8. Технологии взлома механизмов защиты. Методы защиты от исследования программ

*Взлом механизмов, основанных на ключевом сравнении. Уязвимость криптографических систем защиты. Надежность цифровой подписи. Атака полным перебором. Эмуляция ключевых дисков.*

*Средства исследования программ. Защита программ от дизассемблирования. Защита программ от работы под контролем отладчика.*

Программное обеспечение стало массовым товаром. Однако его продажа вызывает трудности, которые не встречаются при продаже традиционных товаров. Операция по производству единицы товара, являющегося программой, оказывается очень простой – это операция копирования строки символов, хотя и довольно длинной. Покупка обычного товара экономически выгодна для покупателя, так как затраты на покупку меньше затрат на производство этого товара. Совсем другая ситуация с программами. После покупки программы покупатель сам вполне может продавать ее без участия настоящего производителя. Поэтому продажа программы, всю цену которой составляет исключительно интеллектуальная собственность, должна выполняться способом, отличающимся от продажи традиционных товаров.

Товары с высокой долей интеллектуальной собственности: книги, музыкальные записи, фильмы и т.д. до появления на товарном рынке программ не имели этой особенности – чрезвычайной дешевизны операции копирования. Поэтому техническая проблема обеспечения возможности продажи программ как обычного товара называется защитой от несанкционированного копирования. Продавать незащищенные программы как массовый товар бессмысленно. Первый же покупатель программы, имеющей широкий рынок, будет продавать ее сам (или распространять ее бесплатно), что полностью лишит дохода ее настоящего владельца. Таким образом, защита программ – это проблема, имеющая чисто экономическую основу. Первоначально у разработчиков программ была надежда решить задачу продажи программного обеспечения как техническую проблему. Для защиты программ требуется разработать такие формы программ, которые сделали бы операцию их несанкционированного копирования достаточно трудоемкой, чтобы покупка программы стала экономически оправданной для покупателя.

Идея защиты программы состоит в том, чтобы сделать часть кода программы «недоступным» для анализа пользователями. Принимая возможность осуществления процедуры, обеспечивающей эту функциональную возможность, можно создать программную систему защиты, в которой регистрации лицензии на легальное использование копии программы данному пользователю осуществляется в «недоступной для анализа части» (потому злоумышленник не может мешать проверке лицензии). Задача метода защиты состоит в том, чтобы обеспечивать эту «недоступность для анализа».

Защита программ находится на стыке защиты данных, криптографии, технологии программирования и других дисциплин. Несмотря на название, защита программ включает предположения, связанные с аппаратными средствами и другими аспектами среды выполнения программ.

Для первоначальной классификации способов защиты защищаемую программу вместе с ее окружением удобно рассматривать как некоторую распределенную систему, состоящую из самой программы и ее среды исполнения. Кроме того, модель конечно должна включать злоумышленника, цель которого состоит в том, чтобы преодолеть защиту программы. Для простоты целесообразно включить злоумышленника не явно, а предполагая его наличие в одной из частей распределенной системы. Часть, в которой наличие злоумышленника не предполагается, заслуживает доверия. Способ, который злоумышленник может использовать для преодоления защиты, называется *угрозой* (*threat*), осуществление этого способа – это *атака*. Естественно и без ограничений общности можно предполагать, что злоумышленник один, поэтому модели нападения злоумышленника – угроз атаки на защищаемую систему – сразу можно разделить на два класса (рисунок 8.1): благонадежная программа в неблагонадежной среде и неблагонадежная программа в благонадежной среде – «доверенной среде».

В связи с широким распространением вычислительных сетей становиться очевидной необходимость исследований в области защиты информации в распределенных информационных и вычислительных систем. Моделью такой сети программ являются мультиагентные системы (MAS). Злоумышленник также строит свою мультиагентную систему для проведения распределенной атаки на распределенную систему. Мультиагентные системы являются просто распределенными программами, и на самом верхнем уровне анализа случай распределенной атаки на распределенную систему не отличается от случая обычной программы.

Так как предполагается, что интеллектуальные и другие возможности злоумышленника не ограничены, то набор возможных угроз не предсказуем. Набор известных конструктору защиты угроз всегда неполон, так как новые виды атак изобретаются постоянно. Из-за того, что может быть известно только подмножество возможных угроз, абсолютная защищенность недостижима. На практике проектирование системы, способной противостоять каждой известной угрозе, слишком дорого. Некоторые угрозы имеют близкую к нулю вероятность атаки, осуществление других приводит к относительно незначительным потерям, третьи слишком дорогостоящи для злоумышленника, хотя теоретически возможны. Цель практического метода защиты состоит в том, чтобы затраты на ее проектирования соответствовали стоимости осуществления угроз злоумышленником.

Рисунок 8.1 – Две модели угроз

Набор угроз, которым должна противостоять проектируемая система защиты программы, составляет модель угроз. Эта модель идентифицирует угрозы, принимая во внимание природу возможных типов нападающих. Модели угроз основаны на предположениях о поведении злоумышленника (которые не всегда правильны), предположениях о свойствах внешней среды (которые могут измениться в любое время), юридических законах и субъективных оценках. Стойкость защиты программы зависит от того, насколько близко модель угроз отражает действительность окружающей среды и виды атак, которые фактически осуществимы. Проектирование защиты программ далеко от точной науки, в лучших образцах это искусство. Почти всегда проектирование защиты – это итеративный процесс приспособления к лучшему знанию, к новым угрозам и к изменяющимся условиям.

Враждебная программа в легальной среде исполнения – эта модель угроз интересна тем, что в ней осуществима абсолютная защита. Парирование любой атаки злоумышленника возможно потому, что его возможности существенно ограничены: злоумышленник может нападать только посредством программы.

До недавнего времени в большинстве исследований о защите программ рассматривался только случай защиты целостности благонадежной среды исполнения и ее данных от нападений злонамеренных клиентских программ. Мировоззрение благонадежности среды исполнения – это основа модели защиты языка Java. Среда исполнения Java разработана так, чтобы защититься от нападений потенциально злонамеренных апплетов – распространяемых как документы интернет программ Java, которые могут выполнять веб-браузеры, или зараженных вирусом приложений. Эти нападения осуществимы исключительно в форме разрушения или компрометации локальных данных среды исполнения.

Чтобы защитить себя и свои данные от злонамеренной программы, среда исполнения Java как правило ограничивает действия, которые программе позволено исполнять. В модели защиты Java, среда исполнения выполняет проверку байт-кода приложений, чтобы гарантировать безопасность от нападений потенциально неблагонадежных клиентских приложений. Дополнительно, в коде, загруженном из потенциально неблагонадежного источника (например, в коде апплета) запрещается выполнения некоторых операций, таких как запись в файловую систему. На этом же принципе построена языково-независимая методика изоляции программных ошибок «песочница» (sandbox), которая состоит в том, что клиентский код изменяется так, чтобы предотвратить возможность записывать данные вне разрешенной области. Для решения проблемы защиты от злонамеренной программы используется проверка источника кода (код должен быть снабжен цифровой подписью, позволяющей идентифицировать разработчика) и проверка антивирусными программами. Очевидно, в предположении благонадежности среды исполнения эти меры гарантируют безопасность.

Легальная программа во враждебной среде исполнения – этой модели угроз соответствует реальная ситуация продажи программы. При продаже нельзя исключать случай, когда легальный покупатель недобросовестен. У недобросовестного покупателя среда исполнения программы враждебна по определению. В этом случае возможности злоумышленника ничем не ограничены, и любая система защиты преодолима. Нападение злонамеренной среды исполнения на легальную программу обычно выполняется для кражи