза с разных адресов и данные, корректируемые таблицей перемещения, в двух разных дампах оказываются неодинаковыми.

Если же система защиты не является такой простой, то в этом случае на помощь могут прийти трассировщики. Они записывают каждую инструкцию, проходящую через процессор (т.е. содержимое ячейки, на которую указывает IP) в специальную область памяти или файл, совершенно «не вникая», путем каких преобразований была получена эта инструкция, запоминая таким об разом список действительно исполняемых команд, трассу (trace).Очевидность этого метода следует из того, что любая система защиты должна восстановить исходный код по крайней мере в момент считывания его центральным процессором. Однако трассировщики применимы только тогда, когда система защиты не включает и динамическую защиту. В противном случае она может распознать изменения, вносимые трассировщиком в вычислительную среду.

Получаемый трассировщиком код не очень похож на компилированный, однако в нем можно автоматически выделить процедуры, циклы и т.п., сократив его в десятки раз. Это можно делать как сразу, по мере получения трассы, так и потом, работая уже с готовой.

Если же в результате этого этапа не удалось получить стандартный код, это уже может служить основанием для отнесения входной программы к подозрительным, т.к. она использует весьма сложную систему защиты, что подавляющему большинству полезных программ явно не нужно.

Лексический анализ – это процесс распознавания транслятором во входном тексте смысловых единиц (лексем): идентификаторов, чисел, операторов и т.п. По аналогии, лексическим анализом при

исследовании исходного кода программ будем называть процесс поиска и распознавания лексем, которыми в данном случае будут являться определенные регулярные последовательности, называемыми сигнатурами РПС.

Определение сигнатуры РПС отличается от определения сигнатуры вируса и звучит как: «Сигнатурой РПС называется любая регулярная последовательность байт, встречающаяся в исполняемом коде РПС». Таким образом, сигнатуры вирусов – это частный случай сигнатур РПС.

Функцией лексического анализатора, несколько выпадающей из общей схемы анализа, является т.н. «экспресс-анализ» – поиск сигнатур известных РПС, нахождение которых однозначно свидетельствует, что программа содержит «вредный» код (т.е. той или иной РПС), и в этом случае не требуется дальнейшего исследования.

До сих пор именно лексический анализ являлся основным и очень эффективным средством поиска вирусов. Это связано с весьма большим быстродействием и простотой в реализации, позволяющей также исправлять и наращивать базу искомых сигнатур. Именно эти характеристики позволяют применять лексический анализ в качестве начального этапа анализа исполняемого кода – «экспресс-анализа».

Рассмотрим подробнее, какого вида регулярные последовательности можно применять для этом этапе для наиболее полного и эффективного поиска разных классов РПС.

Самым простым их видом являются последовательности байт определенной длины или т.н. простые байтовые сигнатуры. Они могут записываться в десятичном, шестнадцатиричном, двоичном или символьном виде, а алгоритмы их поиска эффективны и широко описаны.

Однако не все вирусы и другие РПС имеют такие сигнатуры. В частности, многие вирусы часто вставляют в свой код некоторое переменное число байтов, чтобы довести размер зараженной программы до числа, кратного 16 (округление по границе параграфа). Далее, в теле вируса могут быть неизвестные заранее последовательности байт определенной длины, попадающие в него из заражаемой программы (например, первые байты из начала заражаемой программы).

Наличие таких фрагментов в разных местах тела вируса, с одной стороны, резко уменьшает длину максимальной байтовой сигнатуры

и, с другой, увеличивает число таких сигнатур, каждую из которых можно использовать для поиска вируса. Конечно, для осуществления поиска любых вирусов, имеющих сигнатуру, в принципе можно было бы использовать и такие короткие простые байтовые сигнатуры, но для увеличения длины сигнатуры и, соответственно, существенного повышения качества диагностирования, вводятся так называемые регулярные сигнатуры.

Эти более сложные сигнатуры можно искать, применяя широко известные регулярные выражения, отсюда их название. Эти сигнатуры являются расширениями простых байтовых сигнатур с помощью двух специальных символов: '*' и '?'. Первый означает любое количество (0 и больше) байт, а второй строго один байт. Таким образом, сигнатура

EB ? ? FF * FF

может означать

EB 10 00 FF FF

EB FF FF FF FF FF и т.д.

Мысль создателей РПС с точки зрения их маскировки пошла дальше. Сначала появились вирусы, самошифрующиеся с переменным ключом. Очевидно, что для такого вируса в качестве сигнатуры может выступать только фрагмент незашифрованного кода, т.е. расшифровщик, который обычно представляет собой очень короткую последовательность байт, что ухудшает качество их диагностирования. Но потом появились вирусы, вообще не содержащие ни простых, ни регулярных байтовых сигнатур – так называемые вирусы с самомодифицирующимся расшифровщиком. Это вирусы, использующие помимо шифрования кода, специальную процедуру расшифровки, изменяющую саму себя в каждом новом экземпляре вируса. Достигается это за счет того, что один и тот же алгоритм можно реализовать большим количеством способов на конкретном языке. Например, в машинных кодах (т.е. в исполняемом коде) можно:

1. добавить к нему ничего не значащие команды;
2. поменять используемые регистры;
3. переставить команды местами.

Для поиска таких вирусов пришлось применить новый тип сигнатур: битовые регулярные сигнатуры. Первый способ изменения кода легко обходится использованием уже известного символа '*', который имеет здесь тот же смысл, что и выше: 0 или более байт. (Отметим, что именно байт, а не бит, т.к. вряд ли существует машинный язык,

на котором команды имели бы переменное число бит). Второй способ – чередование используемых регистров – описывается введением битового символа '?', означающего битовый 0 или 1. Дело в том, что регистр, используемый в команде, кодируется либо в каком-то конкретном байте этой команды (и тогда можно применять более эффективный символ '*'), либо в конкретных битах команды, а остальные биты остаются постоянными. (Для MS DOS справедлив второй случай). Третий способ – чередование команд местами – может быть обойден введением асинхронных выражений, выделяемых символами начала '{' и конца '}'. Все группы байт, попадающих в эти выражения, могут быть переставлены местами во входной последовательности. Регулярная битовая сигнатура и ей соответствующая последовательность представлены на рисунке 12.

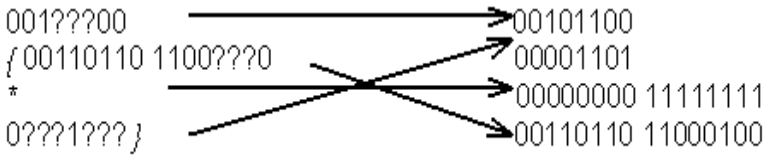

Рисунок 12 – Пример регулярной битовой сигнатуры

Однако в 1992 г. появились вирусы, для поиска которых непригодны даже регулярные битовые сигнатуры! Это все те же вирусы с самомодифицирующимся расшифровщиком, но использующие очень совершенный алгоритм такой самомодификации, называемый MtE (Mutant Engine). В нем, помимо приведенных выше способов мутации, применяется еще и четвертый:

4. использование других команд (xor ax, ax вместо mov ax 0; push/ret вместо jmp и т.д.).

Расшифровщик, сгенерированный по этому алгоритму, не имеет ни одного постоянного бита, а используется в нем более половины всех инструкций процессора 8086. Таким образом, лексический поиск для таких вирусов просто неприменим.

Вместо регулярных битовых сигнатур гораздо более удобно использовать регулярные ассемблерные сигнатуры, являющиеся их расширением. Они избавляют пользователя от необходимости перевода вручную самомодифицирующегося кода в регулярные битовые последовательности. Их элементами являются ассемблерные команды и расширенные регулярными выражениями, состоящие из префикса и регулярного символа. В качестве префикса могут

выступать: x, l, h – означают регистр, его младший и старший байт соответственно;

s – сегментный регистр;

c – команда;

o – операнд;

a – адрес;

без префикса – все команды.

К регулярным символам добавились, помимо '*' и '?', цифры, означающие конкретный объект. Например, x1 означает любой (но один и тот же в данном регулярном выражении) регистр.

Символ '*' применяется только к целой команде и не имеет префиксов.

Приведем пример такой сигнатуры:

```
mov x1, ax cmp x1, 4B00h je 070
```

Эта конструкция означает присваивание в любой регистр регистра AX, а затем сравнение именно этого регистра с числом 4B00h и переход по конкретному относительному адресу.

Макрокоманда

MOV ah, 25h (записывается большими буквами)

означает любую последовательность команд, приводящую в конечном счете к присваиванию в регистр AH числа 25h, причем манипуляции с другими регистрами, памятью и т.п. здесь нас не интересуют. Ясно, что подходящих регулярных последовательностей команд может быть бесконечно много, например:

```
1.  mov ah, 25H
2.  mov ax, 2513H
3.  mov h1, 25H
*
xchg ah, h1
4.  mov ah, 20h
*
add ah, 5h
5.  push 2513
pop ax
```

Однако здесь, также как и при реализации транслятора, необходимо найти разумный компромисс между лексическим и синтаксическим анализом и не перекладывать задачи первого на второй и наоборот. Поэтому поиск макрокоманд вряд ли необходим в

лексическом анализаторе. (По крайней мере, без средств динамического исследования он не может быть реализован).

Дизассемблированием называется процесс перевода программы из исполняемых или объектных кодов на язык ассемблера.

Задача дизассемблирования практически решена и не будет поэтому здесь рассматриваться. Для MS DOS существует много хороших дизассемблеров, почти стопроцентно справляющихся со стандартным кодом. (Дизассемблированный текст может считаться 100% правильным, если при повторном ассемблировании получается исходный код. Аналогично, код считается 100% стандартным, если дизассемблер получает 100% правильный текст).

С помощью этапа дизассемблирования поэтому можно проверить качество выполнения этапа получения чистого кода если дизассемблер не генерирует 100% правильный код, то тот этап был не закончен.

В системе анализа исполняемого кода программ однако удобнее применять не стандартные дизассемблеры, а специализированный (блок 8.2 рис. 8), который, помимо ассемблерного текста, может выдавать и другую полезную информацию. Самой важной для следующего этапа будет информация о передаче управления в программе, т.е. последовательность вызова процедур.

Синтаксическим анализом при проектировании компиляторов называют процесс отождествления лексем, найденных во входной цепочке, одной из языковых конструкций, задаваемых грамматикой языка. Иначе его можно рассматривать как процесс построения дерева грамматического разбора. По аналогии будем говорить, что синтаксический анализ исполняемого кода программ состоит в отождествлении сигнатур, найденных на этапе лексического анализа, одному из видов РПС.

Для синтаксического анализа любого языка необходимо иметь его грамматику, описывающую, по каким правилам из лексем (терминальных символов или терминалов) строятся его предложения. Т.е., применительно к РПС, мы должны формализовать «грамматику» РПС, описывающую, по каким правилам из лексем (сигнатур РПС) строятся «предложения» (т.е. различные типы РПС). Это можно сделать с помощью любой формы записи грамматики, например БНФ. Но т.к. грамматика является очень простой и не содержит, в частности, рекурсии, то нагляднее всего отобразить ее в виде дерева,

которое будем назвать грамматическим деревом или деревом свертки (рисунок 13):

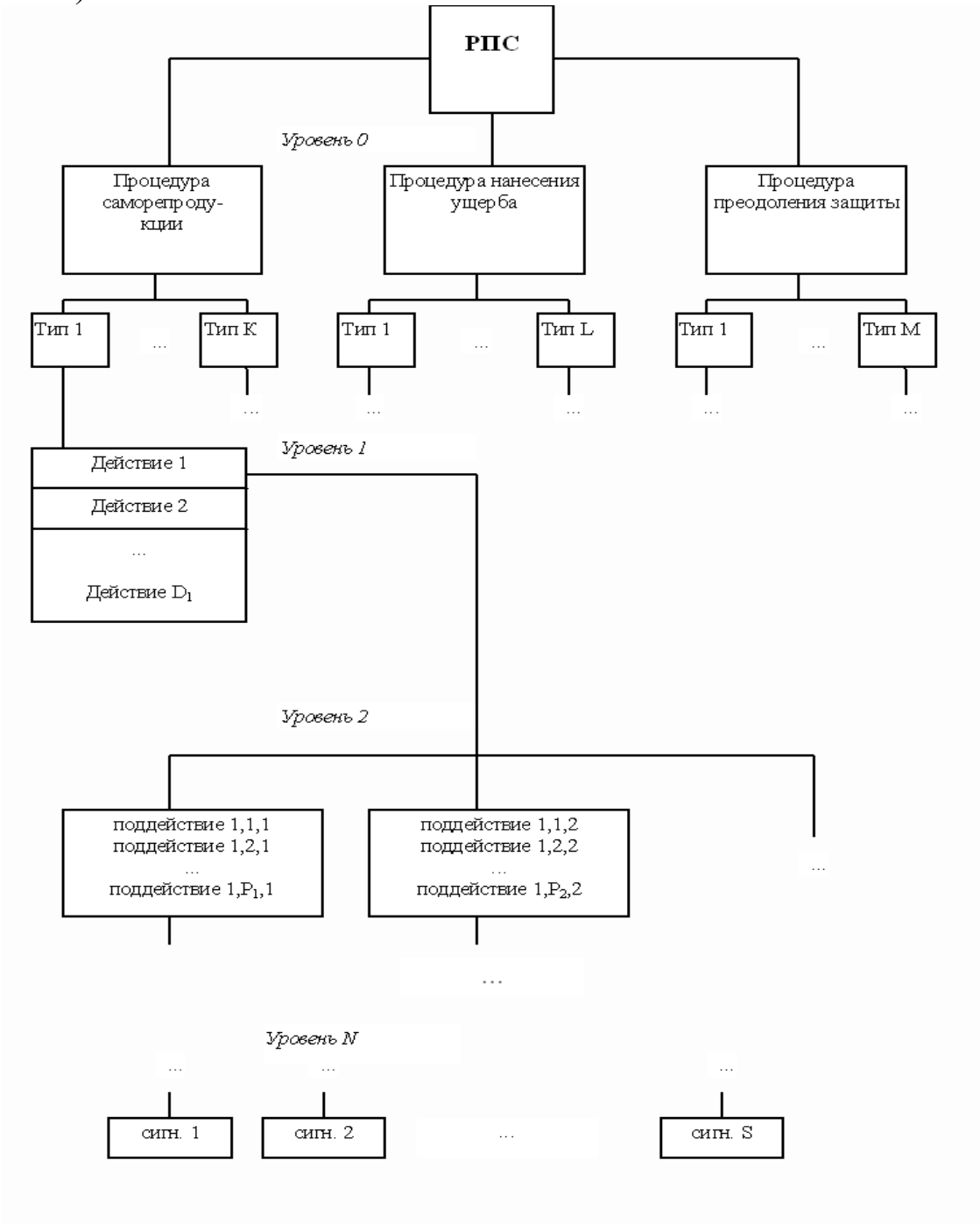

Рисунок 13 – Макет грамматики РПС в виде дерева свертки

Уровень 0 представляет собой конкретные типы РПС, на уровне 1 они раскрываются достаточно общими функциями, описывающими их алгоритм, на уровне 2 каждая из этих функций конкретизируется функциями более низкого уровня и т.д. до уровня N, где появляются сигнатуры. Таким образом, уровни 0..N-1 содержат нетерминалы, а уровень N – терминальные символы.

Если бы существовала полная аналогия между анализом программ в исполняемом коде и компиляцией, то на этом этапе анализ программ мог быть закончен, т.к. входная цепочка лексем (исполняемый код) должна была бы быть либо принята синтаксическим анализатором (РПС присутствует), либо отвергнута (РПС нет). Однако при компиляции программ цепочка задается абсолютно строго: известны ее длина, порядок следования терминальных символов и т.п. Так же строго задается и грамматика входного языка.

При анализе исполняемого кода это не так:

- могут быть не распознаны некоторые лексемы. Это следует из того, что, как отмечалось, макроассемблерные конструкции могут быть представлены бесконечным числом регулярным ассемблерных выражений;
- порядок следования лексем может быть известен с некоторой вероятностью или вообще неизвестен. Передачи управления в программе приводят к тому, что лексемы, стоящие рядом в программном файле, могут исполняться совершенно непоследовательно, и наоборот;
- грамматика языка может пополняться, так как могут возникать новые типы РПС или механизмы их работы.

Таким образом, окончательное заключение об отсутствии или наличии РПС можно дать только на этапе семантического анализа, а задачу этого этапа можно конкретизировать как свертку терминальных символов в нетерминалы как можно более высокого уровня там, где входная цепочка задана строго.

Этап перевода на язык высокого уровня выполняется с полученным ассемблерным текстом программы и состоит в нахождении и выделении соответствующим образом управляющих конструкций, таких как: циклы, подпрограммы и т.п.; основных структур данных. В какой-то мере этот этап также уже реализован: в частности, дизассемблер Sourcer ((c) V Communications) выделяет процедуры и некоторые структуры данных еще на этапе дизассемблирования, а для выделения управляющих конструкций служит специальная утилита ASMTool.

Семантический анализ программы, как уже отмечалось, удобнее всего вести на языке высокого уровня. Однако на сегодняшний день задача о переводе из машинных кодов на язык высокого уровня не имеет приемлемого решения. Поэтому на данном этапе вполне

подходящим языком более высокого уровня мог бы стать специализированный макроассемблер, который был описан в п. 3.1.1. Специализированным он называется в том плане, что нацелен на выделение макроконструкций, используемых в РПС. Не нужно смешивать этот процесс с возможным переводом на язык макроассемблера на предыдущих этапах, т.к. там осуществлялась свертка лексем, а не перевод всего текста.

Семантический анализ программы – исследование программы изучением смысла составляющих ее функций (процедур) в аспекте операционной среды компьютера.

Но перевод программ с языка высокого уровня в исполняемые коды не лишает ее смысла, поэтому такое определение не может претендовать на полноту. Напротив, новое определение подразумевает полное сохранение смысла программы и ее интерпретацию компьютерной системой при любой форме представления. Этот этап должен дать окончательный ответ на вопрос о том, содержит ли входной исполняемый код РПС, и если да, то какого типа. При этом он использует всю полученную на предыдущих этапах информацию, которая, как уже отмечалось, может считаться правильной только с некоторой вероятностью, причем не исключены вообще ложные факты или умозаключения.

Таким образом, формально на этапе семантического должна быть закончена свертка нетерминалов, полученных на этапе синтаксического анализа, в нетерминалы уровней 0, 1, 2. Очевидно, что нечеткость имеющейся информации приводит к появлению на этом этапе некоторой машины логического вывода со всеми присущими ей элементами: форме представления знаний и правил, алгоритмами логического вывода, вероятностными заключениями и т.п.

В целом, задача семантического исследования исполняемого кода программ является очень сложной и нуждается в серьезных исследованиях. Вероятнее всего, что полностью автоматизировать ее будет невозможно.

Как мы видим, оба этапа – синтаксического и семантического анализа – преследуют общую цель: построить дерево грамматического разбора и тем самым закончить анализ кода.

Известно, что при построении компиляторов все методы синтаксического анализа можно разбить на два класса: восходящие и нисходящие. Первые (методы сверху вниз) начинают с правила

грамматики, определяющего конечную цель анализа, и пытаются так наращивать дерево грамматического разбора, чтобы его узлы соответствовали синтаксису анализируемой цепочки. Вторые (методы снизу вверх) берут за основу терминальные символы и пытаются их свернуть в нетерминалы все более и более высоких уровней.

Совершенно аналогично, в теории экспертных систем различают два подхода: с прямой цепочкой рассуждений и с обратной. Первые, опираясь на известные факты, пытаются по ним построить умозаключение. Вторые, наоборот, беря за основу некоторую гипотезу, пытаются найти данные для ее подтверждения или опровержения.

Первые последовательно строят гипотезы, что РПС относится к типу 1, типу 2 и т.д. (т.е. начинает с корня дерева свертки) и постепенно спускаются вниз, на каждом этапе пытаясь опровергнуть или подтвердить свою гипотезу, ища соответствующие сигнатуры. Вторые начинают с терминалов уровня N и постепенно поднимаются вверх, сворачивая их в нетерминалы высших уровней с соответствующими вероятностями.

Можно охарактеризовать эти методы и с другой точки зрения: первые можно назвать методами ведущего семантического разбора (где машина логического вывода является первичной, а синтаксический анализатор помогает ей в подборе фактов), а вторые методами ведущего синтаксического разбора (где первичной будет синтаксическая свертка, которую облегчают вероятностные рассуждения семантического анализатора). К точно таким же выводам приходят авторы в, рассуждая с несколько других позиций. Более того, они предлагают как наиболее эффективный и многообещающий метод, основанный на сочетании этих двух, но ведь именно сочетание методов логического вывода привело к созданию машины логического вывода с косвенной цепочкой рассуждений, которая объединяет достоинства как первого, так и второго подхода.

Общая структура отражает взаимодействие основных модулей системы анализа, описанных выше, а также управляющих модулей, методов статического и динамического исследования и интерфейса с пользователем .

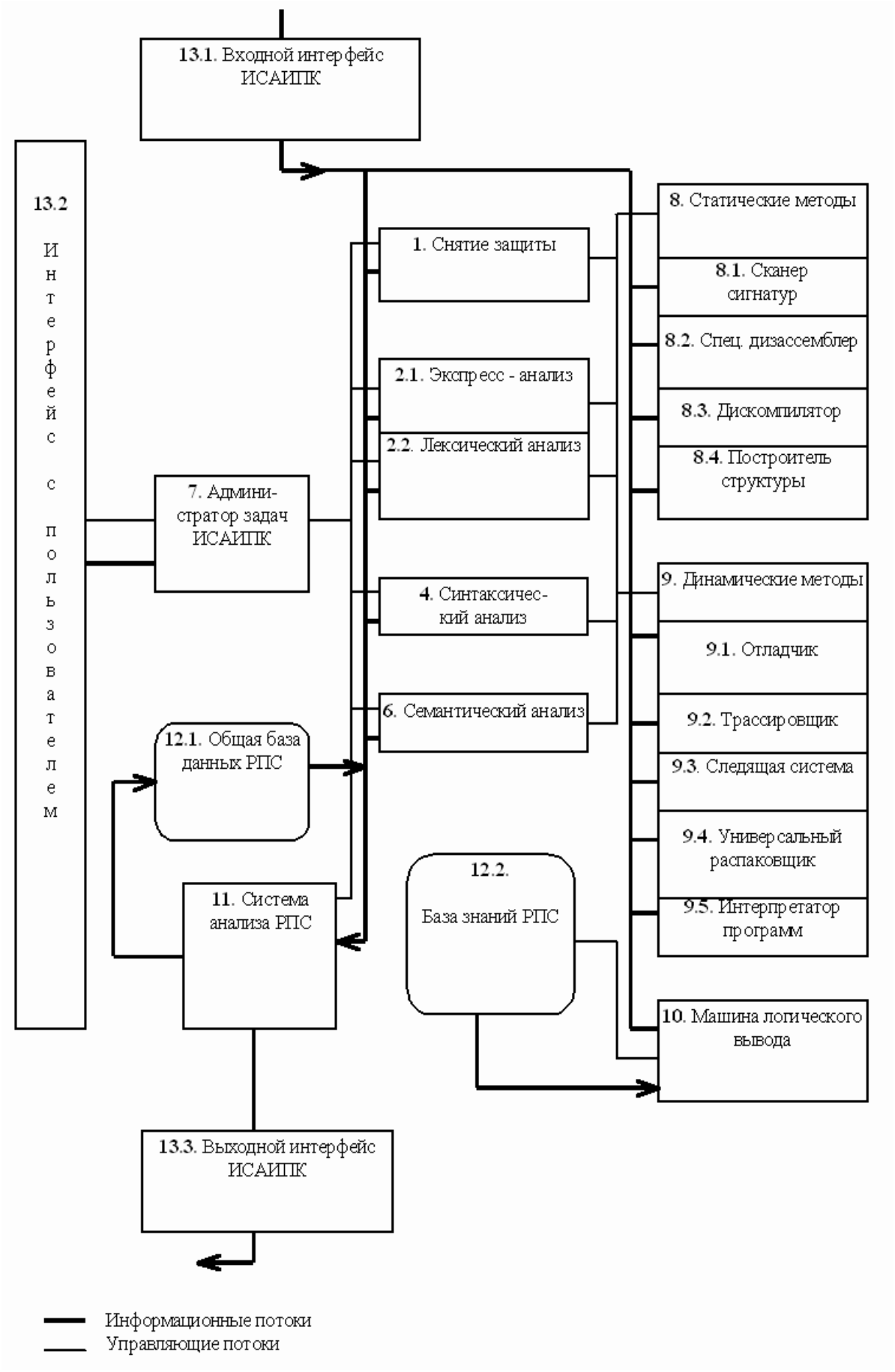

Рисунок 14 – Общая структурная схема системы анализа

5.2 Метод экспериментов

Метод экспериментов заключается в проведении многократных экспериментов с изучаемой программой и сравнительном анализе полученных результатов. Изучаемая программа рассматривается как «черный ящик», для которого известны входные и выходные данные, но неизвестно внутреннее устройство. Задача аналитика заключается в том, чтобы, подбирая входные данные, восстановить алгоритмы функционирования «черного ящика».

Эффективность метода экспериментов слабо зависит от программной реализации системы защиты (на нее, в частности, не влияет применение средств защиты программного обеспечения от изучения) и определяется в первую очередь сложностью анализируемых алгоритмов. Метод экспериментов эффективен при анализе программ, реализующих относительно простые алгоритмы.

Метод экспериментов редко применяется в чистом виде. Чаще он служит дополнением к динамическому или статическому методу. Это обусловлено тем, что, как правило, восстанавливаемые алгоритмы оказываются слишком сложными для данного метода. Системы защиты, сложность алгоритмов которых позволяет ограничиться при анализе методом экспериментов, существуют, но встречаются довольно редко.

5.3 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. Эффективные программы перевода на более понятные языки высокого уровня, к сожалению, отсутствуют.

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. Эффективность метода не зависит от сложности алгоритмов защиты. Успех здесь в меньшей степени, чем в методе экспериментов, зависит от удачливости аналитика. Это достоинства метода.

Для применения статического метода достаточно иметь программное обеспечение. Не требуется, чтобы анализируемая программа была работоспособна. Это очень удобно при исследовании программных комплексов, требующих дорогостоящих технических средств, которые отсутствуют у аналитика.

Статический метод позволяет использовать и дополнительную информацию, содержащуюся в программном обеспечении (имена экспортируемых функций, вызываемых библиотек динамической компоновки и т.п.), что недоступно для метода экспериментов с «черным ящиком». Статический метод в большей мере, чем другие подходы, допускает автоматизацию отдельных этапов анализа.

С другой стороны, при применении статического метода аналитик может встретиться и со значительными трудностями. Далеко не всегда удается найти подходящую программу дизассемблирования. Для того чтобы провести дизассемблирование, надо знать, как осуществляется преобразование информации в вычислительной системе. А документация по этим вопросам для целого ряда мощных вычислительных систем, как правило, малодоступна.

К любой вычислительной системе должны прилагаться программные средства, позволяющие пользователю разрабатывать собственные программы на привычных языках программирования или на модификациях этих языков. Должны прилагаться и соответствующие компиляторы, переводящие листинги программ в

машинные коды соответствующего компьютера. В принципе аналитик на основе этих программ имеет возможность восстанови и способы обработки информации в вычислительной системе (например методом экспериментов с черным ящиком). При этом, однако, аналитик придется решать сложные и трудоемкие задачи, далеко выходящие за рамки анализа конкретной системы защиты информации.

При практической реализации алгоритмов дизассемблирования возникают следующие проблемы.

1. Проблема восстановления символических имен. В программе, на писанной на языке ассемблера, все переменные, метки, процедуры и сегменты имеют символические имена. При компиляции программы эти имена заменяются физическими адресами. В скомпилированном машинной/ 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 экспортируемых функций, необходимую для их импорта программами). Обычно эта проблема решается путем «придумывания» дизассемблером символических имен типа *var*1, *label*2, *proc*3 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 следуют друг за другом подряд, 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.

Например, машинный код F7 D1 2B F9 дизассемблируется следующим образом:

```
not cx
sub di,cx
```

Если же дизассемблер решил, что граница команд пролегает между байтами F7 и D1, то этот же код будет дизассемблирован так:

```
shr word ptr [bp+di],1

stc
```

Таким образом, при работе программ дизассемблирования могут возникать серьезные ошибки.

Дизассемблеры условно можно разделить на «глупые» (dumb) и «умные» (smart). «Глупые» дизассемблеры даже не пытаются решить описанные 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. Для получения полного ассемблеровского текста оверлейной программы необходимо дизассемблировать все ее оверлейные модули. Существует ряд приемов программирования, при применении которых генерируется машинный код, некорректно воспринимаемый дизассемблером.

Следует отметить, что объем современных программных продуктов измеряется десятками и сотнями мегабайтов, размер листинга программы на языке ассемблера обычно в десятки раз превышает размеры исполняемого файла. Значительная часть программного обеспечения связана с организацией интерфейса, анализом конфигурации компьютера и ОС, распределением и

загрузкой памяти и т.п. Таким образом, большая часть программы никак не связана с защитой, и описания алгоритмов защиты (или других алгоритмов, интересующих аналитика) занимают мизерную часть программного обеспечения. Найти их в тексте программы не проще, чем иголку в стоге сена. Особую трудность составляет то обстоятельство, что отдельные, сравнительно небольшие и обозримые участки листинга мало информативны с точки зрения понимания функций, исполняемых данным участком программы. Расположены отдельные блоки программы в исполняемом файле достаточно хаотично.

В связи с вышеизложенным актуальна задача разработки специальных программных средств автоматизации анализа ассемблеровских листингов. Эти средства должны обладать элементами искусственного интеллекта, уметь строить графы вызова процедур, по тем или иным признакам автоматически выделять нужные участки листинга, осуществлять автоматический перевод текста с ассемблера на более понятный язык и т.п. К сожалению, разработка подобных программных средств является весьма трудоемкой, хотя и разрешимой задачей. Указанные программные средства будут иметь достаточно ограниченный спрос. Поэтому проработка вопросов автоматизации анализа находится лишь в зачаточном состоянии.

В настоящее время статический метод используется чаще всего вспомогательный инструмент для проверки предположений о восстанавливаемых алгоритмах защиты. Большинство аналитиков отдают предпочтение описываемому ниже динамическому методу. На наш взгляд, роль статического метода может возрасти по мере развития средств автоматизации анализа и дальнейшего усложнения и увеличения по объему программного обеспечения анализируемых продуктов.

Рассмотрим пример применения статического метода.

Используем статический метод анализа с применением дизассемблера Sourcer для решения задачи восстановления алгоритма генерации псевдослучайных чисел программы Turbo C, вызываемой функциями srand – задание начального положения генератора и rand – получение очередного псевдослучайного числа. Эта задача имеет самостоятельный интерес. Разработчики алгоритмов защиты нередко (когда у них появляется потребность в получении последовательностей псевдослучайных чисел) используют

стандартные датчики псевдослучайных чисел (ДСЧ), встроенные в программное обеспечение. Поэтому полезно знать как устроен такой ДСЧ.

данный пример полезен и как прототип для решения других похожих задач. (Нередко разработчики предлагают специализированные библиотеки, содержащие процедуры защиты. Каждый обладатель библиотеки может включать процедуры из нее в свои программные продукты, решая с минимальными затратами проблемы защиты информации. В подобных случаях Для оценки качества предлагаемой защиты необходимо восстановить алгоритм защиты, включенный в библиотеку.

Один из возможных способов (далеко не единственный) восстановления алгоритма заключается в построении в среде Turbo C программы, включающей процедуру генерации псевдослучайных чисел, в компиляции программы, в дизассемблировании исполняемого файла программы и в восстановлении из полученного листинга исследуемого алгоритма. Ниже приведен листинг на языке C одного из вариантов программы, в которую входят Процедуры srand и rand.

```
#include <stdio.h>
#include <stdlib.h>
void Rand(int u0, int SeqLength);
//
// Точка входа в программу
//
main()
{ Rand (13, 10);
}
//
//  Функция  Rand  генерирует  псевдослучайную  последовательность
длины
// SeqLength
//  Используется  стандартный  ДСЧ  Turbo  C.  В  качестве  начальной
установки
// ДСЧ берется u0.
void Rand (int u0, int SeqLength)
{// Очередное псевдослучайное число
  int r;
  // Счетчик цикла
  int i ;
 // Задаем начальную установку ДСЧ
 srand(u0);
 // Печать пустой строки
 printf("\r\n");
 for (i = 0; i<SeqLength; i++)
```

```
{// Генерируем очередное псевдослучайное число
r = rand();
 // Инвертируем младший бит полученного числа
 r^=1;
 // Распечатываем результат
 printf(«%d»,r);
   }
}
```

Этот нестандартный вариант генерации псевдослучайной последовательности был выбран умышленно, так как в дальнейшем придется в дизассемблированном листинге выделить участок, отвечающий функциям *srand* и *rand*. Сделать это будет проще, если участок будет достаточно большим и нестандартным по внешнему виду.

Следующий шаг состоит в сохранении программы в файле rand.c и его компиляции – создании исполняемого файла rand.exe. При компиляции следует указать модель памяти small.

Далее необходимо получить с помощью дизассемблера листинг программы. Для этого был использован дизассемблер Sourcer. Затем в этом листинге нужно найти участки, отвечающие процедурам srand и rand, и перевести их текст на понятный язык. Заметим, что это непростая задача. Объем файла rand.c составленной программы равен 318 байтам, объем исполняемого файла rand.exe – 9257 байтам. Ситуация, когда объем исполняемого файла более чем в 10 раз превышает объем файла с листингом на языке высокого уровня и, в свою очередь, в десятки или сотки раз меньше результата дизассемблирования, является типичной. Если распечатать листинг средствами 003, то он займет 58 страниц мелкого текста. Для того чтобы найти в этом тексте исследуемые участки, необходимо более детально познакомиться с особенностями компиляции исходных текстов программ в исполняемые файлы.

Вид листинга дизассемблера определяется способом преобразования программы на языке высокого уровня в машинные коды, осуществляемого компилятором. Поэтому при анализе листинга неплохо знать, с помощью какого компилятора был сгенерирован машинный код. Вместе с тем большинство компиляторов придерживаются сложившейся практики генерации кода. Аналитику следует ориентироваться на эти соглашения, отдавая себе отчет, что конкретный компилятор совсем не обязан им следовать. К наиболее существенным для анализа относятся следующие правила.

1. Программа обычно начинается с вспомогательных процедур (проверка версии операционной системы, получение системной даты и системного времени компьютера и т.п.);
2. При компиляции программы код каждой функции занимает непрерывный участок файла и не прерывается фрагментами, относящимися к другим функциям;
3. Параметры передаются функции, как правило, через стек. Вызывающая функция записывает в стек последовательно каждый аргумент, а затем адрес памяти, куда должна возвратиться программа после исполнения вызываемой функции. Таким образом, в момент вызова функции в стеке, начиная с его вершины, оказываются записаны адрес возврата и аргументы функции. Параметры могут записываться, начиная с первого (стиль компиляторов языка Pascal) или начиная с последнего (стиль компиляторов языка C).
4. Для каждого аргумента функции используется целое машинное слово, даже если реальный размер аргумента меньше.
5. Адресация при обращении программы к сегменту стека осуществляется через регистр bp (для 32-разрядных программ – ebp или esp). Если программа откомпилирована с использованием 16-разрядного компилятора в модели памяти small, параметры функции располагаются по адресам [bp+4], [bp+6].
6. Большинство функций 16-разрядной программы начинаются командами

```
push bp
mov bp,sp
```

   а большинство функций 32-разрядной программы – командами

```
push ebp
mov ebp,esp
```

7. При выполнении функции для хранения промежуточных результатов используются регистры процессора. Для того чтобы информация, хранившаяся в регистрах до этого, не была потеряна, обычно в начале функции значения регистров копируются в стек командой push, а в конце восстанавливаются командой pop.
8. Если процедура имеет локальные переменные, то их значения также хранятся в стеке. В этом случае в начале процедуры

присутствует команда sub sp,n (или sub esp,n), где n – количество байтов, выделяемых под локальные переменные. Обращение к данным переменным проводится по адресам вида [bp-1], [bp-2], ... (или [ebp-1], [ebp-2], ...). Многие компиляторы оптимизируют генерируемый код таким образом, что наиболее часто используемые локальные переменные хранятся не в стеке, а в регистрах процессора.

9. Возвращаемое значение функции записывается в зависимости от типа возвращаемого значения и разрядности кода программы в регистр al, ax, eax или в пару регистров dx:ax.
10. Имеются традиционные способы реализации простейших операций языка высокого уровня. Так, обнуление регистра осуществляется командой *xor* или *sub* (например, *xor ax, ax* или *sub ax, ax*). Операции '^' языка С соответствует команда *xor*, операции '=' – команда mov или lea и т.п.

Для восстановления алгоритма генерации псевдослучайных чисел загрузим полученный с помощью дизассемблера Sourcer листинг в какой-нибудь текстовый редактор. Желательно иметь также справочник по командам языка ассемблера.

Прежде всего найдем участок листинга, содержащий функцию rand. Здесь нам может помочь то, что мы заранее включили в эту функцию достаточно нетипичный для подобных программ оператор

r^=1;

Выше отмечалось, что операции '^' на языке ассемблера обычно соответствует команда *xor*. Осуществим средствами выбранного редактора последовательный просмотр всех мест листинга, где встретился этот оператор. Мы многократно встретим команды обнуления регистра вида *xor bx, bx* и лишь один раз – команду

```
70A7:0346       xor ax,1.
```

Эта команда относится к функции, которой дизассемблер присвоил имя sub_10.

5.4 Проблемы автоматизации анализа при применении статистического метода

Однако более сложных ситуациях попытка разобраться в сущности алгоритм, программы без использования вспомогательных средств, автоматизирующих некоторые рутинные операции, может стать довольно трудоемкой задачей. В связи с этим проблема

разработки формальных методов анализа программ является актуальной.

Одной из задач, которые удобно возложить на средства автоматизации анализа программ, является задача выделения участков кода, связанных с обработкой информации, хранящейся в определенных ячейка) памяти. Это может оказаться полезным, например, при анализе программного обеспечения на наличие разрушающих воздействий (вирусов, закладок и т.п.). Допустим, перед аналитиком поставлена задача изучения алгоритма функционирования нового полиморфного вируса. В этом случае одной из целей анализа должно стать выявление кода внутри инфицированной программы, реализующего механизмы шифрования/расшифровки основного тела вируса. Из этого следует, что на начальной стадии анализа необходимо провести поиск фрагментов программы, в которых осуществляются преобразования информации, расположенной в кодовых сегментах, и в случае обнаружения таковых изучить алгоритмы их работы. Таким образом, анализ программного кода разбивается на два этапа: на первом этапе определяется, какая именно часть программы должна подвергнуться изучению, а на втором проводится непосредственное изучение выбранного фрагмента.

Следует отметить, что, имея дизассемблированный текст программы, не всегда удается простым поиском выражений выделить необходимые участки кода. Рассмотрим пример простейшего расшифровщика полиморфного вируса, созданного на основе общей схемы:

```
…
0000:3E80: 1E                  push ds
0000:3E81: 8CC8              push ds
0000:3E83: 05D703           push ds
0000:3E86: 8ED8               push ds
0000:3E88: 33DB               push ds
0000:3E8A: BE3001           push ds
0000:3E8D: 8A00               push ds
0000:3E8F: 3401                 push ds
0000:3E91: 8800                 push ds
0000:3E93: 43                     push ds
0000:3E94: 83FB04            push ds
```

```
0000:3E97: 75F4                 push ds
0000:3E99: A1EA03           push ds
```

Решать предложенную выше задачу путем поиска строк в листинге, имеющих вхождения подстроки «cs» (регистр кодового сегмента), бессмысленно, так как в результате мы получим только строку

```
0000:3E81: 8CC8                 mov ax,cs
```

На самом же деле этого недостаточно. Как будет показано далее, строки

```
0000:3ED8:  8A00          mov al,[si+bx]
0000:3E8F: 3401           xor al,01
0000:3E91: 8800           mov [si+bp],al
```

реализуют механизм наложения маски на код программы.
Алгоритм работы этой части программы состоит в следующем.

1. Поместить в регистр al содержимое ячейки, расположенной по адресу ds:[si+bx].

2. Произвести побитовое сложение по модулю 2 содержимого регистра al и числа 01.

3. Записать обратно в ячейку ds:[si+bx] результат, полученный на предыдущем шаге.

Однако анализ предыдущих строк кода показывает, что адрес ячейки ds:[si+bx] вычисляется на основе следующих значений регистров:

```
si=129h(mov si,0129h)
bx=0(xor bx,bx)
ds=cs+3D7h(mov ax,cs; add ax, 03D7; mov ds,ax)
```

Отсюда следует, что физический адрес равен cs + 3D7:129 = cs:3E99. Согласно листингу значение регистра cs полагается равным 0000, а это означает, что инструкции программы, расположенные по адресам 0000:3E8D – 0000:3E91, осуществляют преобразование содержимого ячейки 0000:3E99. Если обратить внимание на команды, расположенные по адресам 0000:3E93 – 0000:3E97, то становится очевидной организация циклического наложения маски на код, начинающийся с адреса 0000:3EA0.

Таким образом, значения адресов ячеек памяти, используемых в инструкциях 0000:3E8D – 0000:3E91, вычисляются на основе результатов исполнения инструкций 0000:3E81 – 0000:3E86, т.е. существует определенная «вычислительная» зависимость между

указанными блоками кода. Следовательно, решение проблемы формализации выявления участков кода, связанных с обработкой определенной информации, требует анализа подобных зависимостей между инструкциями программы.

Формальные модели, базирующиеся на фиксации зависимостей между операторами, были разработаны в отечественной науке в 80-х годах в рамках исследований по структурному синтезу программ. В их основе лежит понятие вычислительной модели, под которой понимается двойка

$S(X, F)$,

где X – упорядоченный набор переменных величин предметной области, F – упорядоченный набор зависимостей между переменными. Зависимость возникает между двумя переменными в том случае, если значение одной из них вычисляется на основе значения другой. В общем случае элементы множества F можно рассматривать как функции вида

$f(U,V)$ или $f(U \to V)$,

где U с X – список входных параметров зависимости, а V с X – список выходных параметров, значения которых вычисляются на основе значений переменных из X. Частный случай, когда $U = 0$, соответствует ситуации когда все величины из V являются константами. Для описания подобных зависимостей иногда используется выражение $f(U \to V)$.

Понятие вычислительной модели можно использовать для формализации анализа программного кода. В этом случае программа рассматривается как вычислительная модель, множество X которой составляют ячейки памяти, данные из которых обрабатываются программой, и регистры процессора. Множество F задается последовательностью инструкций программы и представляет собой схему программы, которая описывает каналы перемещения информации между ячейками памяти и регистрами процессора, но не содержит описаний алгоритмов ее обработки. Задача выделения участков кода, связанных с обработкой информации из определенных ячеек памяти (обозначим их набор через Us с X), сводится к выявлению множества переменных, значения которых вычисляются на их основе (обозначим Uf с X), и множества инструкций (обозначим Gf с F), реализующих соответствующие вычисления. В дальнейшем будем полагать, что множества Us, Uf и Gf упорядоченные. Общая схема алгоритма решения этой задачи выглядит следующим образом.

1. Определяются инструкция f_0, с которой начинается анализ (это может быть точка входа в программу, в интересующую аналитика функцию и т.п.), и инструкция f_k, до которой будет производиться анализ. На этом этапе в множество *Uf* заносятся элементы множества *Us*.

2. Выбирается очередной элемент u_i, из множества *Uf*. Далее последовательно в порядке, в котором должна выполняться программа (с учетом таких конструкций, как бинарные ветвления, вызовы подпрограмм и т. д.), просматриваются инструкции и отбираются те, в список входных параметров которых входит элемент u_i. Для отобранных инструкций определяются выходные параметры, которые должны быть занесены в *Uf*.

3. Определяется необходимость дальнейшей работы алгоритма. Если просмотрены все инструкции из диапазона $f_0 – f_k$ или второй шаг выполнен для всех элементов U_f, то работа алгоритма заканчивается, иначе снова выполняется второй шаг.

При реализации этого алгоритма следует учитывать проблемы, связанные с особенностями функционирования программ. Рассмотрим их на приведенном выше примере. Итак, будем решать задачу выявления инструкций, выполняющих преобразование информации в кодовом сегменте. В качестве начальной инструкции удобно выбрать команду

```
0000:3E81: 8CC8           mov ax,cs
```

так как в ней впервые встречается явное обращение к сегменту кода. Соответственно множество исходных данных U_s, а также множество U_f перед началом работы будет содержать один элемент u_1 – регистр *cs*. В качестве конечной инструкции для простоты выберем

```
0000:3E97: 75F4 jne 0000:3E8D
```

Первое выполнение второго шага приведет к добавлению в множество U_f элемента u_2 (регистра ax), а в множество G_f – элемента g_1, соответствующего команде

```
0000:3E81: 8CC8      mov ax,cs
```

При следующем выполнении второго шага в список отобранных инструкций попадут три команды:

```
0000:3E83:   05D703      add         ax,03D7h

0000:3E86:8ED8    mov  ds,ax

0000:3E91:8800    mov  [si+bx],al
```

Очевидно, что значение регистра аз вычисляется командой, расположенной по адресу 0000:3E86, на основе значения регистра ах, полученного при выполнении инструкции 0000:3E83, а это значение не соответствует значению регистра ах, вычисленного инструкцией 0000:3E81. По сути это уже другой элемент данных, отличный от того, который был занесен в множество U_f на предыдущей итерации второго шага алгоритма. Аналогично инструкция 0000:3E91 имеет дело с третьим значением регистра ах, отличным от двух предыдущих. Следовательно, при реализации второго шага алгоритма необходимо учитывать время жизни значения элемента данных, определяемое участком кода, при выполнении которого вычисленное значение этого элемента не меняется.

Еще одна проблема, которую необходимо учитывать при реализации второго шага предложенного алгоритма, связана с различением особенностей вычислительных зависимостей между элементами данных. Если рассмотреть в нашем примере команды, расположенные по адресам 0000:3E81-0000:3E86, то очевидно, что для вычисления значения регистра ds необходимо и достаточно иметь значение регистра cs. Однако если бы по адресу 0000:3E81 была расположена команда типа mov ax, cx, то значение регистра ах оказалось бы необходимым, но недостаточным для вычисления ds. Таким образом, программа, реализующая данный алгоритм, должна уметь различать такие «сильные» и «слабые» зависимости.

Следует обратить внимание на то, что использование подобных методов выделения фрагментов кода упирается в общие для статического метода проблемы использования косвенной адресации и косвенных переходов, когда значения адресов данных или точек передачи управление могут быть определены только во время выполнения программы. В связи с этим такая методика должна применяться в комплексе с динамические методом.

6 Запутывающие преобразования программ

6.1 Понятие запутывающего преобразования

В этом разделе мы дадим формальное определение запутывателя свойства π класса программ P. Мы введём некоторые показатели качества запутывающих преобразований и перечислим разнообразные запутывающие преобразования, каждое из которых по отдельности усложняет граф потока управления программы. Опубликованные методы запутывания, которые в настоящее время применяются в программных инструментах, как свободно-распространяемых, так и коммерческих, являются комбинацией нескольких перечисленных запутывающих преобразований. Некоторые из методов запутывания будут описаны в конце раздела.

Приводимое здесь определение сформулировано В. А. Захаровым.

Эффективное вычисление – это вычисление, требующее полиномиального от длины входа времени и полиномиальной от длины входа памяти. Эффективная программа (машина Тьюринга) – программа, работающая полиномиальное от длины входа время и требующая полиномиальную от длины входа рабочую память на всех входах, на которых программа завершается.

1. Пусть П – множество всех программ (машин Тьюринга), удовлетворяющих сформулированным выше ограничениям, и пусть программа $p \in \Pi$ вычисляет функцию

$$f_p : Input \to Output,$$

подмножество $\pi \subseteq \Pi$ называется функциональным свойством если

$$\forall p_1, p_2 \in \Pi\,(f_{p_1} = f_{p_2} \Rightarrow (p_1 \in \pi \Leftrightarrow p_2 \in \pi))$$

2. Пусть π – функциональное свойство, $P \subseteq \Pi$ – класс программ такой, что существует эффективная программа с такая, что для любой программы $p \in P$

$$c(p) = \begin{cases} 1, & если\ p \in \pi \\ 0, & если\ p \notin \pi \end{cases}$$

Другими словами, для функционального свойства π мы определяем класс программ P таких, что существует эффективная программа-распознаватель с свойства π по программе p из класса P .

3. Вероятностная программа o называется запутывателем класса P относительно свойства π,(P,π)-запутывателем, если выполняются условия:

а) (эквивалентность преобразования запутывания). Для любой $p \in P$ и

$$p' \in o(p)$$
$$f_p = f_{p'},$$
$$|p'| = poly(|p|),$$
$$\forall x \in Dom_{f_p} \quad time_{p'}(x) = poly(time_p(x))$$

Здесь $y = poly(x)$ означает, что y ограничен полиномом некоторой степени от переменной x, $time_p(x)$ – время выполнения программы p на входе x, $|p|$ – размер программы p.

б) (трудность определения свойств по запутанной программе). Для любого полинома q и для любой программы (вероятностной машины Тьюринга) a такой, что $a(o(P)) = \{0,1\}$, и для любой $p \in P$ выполняется $time_a(o(p)) = poly(|o(p)|)$, существует программа (вероятностная машина Тьюринга с оракулом) b, и при этом для любой $p \in P$.

$$|\Pr(a(o(p)) = c(p)) - \Pr(b^p(1^{|p|}) = c(p))| \leq \frac{1}{q(|p|)}.$$

Другими словами, вероятность определить свойство π по запутанной программе равна вероятности определения свойства π только по входам и выходам функции f_p. То есть, наличие текста запутанной программы ничего не даёт для выявления свойств этой программы.

Универсальный запутыватель – это программа O, которая для любого класса программ P и любого свойства π является (P, π)-запутывателем. Как сказано ранее, универсального запутывателя не существует. Доказательство заключается в построении специального класса программ P и выборе такого свойства π, что для любого преобразования программы из этого класса свойство π устанавливается легко. Однако вопрос о том, существуют ли запутыватели для отдельных классов свойств программ, и насколько широки и практически значимы эти классы свойств, остаётся открытым. С практической точки зрения запутывание программы можно рассматривать как такое преобразование программы, которое делает её обратную инженерию экономически невыгодной. Несмотря на слабую теоретическую проработку, уже разработано большое количество инструментов для запутывания программ.

6.2 Классификация запутывающих преобразований

Запутывающие преобразования можно разделить на несколько групп в зависимости от того, на трансформацию какой из компонент программы они нацелены.

- Преобразования форматирования, которые изменяют только внешний вид программы. К этой группе относятся преобразования, удаляющие комментарии, отступы в тексте программы или переименовывающие идентификаторы;
- Преобразования структур данных, изменяющие структуры данных, с которыми работает программа. К этой группе относятся, например, преобразование, изменяющее иерархию наследования классов в программе, или преобразование, объединяющее скалярные переменные одного типа в массив. В данной работе мы не будем рассматривать запутывающие преобразования этого типа;
- Преобразования потока управления программы, которые изменяют структуру её графа потока управления, такие как развёртка циклов, выделение фрагментов кода в процедуры, и другие. Данная статья посвящена анализу именно этого класса запутывающих преобразований;
- Превентивные преобразования, нацеленные против определённых методов декомпиляции программ или использующие ошибки в определённых инструментальных средствах декомпиляции.

6.2.1 Преобразования форматирования

К преобразованиям форматирования относятся удаление комментариев, переформатирование программы, удаление отладочной информации, изменение имён идентификаторов**.**

Удаление комментариев и переформатирование программы применимы, когда запутывание выполняется на уровне исходного кода программы. Эти преобразования не требуют только лексического анализа программы. Хотя удаление комментариев – одностороннее преобразование, их отсутствие не затрудняет сильно обратную инженерию программы, так как при обратной инженерии наличие хороших комментариев к коду программы является скорее исключением, чем правилом. При переформатировании программы

исходное форматирование теряется безвозвратно, но программа всегда может быть переформатирована с использованием какого-либо инструмента для автоматического форматирования программ (например, indent для программ на Си).

Удаление отладочной информации применимо, когда запутывание выполняется на уровне объектной программы. Удаление отладочной информации приводит к тому, что имена локальных переменных становятся невосстановимы.

Изменение имён локальных переменных требует семантического анализа (привязки имён) в пределах одной функции. Изменение имён всех переменных и функций программы помимо полной привязки имён в каждой единице компиляции требует анализа межмодульных связей. Имена, определённые в программе и не используемые во внешних библиотеках, могут быть изменены произвольным, но согласованным во всех единицах компиляции образом, в то время как имена библиотечных переменных и функций меняться не могут. Данное преобразование может заменять имена на короткие автоматически генерируемые имена (например, все переменные программы получат имя *v*<номер> в соответствии с их некоторым порядковым номером). С другой стороны, имена переменных могут быть заменены на длинные, но бессмысленные (случайные) идентификаторы в расчёте на то, что длинные имена хуже воспринимаются человеком.

6.2.2 Преобразования потока управления

Преобразования потока управления изменяют граф потока управления одной функции. Они могут приводить к созданию в программе новых функций. Краткая характеристика методов приведена ниже.

Открытая вставка функций (function inlining) заключается в том, что тело функции подставляется в точку вызова функции. Данное преобразование является стандартным для оптимизирующих компиляторов. Это преобразование одностороннее, то есть по преобразованной программе автоматически восстановить вставленные функции невозможно. В рамках данной статьи мы не будем рассматривать подробно прямую вставку функций и её эффект на запутывание и распутывание программ.

Вынос группы операторов (function outlining). Данное преобразование является обратным к предыдущему и хорошо дополняет его. Некоторая группа операторов исходной программы выделяется в отдельную функцию. При необходимости создаются формальные параметры. Преобразование может быть легко обращено компилятором, который (как было сказано выше) может подставлять тела функций в точки их вызова.

Отметим, что выделение операторов в отдельную функцию является сложным для запутывателя преобразованием. Запутыватель должен провести глубокий анализ графа потока управления и потока данных с учётом указателей, чтобы быть уверенным, что преобразование не нарушит работу программы.

Непрозрачные предикаты (opaque predicates). Основной проблемой при проектировании запутывающих преобразований графа потока управления является то, как сделать их не только дешёвыми, но и устойчивыми. Для обеспечения устойчивости многие преобразования основываются на введении непрозрачных переменных и предикатов. Сила таких преобразований зависит от сложности анализа непрозрачных предикатов и переменных.

Переменная v является непрозрачной, если существует свойство πотносительно этой переменной, которое априори известно в момент запутывания программы, но трудноустанавливаемо после того, как запутывание завершено. Аналогично, предикат P называется непрозрачным, если его значение известно в момент запутывания программы, но трудноустанавливаемо после того, как запутывание завершено.

Непрозрачные предикаты могут быть трёх видов: P^F – предикат, который всегда имеет значение «ложь», P^T – предикат, который всегда имеет значение «истина», и $P^?$ – предикат, который может принимать оба значения, и в момент запутывания текущее значение предиката известно.

Группой Томборсона разработаны методы построения непрозрачных предикатов и переменных, основанные на «встраивании» в программу вычислительно сложных задач, например, задачи 3-выполнимости . Некоторые возможные способы введения непрозрачных предикатов и непрозрачных выражений вкратце перечислены ниже.

– использование разных способов доступа к элементы массива. Например, в программе может быть создан массив (скажем, a),

который инициализируется заранее известными значениями, далее в программу добавляются несколько переменных (скажем, *i*, *j*), в которых хранятся индексы элементов этого массива. Теперь непрозрачные предикаты могут иметь вид $a[i] = a[j]$. Если к тому же переменные *i* и *j* в программе изменяются, существующие сейчас методы статического анализа алиасов позволят только определить, что и *i*, и *j* могут указывать на любой элемент массива *a*;

- использование указателей на специально создаваемые динамические структуры. В этом подходе в программу добавляются операции по созданию ссылочных структур данных (списков, деревьев), и добавляются операции над указателями на эти структуры, подобранные таким образом, чтобы сохранялись некоторые инварианты, которые и используются как непрозрачные предикаты.
- конструирование булевских выражений специального вида;
- построение сложных булевских выражений с помощью эквивалентных преобразований из формулы *true*. В простейшем случае мы можем взять *k* произвольных булевских переменных $x_1 ... x_k$ и построить из них тождество $(x_1 \vee \overline{x_1}) \wedge \ldots \wedge (x_k \vee \overline{x_k})$. Далее с помощью эквивалентных алгебраических преобразований часть скобок (или все) раскрываются, и в результате получается искомый непрозрачный предикат.
- использование комбинаторных тождеств, например $\sum_{i=0}^{n} C_n^i = 2^n$.

Внесение недостижимого кода (adding unreachable code). Если в программу внесены непрозрачные предикаты видов P^F или P^T, ветки условия, соответствующие условию «истина» в первом случае и условию «ложь» во втором случае, никогда не будут выполняться. Фрагмент программы, который никогда не выполняется, называется недостижимым кодом. Эти ветки могут быть заполнены произвольными вычислениями, которые могут быть похожи на действительно выполняемый код, например, собраны из фрагментов той же самой функции. Поскольку недостижимый код никогда не выполняется, данное преобразование влияет только на размер запутанной программы, но не на скорость её выполнения. Общая задача обнаружения недостижимого кода, как известно, алгоритмически неразрешима. Это значит, что для выявления недостижимого кода должны применяться различные эвристические

методы, например, основанные на статистическом анализе программы.

Внесение мёртвого кода (adding dead code). В отличие от недостижимого кода, мёртвый код в программе выполняется, но его выполнение никак не влияет на результат работы программы. При внесении мёртвого кода запутыватель должен быть уверен, что вставляемый фрагмент не может влиять на код, который вычисляет значение функции. Это практически значит, что мёртвый код не может иметь побочного эффекта, даже в виде модификации глобальных переменных, не может изменять окружение работающей программы, не может выполнять никаких операций, которые могут вызвать исключение в работе программы.

Внесение избыточного кода (adding redundant code). Избыточный код, в отличие от мёртвого кода выполняется, и результат его выполнения используется в дальнейшем в программе, но такой код можно упростить или совсем удалить, так как вычисляется либо константное значение, либо значение, уже вычисленное ранее. Для внесения избыточного кода можно использовать алгебраические преобразования выражений исходной программы или введение в программу математических тождеств. Например, можно воспользоваться комбинаторным тождеством $\sum_{i=0}^{8} C_8^i = 2^8 = 256$ и заменить везде в программе использование константы 256 на цикл, который вычисляет сумму биномиальных коэффициентов по приведённой формуле.

Подобные алгебраические преобразования ограничены целыми значениями, так как при выполнении операций с плавающей точкой возникает проблема накопления ошибки вычислений. Например, выражение sin^2x+cos^2x при вычислении на машине практически никогда не даст в результате значение 1. С другой стороны, при операциях с целыми значениями возникает проблема переполнения. Например, если использование 32-битной целой переменной x заменено на выражение $x*p/q$, где p и q гарантированно имеют одно и то же значение, при выполнении умножения $x*p$ может произойти переполнение разрядной сетки, и после деления на q получится результат не равный p. В качестве частичного решения задачи можно выполнять умножение в 64-битных целых числах.

Преобразование сводимого графа потока управления к несводимому (transforming reducible to non-reducible flow graph). Когда целевой язык (байт-код или машинный язык) более

выразителен, чем исходный, можно использовать преобразования, «противоречащие» структуре исходного языка. В результате таких преобразований получаются последовательности инструкций целевого языка, не соответствующие ни одной из конструкций исходного языка.

Например, байт-код виртуальной машины Java содержит инструкцию *goto*, в то время как в языке Java оператор *goto* отсутствует. Графы потока управления программ на языке Java оказываются всегда сводимыми, в то время как в байт-коде могут быть представлены и несводимые графы.

Можно предложить запутывающее преобразование, которое трансформирует сводимые графы потока управления функций в байт-коде, получаемых в результате компиляции Java-программ, в несводимые графы. Например, такое преобразование может заключаться в трансформации структурного цикла в цикл с множественными заголовками с использованием непрозрачных предикатов. С одной стороны, декомпилятор может попытаться выполнить обратное преобразование, устраняя несводимые области в графе, дублируя вершины или вводя новые булевские переменные. С другой строны, распутыватель может с помощью статических или статистических методов анализа определить значение непрозрачных предикатов, использованных при запутывании, и устранить никогда не выполняющиеся переходы. Однако, если догадка о значении предиката окажется неверна, в результате получится неправильная программа.

Устранение библиотечных вызовов (eliminating library calls). Большинство программ на языке Java существенно используют стандартные библиотеки. Поскольку семантика библиотечных функций хорошо известна, такие вызовы могут дать полезную информацию при обратной инженерии программ. Проблема усугубляется ещё и тем, что ссылки на классы библиотеки Java всегда являются именами, и эти имена не могут быть искажены.

Во многих случаях можно обойти это обстоятельство, просто используя в программе собственные версии стандартных библиотек. Такое преобразование не изменит существенно время выполнения программы, зато значительно увеличит её размер и может сделать её непереносимой.

Для программ на традиционных языках эта проблема стоит менее остро, так как стандартные библиотеки, как правило, могут

быть скомпонованы статически вместе с самой программой. В данном случае программа не содержит никаких имён функций из стандартной библиотеки.

Переплетение функции (function interleaving). Идея этого запутывающего преобразования в том, что две или более функций объединяются в одну функцию. Списки параметров исходных функций объединяются, и к ним добавляется ещё один параметр, который позволяет определить, какая функция в действительности выполняется.

Клонирование функций (function cloning). При обратной инженерии функций в первую очередь изучается сигнатура функции, а также то, как эта функция используется, в каких местах программы, с какими параметрами и в каком окружении вызывается. Анализ контекста использования функции можно затруднить, если каждый вызов некоторой функции будет выглядеть как вызов какой-то другой, каждый раз новой функции. Может быть создано несколько клонов функции, и к каждому из клонов будет применён разный набор запутывающих преобразований.

Развёртка циклов (loop unrolling). Развёртка циклов применяется в оптимизирующих компиляторах для ускорения работы циклов или их распараллеливания. Развёрка циклов заключается в том, что тело цикла размножается два или более раз, условие выхода из цикла и оператор приращения счётчика соответствующим образом модифицируются. Если количество повторений цикла известно в момент компиляции, цикл может быть развёрнут полностью.

Разложение циклов (loop fission). Разложение циклов состоит в том, что цикл с сложным телом разбивается на несколько отдельных циклов с простыми телами и с тем же пространством итерирования.

Реструктуризация графа потока управления. Структура графа потока управления, наличие в графе потока управления характерных шаблонов для циклов, условных операторов и т. д. даёт ценную информацию при анализе программы. Например, по повторяющимся конструкциям графа потока управления можно легко установить, что над функцией было выполнено преобразование развёртки циклов, а далее можно запустить специальные инструменты, которые проанализируют развёрнутые итерации цикла для выделения индуктивных переменных и свёртки цикла. В качестве меры противодействия может быть применено такое преобразование графа потока управления, которое приводит граф к однородному

(«плоскому») виду. Операторы передачи управления на следующие за ними базовые блоки, расположенные на концах базовых блоков, заменяются на операторы передачи управления на специально созданный базовый блок диспетчера, который по предыдущему базовому блоку и управляющим переменным вычисляет следующий блок и передаёт на него управление. Технически это может быть сделано перенумерованием всех базовых блоков и введением новой переменной, например state, которая содержит номер текущего исполняемого базового блока. Запутанная функция вместо операторов if, for и т. д. будет содержать оператор switch, расположенный внутри бесконечного цикла.

Локализация переменных в базовом блоке. Это преобразование локализует использование переменных одним базовым блоком. Для каждого запутываемого базового блока функции создаётся свой набор переменных. Все использования локальных и глобальных переменных в исходном базовом блоке заменяются на использование соответствующих новых переменных. Чтобы обеспечить правильную работу программы между базовыми блоками вставляются так называемые связующие (connective) базовые блоки, задача которых скопировать выходные переменные предыдущего базового блока в входные переменные следующего базового блока.

Применение такого запутывающего преобразование приводит к появлению в функции большого числа новых переменных, которые, однако, используются только в одном-двух базовых блоках, что запутывает человека, анализирующего программу.

При реализации этого запутывающего преобразования возникает необходимость точного анализа указателей и контекстно-зависимого межпроцедурного анализа. В противном случае нельзя гарантировать, что запись по какому-либо указателю или вызов функции не модифицируют настоящую переменную, а не текущую рабочую копию.

Расширение области действия переменных. Данное преобразование по смыслу обратно предыдущему. Это преобразование пытается увеличить время жизни переменных настолько, насколько можно. Например, вынося блочную переменную на уровень функции или вынося локальную переменную на статический уровень, расширяется область действия переменной и усложняется анализ программы. Здесь используется то, что

глобальные методы анализа (то есть, методы, работающие над одной функцией в целом) хорошо обрабатывают локальные переменные, но для работы со статическими переменными требуются более сложные методы межпроцедурного анализа.

Для дальнейшего запутывания можно объединить несколько таких статических переменных в одну переменную, если точно известно, что переменные не могут использоваться одновременно. Очевидно, что преобразование может применяться только к функциям, которые никогда не вызывают друг друга непосредственно или через цепочку других вызовов.

6.3 Анализ запутанных программ

Статическое устранение мёртвого кода (dead-code elimination) имеет целью выявить в программе код, который выполняется, но не оказывает влияние на результат работы программы.

Статическая минимизация количества переменных (variable minimization) имеет целью уменьшить количество используемых в функции локальных переменных за счёт объединения переменных, времена жизни значений в которых не пересекаются, в одну переменную. Стандартная техника, которая используется для минимизации количества переменных, состоит в построении графа перекрытия переменных с помощью итерационного решения уравнения потока данных и последующей раскраске вершин этого графа в минимальное или близкое к минимальному количество цветов.

Статическое продвижение констант и копий (constant and copy propagation) заключается в продвижении константных выражений как можно дальше по тексту функции. Если выражение использует только значения переменных, которые в данной точке программы заведомо содержат одно известное при анализе программы значение, такое выражение может быть вычислено на этапе анализа программы. Если в выражении используется переменная, которая в данной точке программы заведомо является копией какой-то другой переменной, в выражение может быть подставлена исходная переменная.

Статический анализ доменов (domain analysis) является расширением алгоритма продвижения констант. Он позволяет определить множество значений, которые может принимать данная

переменная в данной точке программы, если это множество не велико.

Статический слайсинг (slicing) – это построение «сокращённой» программы, из которой удалён весь код, не влияющий на вычисление заданной переменной в заданной точке (обратный слайс), но при этом программа остаётся синтаксически и семантически корректной и может быть выполнена. Кроме описанного выше обратного слайсинга разработаны алгоритмы прямого слайсинга. Прямой слайсинг оставляет в программе только те операторы, которые зависят от значения переменной, вычисленного в данной точке программы. Методы слайсинга могут быть полезны при разделении «переплетённых» вычислений, когда одновременно вычисляются две независимые друг от друга величины. Например, в одном цикле может вычисляться скалярное произведение двух векторов, а также минимальный и максимальный элемент каждого вектора, и такие циклы могут быть расщеплены с помощью построения слайсов.

Статистический анализ покрытия базовых блоков программы позволяет установить, выполнялся ли когда-либо при выполнении программы на заданном множестве наборов входных данных заданный базовый блок.

Статистическое сравнение трасс позволяет выявить, одинаковы ли трассы программы, полученные при разных запусках на одном и том же наборе входных данных.

Статистическое построение графа потока управления строит граф потока управления на основании информации о порядке следования базовых блоков на одном наборе или на множестве наборов входных данных.

Динамическое продвижение копий вдоль трасс необходимо для точного межпроцедурного анализа зависимостей по данным на основе трассы выполнения программы. Поскольку трасса выполнения программы, по сути, является одним большим базовым блоком, продвижение копий – несложная задача.

Динамическое выделение мёртвого кода позволяет выявить инструкции программы, которые выполнялись при данном запуске программы, но не оказали никакого влияния на результат работы программы. Если анализируется совокупность запусков программы на множестве наборов входных данных, можно говорить о статистическом выделении мёртвого кода.

Динамический слайсинг оставляет в трассе программы только те инструкции, которые повлияли на вычисление данного значения в данной точке программы (прямой динамический слайсинг), или только те инструкции, на которые повлияло присваивание значения данной переменной в данной точке программы.

Заметим, что о точности динамических методов анализа можно говорить, только если известно полное тестовое покрытие программы (построение полного тестового покрытия – алгоритмически неразрешимая задача). В противном случае статистическое выявление свойств программы не позволяет нам утверждать, что данное свойство справедливо на всех допустимых наборах входных данных. Например, условие if (leap_year(current_data)) всегда будет равно значению «истина», если текущий год високосный, и значению «ложь» в противном случае, однако удаление этого оператора из программы приведёт к её неправильной работе.

Поэтому описанные выше динамические методы не могут применяться в автоматическом инструменте анализа программ. Роль этих методов в том, чтобы привлечь внимание пользователя инструмента анализа программ к особенностям работы программы. В дальнейшем пользователь может изучить «подозрительный» фрагмент кода более детально с применением других инструментов, чтобы подтвердить или опровергнуть выдвинутую гипотезу.

Если непрозрачные предикаты и недостижимый код устраняются только на основании статистического анализа, всегда остаётся возможность, что предикат был существенным (как в примере выше). Чтобы всё же упростить программу, можно, например, вынести предположительно недостижимый код из общего графа потока управления функции в обработчик специального исключения, которое возбуждается каждый раз, когда предикат примет значение, отличное от обычного. С одной стороны, граф потока управления и потока данных основной программы в результате упростится, а с другой стороны, программа сохранит свою функциональность.

В данном разделе мы сопоставим методы запутывания, описанные в разделе 2, и методы анализа программ, рассмотренные в разделе 3. На основании этого сопоставления вводится новая мера устойчивости запутывающих преобразований, а именно:

Таблица 1. Методы запутывания программ и методы, которые могут применятся для их анализа.

Метод запутывания	**Метод распутывания**
искажение имён переменных	переименование переменных
использование специфических языковых конструкций	упрощение специфических языковых конструкций
развёртка цикла	визуализация графа потока управления функции для выделения потенциальных кандидатов на свертку в цикл; выявление индуктивной переменной, что требует (интерактивного) сравнения базовых блоков и (интерактивных) эквивалентных преобразований выражений, причём при таких преобразованиях выражение может даже (незначительно) усложняться; свёртка цикла
использование уникальных переменных в базовых блоках	продвижение копий (статическое и статистическое), минимизация количества используемых переменных
введение детерминированного диспетчера	статистическое восстановление графа потока управления
введение недетерминированного диспетчера	сравнение трасс, полученных на одном и том же наборе входных данных
переплетение кода нескольких базовых блоков в один запутанный базовый блок	статистическое устранение мёртвого кода
введение непрозрачных предикатов	статистический анализ покрытия для выявления потенциальных непрозрачных предикатов, поиск по образцам известных непрозрачных предикатов, алгебраическое упрощение, доказательство теорем
все методы	свёртка констант, продвижение констант, продвижение копий, статическое устранение мёртвого кода – могут выполняться после каждого шага распутывающих преобразований

Таблица 2. Классификация запутывающих преобразований

Преобразование	**Сложность распутывания** (необходимый тип анализа)	**Автоматизируемость** (тип распутывателя)
Удаление комментариев	Одностороннее преобразование	hh
Переформатирование программы	Синтаксический	автомат.
Удаление отладочной информации	Одностороннее преобразование	hh
Изменение имён идентификаторов	Синтаксический	полуавтомат.
Языково-специфические преобразования	Синтаксический	автомат.
Открытая вставка функций	Одностороннее преобразование	поддерж.
Вынос группы операторов	Синтаксический	автомат.
Непрозрачные предикаты и выражения	Синтаксический – статистический (зависит от вида предиката)	автомат. – поддерж.
Внесение недостижимого кода	Зависит от стойкости непрозрачных предикатов	автомат., полуавтомат.
Внесение мёртвого кода	Синтаксический – статистический	автомат., полуавтомат.
Внесение избыточного кода	Синтаксический – статистический	автомат., поддерж.
Внесение несводимости в граф	Статический, но зависит от стойкости непрозрачных предикатов	автомат., полуавтомат.
Устранение библиотечных вызовов	Одностороннее преобразование	поддерж.
Переплетение функций	Статический – статистический	автомат. – поддерж.
Клонирование функций или базовых блоков	Статический	автомат. – поддерж.
Развёртка циклов	Одностороннее преобразование	поддерж.
Разложение циклов	Статический	автомат. – поддерж.
Введение диспетчера	Статический – статистический	автомат., полуавтомат.
Локализация переменных в базовом блоке	Статический – статистическийе	автомат., полуавтомат.
Расширение области действия переменных	Статический – статистический	автомат., полуавтомат.

Запутывающее преобразование называется устойчивым относительно некоторого класса методов анализа программ, если методы этого класса не позволяют надёжно раскрыть данное запутывающее преобразование Перечисление некоторых методов запутывания программ с точки зрения методов их возможного распутывания дано в таблице 1.

В таблице 2 приведена классификация методов запутывания по отношению к требуемым методам анализа программ. В третьем столбце таблицы указана степень, в которой возможно автоматическое распутывание. Степень автоматизма оценивается по следующей шкале: автоматический – поиск в программе запутанных фрагментов и их распутывание возможны полностью автоматически; полуавтоматический – поиск в программе подозрительных фрагментов и их распутывание по отдельности выполняются автоматически, но пользователь должен подтвердить применение распутывающего преобразования; поддерживаемый – поиск в программе запутанных фрагментов и применение распутывающих преобразований требуют существенного участия человека, но процесс может быть поддержан специальными инструментальными средствами; неавтоматизируемый – автоматизация выполнения распутывающего преобразования принципиально затруднена.

Для некоторых видов запутывающих преобразований требуемые инструменты (синтаксические, статические, статистические) зависят от того, каким способом было реализовано преобразование. Например, непрозрачные предикаты могут быть самого разного вида, от простейших if(0), до очень сложных. Для анализа и устранения простейших непрозрачных предикатов достаточно инструментов уровня синтаксического анализа, которые работают автоматически, а для устранения сложных непрозрачных предикатов требуется статистический анализ, либо сложные инструменты, такие как полуавтоматический доказатель теорем. Поэтому некоторые ячейки таблицы содержат несколько необходимых видов анализа или несколько оценок автоматизируемости.

7 Защита информаци систем на этапах жизненного цикла

7.1 Жизненный цикл программного обеспечения компьютерных систем

Необходимость определения этапов жизненного цикла (ЖЦ) ПО обусловлена стремлением разработчиков к повышению качества ПО за счет оптимального управления разработкой и использования разнообразных механизмов контроля качества на каждом этапе, начиная от постановки задачи и заканчивая авторским сопровождением ПО. Наиболее общим представлением жизненного цикла ПО является модель в виде базовых этапов – процессов, к которым относятся:

- системный анализ и обоснование требований к ПО;
- предварительное (эскизное) и детальное (техническое) проектирование ПО;
- разработка программных компонент, их комплексирование и отладка ПО в целом;
- испытания, опытная эксплуатация и тиражирование ПО;
- регулярная эксплуатация ПО, поддержка эксплуатации и анализ результатов;
- сопровождение ПО, его модификация и совершенствование, создание новых версий.

Данная модель является общепринятой и соответствует как отечественным нормативным документам в области разработки программного обеспечения, так и зарубежным. С точки зрения обеспечения технологической безопасности целесообразно рассмотреть более подробно особенности представления этапов ЖЦ в зарубежных моделях, так как именно зарубежные программные средства являются наиболее вероятным носителем программных дефектов диверсионного типа.

Графическое представление моделей ЖЦ позволяет наглядно выделить их особенности и некоторые свойства процессов. Первоначально была создана каскадная модель ЖЦ (рисунок 15), в которой крупные этапы начинались друг за другом с использованием результатов предыдущих работ.

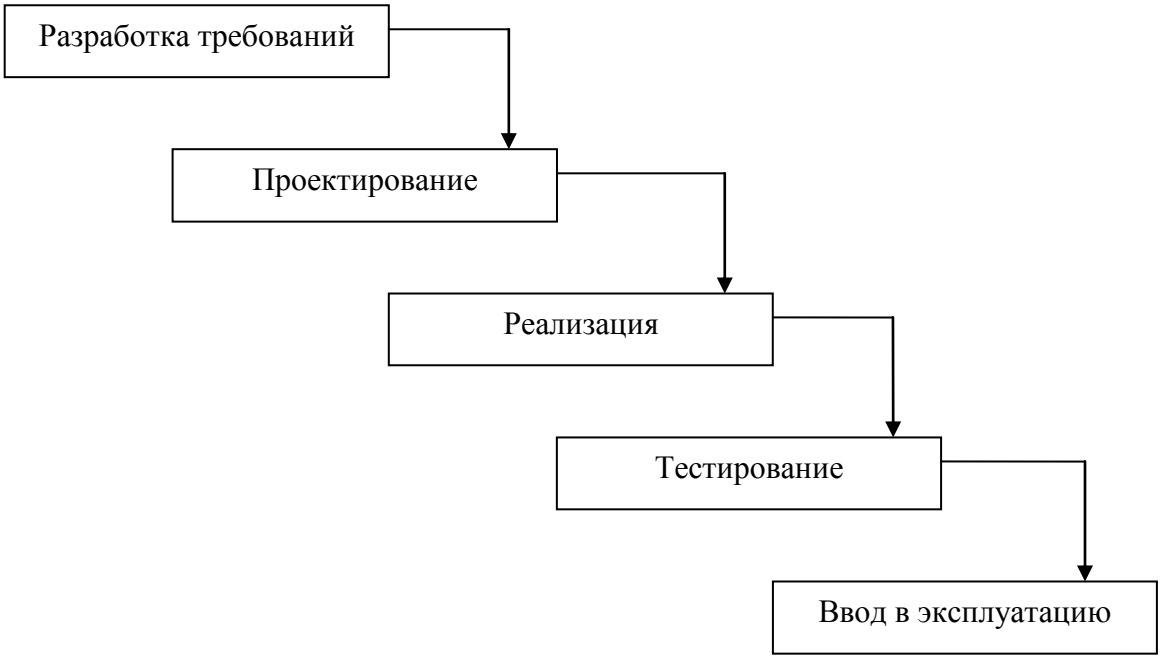

Рисунок 15 – Каскадная модель жизненного цикла информационной системы

Наиболее специфической является спиралевидная модель ЖЦ (рисунок 16). В этой модели внимание концентрируется на итерационном процессе начальных этапов проектирования. На этих этапах последовательно создаются концепции, спецификации требований, предварительный и детальный проект. На каждом витке уточняется содержание работ и концентрируется облик создаваемого ПО.

В настоящее время так же применяется Поэтапная модель с промежуточным контролем (рисунок 17). Разработка ИС ведется итерациями с циклами обратной связи между этапами. Межэтапные корректировки позволяют учитывать реально существующее взаимовлияние результатов разработки на различных этапах; время жизни каждого из этапов растягивается на весь период разработки.

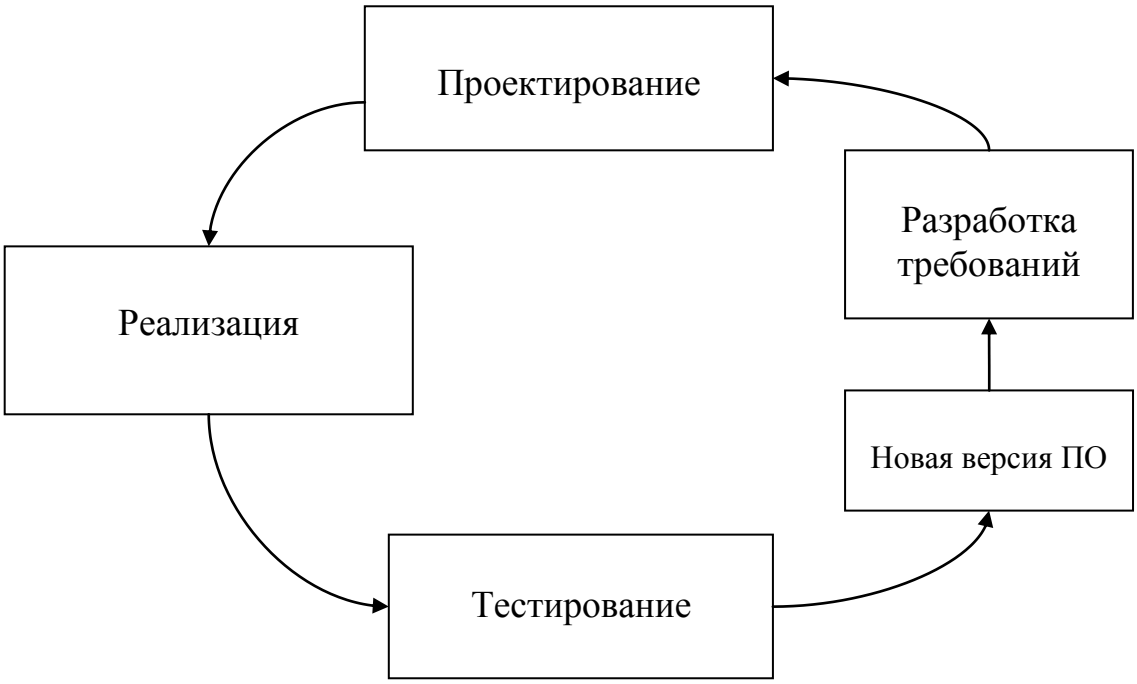

Рисунок 16 – Спиралевидная модель жизненного цикла информационной системы

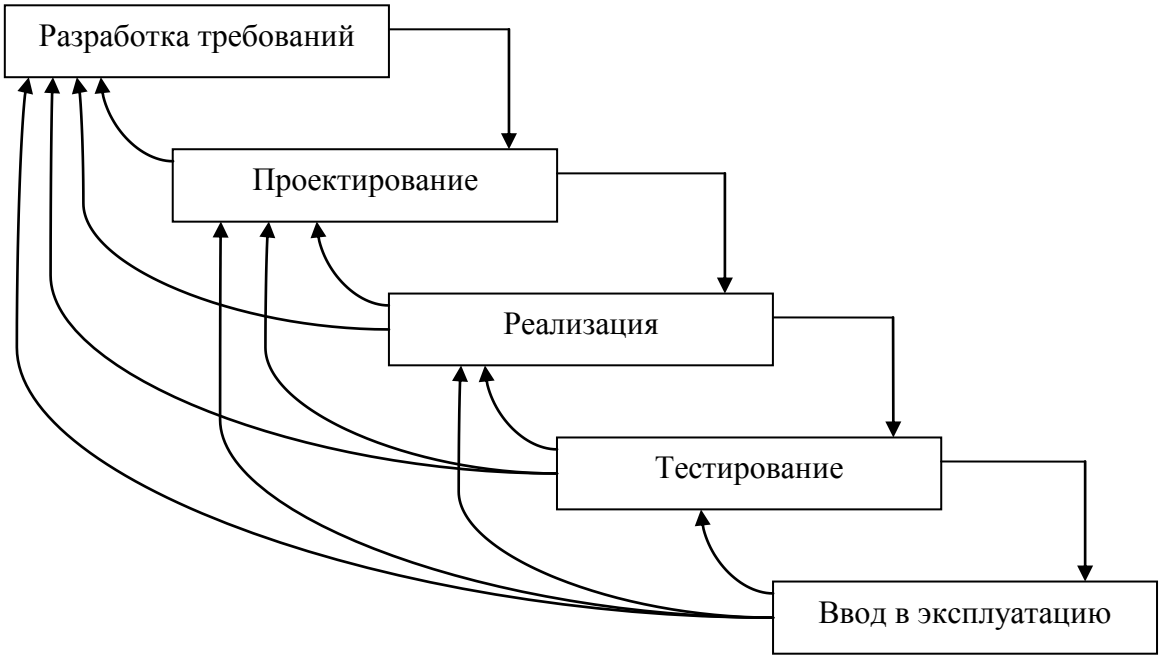

Рисунок 17 – Поэтапная модель ЖЦ ПИ с промежуточным контролем

7.2 Модель угроз и принципы обеспечения безопасности программного обеспечения

Использование при создании программного обеспечения КС сложных операционных систем, инструментальных средств разработки ПО, импортного производства, увеличивает потенциальную возможность внедрения в программы преднамеренных дефектов диверсионного типа. Помимо этого, при создании целевого программного обеспечения всегда необходимо исходить из возможности наличия в коллективе разработчиков программистов – злоумышленников, которые в силу тех или иных причин могут внести в разрабатываемые программы разрушающие программные средства (РПС).

Характерным свойством РПС в данном случае является возможность внезапного и незаметного нарушения или полного вывода из строя КС. Функционирование РПС реализуется в рамках модели угроз безопасности ПО. Один из возможных подходов к созданию модели технологической безопасности ПО АСУ может основываться на обобщенной концепции технологической безопасности компьютерной инфосферы, которая определяет методологический базис, направленный на решение, в том числе, следующих основных задач:

- создания теоретических основ для практического решения проблемы технологической безопасности ПО;
- создания безопасных информационных технологий;
- развертывания системы контроля технологической безопасности компьютерной инфосферы.

Модель угроз технологической безопасности ПО должна представлять собой официально принятый нормативный документ, которым должен руководствоваться заказчики и разработчики программных комплексов.

Модель угроз должна включать:

- полный реестр типов возможных программных закладок;
- описание наиболее технологически уязвимых мест компьютерных систем (с точки зрения важности и наличия условий для скрытого внедрения программных закладок);
- описание мест и технологические карты разработки программных средств, а также критических этапов, при которых наиболее вероятно скрытое внедрение программных закладок;

– реконструкцию замысла структур, имеющих своей целью внедрение в ПО заданного типа (класса, вида) программных закладок диверсионного типа;
– психологический портрет потенциального диверсанта в компьютерных системах.

В указанной Концепции также оговариваются необходимость содержания в качестве приложения банка данных о выявленных программных закладках и описания связанных с их обнаружением обстоятельств, а также необходимость периодического уточнения и совершенствования модели на основе анализа статистических данных и результатов теоретических исследований.

На базе утвержденной модели угроз технологической безопасности компьютерной инфосферы, как обобщенного, типового документа должна разрабатываться прикладная модель угроз безопасности для каждого конкретного компонента защищаемого комплекса средств автоматизации КС. В основе этой разработки должна лежать схема угроз, типовой вид которой применительно к ПО КС представлен на рисунке 18.

Наполнение модели технологической безопасности ПО должно включать в себя следующие элементы: матрицу чувствительности КС к «вариациям» ПО (то есть к появлению искажений), энтропийный портрет ПО (то есть описание «темных» запутанных участков ПО), реестр камуфлирующих условий для конкретного ПО, справочные данные о разработчиках и реальный (либо реконструированный) замысел злоумышленников по поражению этого ПО.

Возможные деструктивные действия на этапах разработки программ:

1. проектирование

– Проектные решения:

Злоумышленный выбор нерациональных алгоритмов работы. Облегчение внесения закладок и затруднение их обнаружения. Внедрение злоумышленников в коллективы, разрабатывающие наиболее ответственные части ПО.

– Используемые информационные технологии:

Внедрение злоумышленников, в совершенстве знающих «слабые» места и особенности используемых технологий. Внедрение информационных технологий или их элементов, содержащих программные закладки. Внедрение неоптимальных информационных технологий.

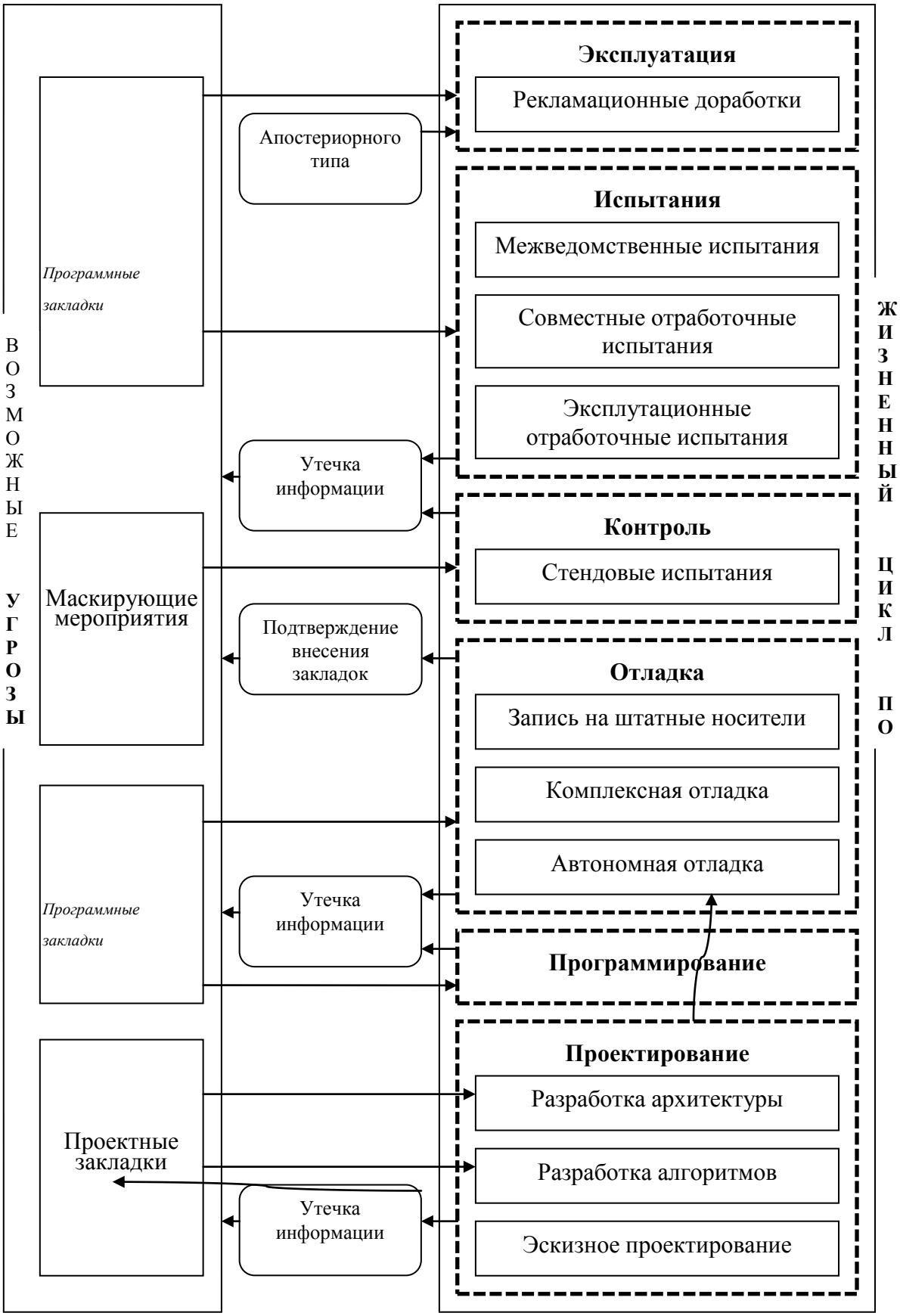

Рисунок 18 – Схема угроз на этапах жизненного цикла ПО

– Используемые аппаратно-технические средства:

Поставка вычислительных средств, содержащих программные, аппаратные или программно-аппаратные закладки. Поставка вычислительных средств с низкими реальными характеристиками. Поставка вычислительных средств, имеющих высокий уровень экологической опасности. Задачи коллективов разработчиков и их персональный состав. Внедрение злоумышленников в коллективы разработчиков программных и аппаратных средств. Вербовка сотрудников путем подкупа, шантажа и т.п.

2. Кодирование

– Архитектура программной системы, взаимодействие ее с внешней средой и взаимодействие подпрограмм программной системы:

Доступ к «чужим» подпрограммам и данным. Нерациональная организация вычислительного процесса. Организация динамически формируемых команд или параллельных вычислительных процессов. Организация переадресации команд, запись злоумышленной информации в используемые программной системой или другими программами ячейки памяти.

– Функции и назначение кодируемой части программной системы, взаимодействие этой части с другими подпрограммами:

Формирование программной закладки, воздействующей на другие части программной системы. Организация замаскированного спускового механизма программной закладки. Формирование программной закладки, изменяющей структуру программной системы.

– Технология записи программного обеспечения и исходных данных:

Поставка программного обеспечения и технических средств со встроенными дефектами.

3. Отладка и испытания

– Назначение, функционирование, архитектура программной системы:

Встраивание программной закладки как в отдельные подпрограммы, так и в управляющую программу программной системы. Формирование программной закладки с динамически формируемыми командами. Организация переадресации отдельных команд программной системы.

– Сведения о процессе испытаний (набор тестовых данных, используемые вычислительные средства, подразделения и лица, проводящие испытания, используемые модели):

Формирование набора тестовых данных, не позволяющих выявить программную закладку. Поставка вычислительных средств, содержащих программные, аппаратные или программно-аппаратные закладки. Формирование программной закладки, не обнаруживаемой с помощью используемой модели объекта в силу ее неадекватности описываемому объекту. Вербовка сотрудников коллектива, проводящих испытания.

4. Контроль

Формирование спускового механизма программной закладки, не включающего ее при контроле на безопасность. Маскировка программной закладки путем внесения в программную систему ложных «непреднамеренных» дефектов. Формирование программной закладки в ветвях программной системы, не проверяемых при контроле. Формирование «вирусных» программ, не позволяющих выявить их внедрение в программную систему путем контрольного суммирования. Поставка программного обеспечения и вычислительной техники, содержащих программные, аппаратные и программно-аппаратные закладки.

5. Эксплуатация

– Сведения о персональном составе контролирующего подразделения и испытываемых программных систем**ах:**

Внедрение злоумышленников в контролирующее подразделение. Вербовка сотрудников контролирующего подразделения. Сбор информации о испытываемой программной системе.

– Сведения об обнаруженных при контроле программных закладках**:**

Разработка новых программных закладок при доработке программной системы. Сведения об обнаруженных незлоумышленных дефектах и программных закладках. Сведения о доработках программной системы и подразделениях, их осуществляющих. Сведения о среде функционирования программной системы и ее изменениях. Сведения о функционировании программной системы, доступе к ее загрузочному модулю и исходным данным, алгоритмах проверки сохранности программной системы и данных.

7.3 Элементы модели угроз эксплуатационной безопасности ПО

Анализ угроз эксплуатационной безопасности ПО КС позволяет, разделить их на два типа: случайные и преднамеренные, причем последние подразделяются на активные и пассивные. Активные угрозы направлены на изменение технологически обусловленных алгоритмов, программ функциональных преобразований или информации, над которой эти преобразования осуществляются. Пассивные угрозы ориентированы на нарушение безопасности информационных технологий без реализации таких модификаций.

Вариант общей структуры набора потенциальных угроз безопасности информации и ПО на этапе эксплуатации КС приведен в таблице 3.

Рассмотрим основное содержание данной таблицы. Угрозы, носящие случайный характер и связанные с отказами, сбоями аппаратуры, ошибками операторов и т.п. предполагают нарушение заданного собственником информации алгоритма, программы ее обработки или искажение содержания этой информации. Субъективный фактор появления таких угроз обусловлен ограниченной надежностью работы человека и проявляется в виде ошибок (дефектов) в выполнении операций формализации алгоритма функциональных преобразований или описания алгоритма на некотором языке, понятном вычислительной системе.

Угрозы, носящие злоумышленный характер вызваны, как правило, преднамеренным желанием субъекта осуществить несанкционированные изменения с целью нарушения корректного выполнения преобразований, достоверности и целостности данных, которое проявляется в искажениях их содержания или структуры, а также с целью нарушения функционирования технических средств в процессе реализации функциональных преобразований и изменения конструктива вычислительных систем и систем телекоммуникаций.

На основании анализа уязвимых мест и после составления полного перечня угроз для данного конкретного объекта информационной защиты, например, в виде указанной таблицы, необходимо осуществить переход к неформализованному или формализованному описанию модели угроз эксплуатационной безопасности ПО КС.

Таблица 3 – Вариант общей структуры набора потенциальных угроз безопасности информации и ПО на этапе эксплуатации КС

Угрозы нарушения безопасности ПО	Несанкционированные действия		
	Случайные	Преднамеренные	
		Пассивные	Активные
Прямые	невыявленные ошибки программного обеспечения КС; отказы и сбои технических средств КС; ошибки операторов; неисправность средств шифрования; скачки электропитания на технических средствах; старение носителей информации; разрушение информации под воздействием физических факторов (аварии и т.п.).	маскировка несанкционированных запросов под запросы ОС; обход программ разграничения доступа; чтение конфиденциальных данных из источников информации; подключение к каналам связи с целью получения информации («подслушивание» и/или «ретрансляция»); при анализе трафика; использование терминалов и ЭВМ других операторов; намеренный вызов случайных факторов.	включение в программы РПС, выполняющих функции нарушения целостности и конфиденциальности информации и ПО; ввод новых программ, выполняющих функции нарушения безопасности ПО; незаконное применение ключей разграничения доступа; обход программ разграничения доступа; вывод из строя подсистемы регистрации и учета; уничтожение ключей шифрования и паролей; подключение к каналам связи с целью модификации, уничтожения, задержки и переупорядочивания данных; вывод из строя элементов физических средств защиты информации КС; намеренный вызов случайных факторов.
Косвенные	нарушение пропускного режима и режима секретности; естественные потенциальные поля; помехи и т.п.	перехват ЭМИ от технических средств; хищение производственных отходов (распечаток); визуальный канал; подслушивающие устройства; дистанционное фотографирование и т.п.	помехи; отключение электропитания; намеренный вызов случайных факторов.

Такая модель, в свою очередь, должна соотноситься (или даже являться составной частью) обобщенной модели обеспечения безопасности информации и ПО объекта защиты.

К неформализованному описанию модели угроз приходится прибегать в том случае, когда структура, состав и функциональная наполненность компьютерных системы управления носят многоуровневый, сложный, распределенный характер, а действия потенциального нарушителя информационных и функциональных ресурсов трудно поддаются формализации.

После окончательного синтеза модели угроз разрабатываются практические рекомендации и методики по ее использованию для конкретного объекта информационной защиты, а также механизмы оценки адекватности модели и реальной информационной ситуации и оценки эффективности ее применения при эксплуатации КС.

Таким образом, разработка моделей угроз безопасности программного обеспечения КС, являясь одним из важных этапов комплексного решения проблемы обеспечения безопасности информационных технологий, на этапе создания КС отличается от разработки таких моделей для этапа их эксплуатации.

Принципиальное различие подходов к синтезу моделей угроз технологической и эксплуатационной безопасности ПО заключается в различных мотивах поведения потенциального нарушителя информационных ресурсов, принципах, методах и средствах воздействия на ПО на различных этапах его жизненного цикла.

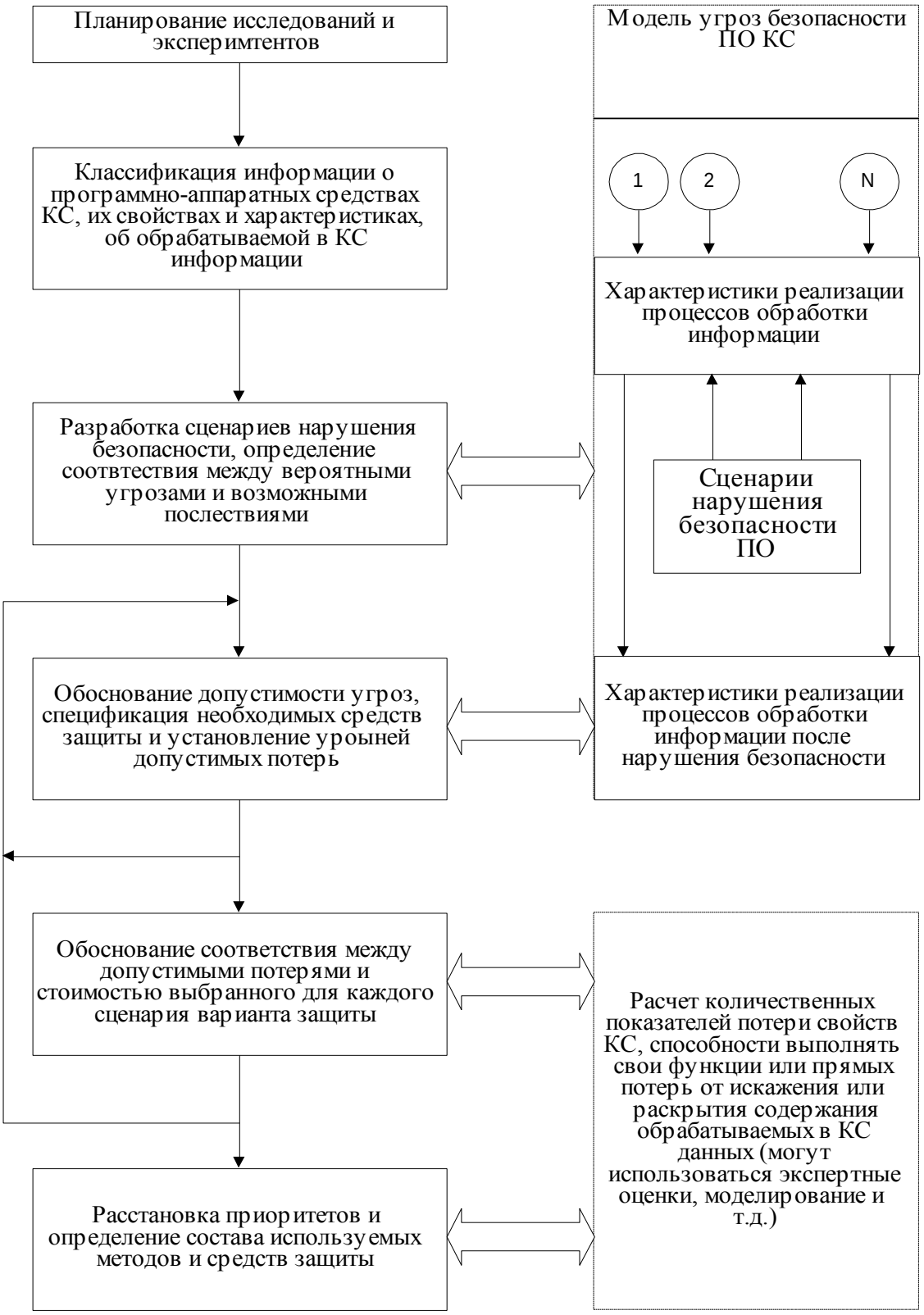

Рисунок 19 – Неформализованное описание модели угроз безопасности ПО на этапе исследований попыток несанкционированных действий в отношении информационных ресурсов КС

7.4 Основные принципы обеспечения безопасности ПО

В качестве объекта обеспечения технологической и эксплуатационной безопасности ПО рассматривается вся совокупность его компонентов в рамках конкретной КС. В качестве доминирующей должна использоваться стратегия сквозного тотального контроля технологического и эксплуатационного этапов жизненного цикла компонентов ПО. Совокупность мероприятий по обеспечению технологической и эксплуатационной безопасности компонентов ПО должна носить конфиденциальный характер. Необходимо обеспечить постоянный, комплексный и действенный контроль за деятельностью разработчиков и пользователей компонентов. Кроме общих принципов, обычно необходимо конкретизировать принципы обеспечения безопасности ПО на каждом этапе его жизненного цикла. Далее приводятся один из вариантов разработки таких принципов.

1. Принципы обеспечения технологической безопасности при обосновании, планировании работ и проектном анализе ПО.

- Комплексности обеспечения безопасности ПО, предполагающей рассмотрение проблемы безопасности информационно – вычислительных процессов с учетом всех структур КС, возможных каналов утечки информации и несанкционированного доступа к ней, времени и условий их возникновения, комплексного применения организационных и технических мероприятий.
- Планируемости применения средств безопасности программ, предполагающей перенос акцента на совместное системное проектирование ПО и средств его безопасности, планирование их использования в предполагаемых условиях эксплуатации.
- Обоснованности средств обеспечения безопасности ПО, заключающейся в глубоком научно-обоснованном подходе к принятию проектных решений по оценке степени безопасности, прогнозированию угроз безопасности, всесторонней априорной оценке показателей средств защиты.
- Достаточности безопасности программ, отражающей необходимость поиска наиболее эффективных и надежных мер безопасности при одновременной минимизации их стоимости.

- Гибкости управления защитой программ, требующей от системы контроля и управления обеспечением информационной безопасности ПО способности к диагностированию, опережающей нейтрализации, оперативному и эффективному устранению возникающих угроз в условиях резких изменений обстановки информационной борьбы.
- Заблаговременности разработки средств обеспечения безопасности и контроля производства ПО, заключающейся в предупредительном характере мер обеспечения технологической безопасности работ в интересах недопущения снижения эффективности системы безопасности процесса создания ПО.
- Документируемости технологии создания программ, подразумевающей разработку пакета нормативно-технических документов по контролю программных средств на наличие преднамеренных дефектов.

2. Принципы достижения технологической безопасности ПО в процессе его разработки.

- Регламентации технологических этапов разработки ПО, включающей упорядоченные фазы промежуточного контроля, спецификацию программных модулей и стандартизацию функций и формата представления данных.
- Автоматизации средств контроля управляющих и вычислительных программ на наличие дефектов, создания типовой общей информационной базы алгоритмов, исходных текстов и программных средств, позволяющих выявлять преднамеренные программные дефекты.
- Последовательной многоуровневой фильтрации программных модулей в процессе их создания с применением функционального дублирования разработок и поэтапного контроля.
- Типизации алгоритмов, программ и средств информационной безопасности, обеспечивающей информационную, технологическую и программную совместимость, на основе максимальной их унификации по всем компонентам и интерфейсам.
- Централизованного управления базами данных проектов ПО и администрирование технологии их разработки с жестким разграничением функций, ограничением доступа в соответствии со средствами диагностики, контроля и защиты.

- Блокирования несанкционированного доступа соисполнителей и абонентов государственных сетей связи, подключенных к стендам для разработки программ.
- Статистического учета и ведения системных журналов о всех процессах разработки ПО с целью контроля технологической безопасности.
- Использования только сертифицированных и выбранных в качестве единых инструментальных средств разработки программ для новых технологий обработки информации и перспективных архитектур вычислительных систем.

3. Принципы обеспечения технологической безопасности на этапах стендовых и приемо-сдаточных испытаний.

- Тестирования ПО на основе разработки комплексов тестов, параметризуемых на конкретные классы программ с возможностью функционального и статистического контроля в широком диапазоне изменения входных и выходных данных.
- Проведения натурных испытаний программ при экстремальных нагрузках с имитацией воздействия активных дефектов.
- Осуществления «фильтрации» программных комплексов с целью выявления возможных преднамеренных дефектов определенного назначения на базе создания моделей угроз и соответствующих сканирующих программных средств.
- Разработки и экспериментальной отработки средств верификации программных изделий.
- Проведения стендовых испытаний ПО для определения непреднамеренных программных ошибок проектирования и ошибок разработчика, приводящих к невыполнению целевых функций программ, а также выявление потенциально «узких» мест в программных средствах для разрушительного воздействия.
- Отработки средств защиты от несанкционированного воздействия нарушителей на ПО.
- Сертификации программных изделий АСУ по требованиям безопасности с выпуском сертификата соответствия этого изделия требованиям технического задания.

4. Принципы обеспечения безопасности при эксплуатации программного обеспечения.

- Сохранения и ограничения доступа к эталонам программных средств, недопущение внесения изменений в них.
- Профилактического выборочного тестирования и полного сканирования программных средств на наличие преднамеренных дефектов.
- Идентификации ПО на момент ввода его в эксплуатацию в соответствии с предполагаемыми угрозами безопасности ПО и его контроль.
- Обеспечения модификации программных изделий во время их эксплуатации путем замены отдельных модулей без изменения общей структуры и связей с другими модулями.
- Строгого учета и каталогизации всех сопровождаемых программных средств, а также собираемой, обрабатываемой и хранимой информации.
- Статистического анализа информации обо всех процессах, рабочих операциях, отступлениях от режимов штатного функционирования ПО.
- Гибкого применения дополнительных средств защиты ПО в случае выявления новых, непрогнозируемых угроз информационной безопасности.

Заключение

Рассмотрев проблему защиты информации в информационных системах, можно отметить, что для достижения удачных решений проблем защиты информации необходимо понять, тот факт, что обеспечение безопасности информации – это **система мероприятий,** как технических, так и организационных. Эта система определяется секретностью защищаемой информации, характером угроз и наличием средств. Нельзя опускать ни технические меры, ни организационные способы защиты; каждая мера дополняет другую, и недостаток или отсутствие любого способа приведет к нарушению секретности системы. Оценку безопасности системы не достаточно проводить единожды. Вопросы безопасности следует пересматривать периодически, и особенно во всех тех случаях, когда применяется новая аппаратура или предлагаются новые виды обслуживания.

Способы атак, а также методы защиты от них совершенствуются, а это значит, что нужно постоянно уделять большое внимание безопасности хранимой информации. Незащищенная ИС может стать легкой добычей не только для профессиональных взломщиков, но даже для новичков или просто посторонних заинтересованных людей.

Какой бы деятельностью не занималась организация и каким бы бюджетом не располагала, защите своей информации она должна уделять особое внимание.

Литература

1. **Чернов А. В.** Интегрированная среда для исследования «обфускации» программ [Электронный ресурс]. – Электрон. текстовые дан. – Доклад на конференции, посвящённой 90-летию со дня рождения А.А.Ляпунова. Россия, Новосибирск, 8-11.10.2001г. [Режим доступа: http://www.ict.nsc.ru/ws/Lyap2001/2350/]

2. **Cifuentes C., Gough K. J.** Decompilation of Binary Programs [Электронный ресурс]. – Электрон. текстовые дан. – Technical report FIT-TR-1994-03. Queensland University of Technology, 1994. [Режим доступа: http://www.fit.qut.edu.au/TR/techreports/FIT-TR-94-03.ps]

3. **Collberg C., Thomborson C., Low D.** A Taxonomy of Obfuscating Transformations [Электронный ресурс]. – Электрон. текстовые дан. – Department of Computer Science, The University of Auckland, 1997. [Режим доступа: http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborsonLow97a]

4. **Collberg C., Thomborson C.** On the Limits of Software Watermarking [Электронный ресурс]. – Электрон. текстовые дан. – Technical Report №164. Department of Computer Science, The University of Auckland, 1998. [Режим доступа: http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborson98e]

5. **Collberg C., Thomborson C.** Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection [Электронный ресурс]. – Электрон. текстовые дан. – Technical Report, Department of Computer Science, University of Arizona, 2000. [Режим доступа: http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborson2000a]

6. **Hachez G., Vasserot C.** State of the Art in Software Protection [Электронный ресурс]. – Электрон. текстовые дан. – Project FILIGRANE (Flexible IPR for Software Agent Reliance) deliverable/V2. [Режим доступа: http://www.dice.ucl.ac.be/crypto/filigrane/External/d21.pdf]

7. The International Obfuscated C Code Contest [Электронный ресурс]. – Электрон. текстовые дан., [Режим доступа: http://www.ioccc.org/]

8. **Lai H.** A comparative survey of Java obfuscators available on the Internet. [Электронный ресурс]. – Электрон. текстовые дан., [Режим доступа: http://www.cs.auckland.ac.nz/~cthombor/Students/hlai]

9. **Low D.** Java Control Flow Obfuscation. MSc Thesis [Электронный ресурс]. – Электрон. текстовые дан. – University of Auckland, 1998. [Режим доступа: http://www.cs.arizona.edu/~collberg/Research/Students/DouglasLow/thesis.ps]

10. **Walle E.** Methodology and Applications of Program Code Obfuscation. Faculty of Computer and Electrical Engineering [Электронный ресурс]. – Электрон. текстовые дан. – University of Waterloo, 2001. [Режим доступа: http://walle.dyndns.org/morass/misc/wtr3b.doc]

11. **Wang C.,** Security Architecture for Survivability Mechanisms. PhD Thesis [Электронный ресурс]. – Электрон. текстовые дан. – Department of Computer Science, University of Virginia, 2000. [Режим доступа: http://www.cs.virginia.edu/~survive/pub/wangthesis.pdf]

12. **Wang C., Davidson J., Hill J., Knight J.** Protection of Software-based Survivability Mechanisms [Электронный ресурс]. – Электрон. текстовые дан. – Department of Computer Science, University of Virginia, 2001. [Режим доступа: http://www.cs.virginia.edu/~jck/publications/dsn_distribute.pdf]

13. **Козленко Л.** Информационная безопасность в современных системах управления базами данных [Электронный ресурс]. – Электрон. текстовые дан. – ЦитФорум [Режим доступа: http://www.citforum.ru/security/articles/safe_db]

14. **Портнов В.А.** количественная и качественная оценка риска в управлении промышленной безопасностью [Текст]. В.А. Портнов, Н.А. Махутов, И.Б. Зеленов. – М.: ОИ/ВИНИТИ, 2003. Вып. 6, – 100 с.

15. Федеральный закон «О техническом регулировании» № 184-ФЗ [Текст]

16. Information technology – Security techniques – Security assessment of operational systems [Текст] – ISO/IEC 2nd PDTR 19791, 17.12.2004

17. Концепция защиты СВТ и АС от НСД к информации [Текст] – М: Гостехкомиссия России. Руководящий документ, 1992.

18. Автоматизированные системы. Защита от несанкционированного доступа к информации. Классификация

автоматизированных систем и требования по защите информации [Текст] – М: Гостехкомиссия России. Руководящий документ, 1992.

19. Средства вычислительной техники. Защита от несанкционированного доступа к информации. Показатели защищенности от несанкционированного доступа к информации [Текст] – М: Гостехкомиссия России. Руководящий документ, 1992.

20. Средства вычислительной техники. Межсетевые экраны. Защита от несанкционированного доступа к информации. Показатели защищенности от несанкционированного доступа к информации [Текст] – М: Гостехкомиссия России. Руководящий документ, 1997.

21. **Галатенко В.А.** Стандарты информационной безопасности [Текст] / Под ред. академика РАН В.Б. Бетелина. – М.: ИНТУИТ.РУ, 2005. – 264 с. – ISBN: 5-9556-0053-1

22. **Бетелин В.Б.,** Информационная (компьютерная) безопасность с точки зрения технологии программирования [Текст] / В. Бетелин, В. Галатенко Труды 4-й Ежегодной конференции консорциума ПрМ «Построение стратегического сообщества через образование и науку» – М.: МГУ, 2001. – 250 с.

23. Безопасность баз данных [Электронный ресурс]. – Электрон. текстовые дан. – [Режим доступа http://www.osp.ru/dbms/1997/01/78_print.htm]

24. **Семьянов П.** Анализ средств противодействия исследованию программного обеспечения [Электронный ресурс]. – Электрон. текстовые дан. / П. Семьянов, Д. Зегжда, [Режим доступа: http://www.ssl.stu.neva.ru/psw/publications/research.txt]

25. **Саймон А.** СУБД №1/97. Глава 21. Безопасность баз данных [Электронный ресурс]. – Электрон. текстовые дан. – [Режим доступа: http://www.osp.ru/dbms/1997/01/78.htm]

26. **Дейт К.Дж.** Введение в системы баз данных [Электронный ресурс]. – Электрон. текстовые дан. – [Режим доступа: http://tothbenedek.hu/ed2kstats/ed2k?hash=bf3d232869501ea544c09f796ca01e6e]

27. **Bell D.E.,** Secure Computer Systems: Mathematical foundations and model [Текст] / L. La Padula, D. Bell Report ESD-TR-73-278. Vol. 1-3. Mitre Corp. Bedford. Mass. Nov.1973 – June 1974

28. Положение по организации разработки, изготовления и эксплуатации программных и технических средств защиты секретной информации от НСД в автоматизированных системах и средствах вычислительной техники [Текст]

Учебное издание

Еременко Владимир Тарасович

Фисун Александр Павлович

ПРОГРАММНО-АППАРАТНЫЕ СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

Учебное пособие

Редактор
Технический редактор

Федеральное государственное бюджетное образовательное
Учреждение высшего профессионального образования
«Государственный университет – учебно-научно-
производственный комплекс»

Подписано к печати Формат 60x90 1/16
Усл. печ. л. Тираж экз.
Заказ №

Отпечатано с готового оригинал-макета

кафедра «Электроника, вычислительная техника и
информационная безопасность»
Телефон: 8-9066646161, 8-9065701666

Еременко Владимир Тарасович

Фисун Александр Павлович

ПРОГРАММНО-АППАРАТНЫЕ СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ

Учебное пособие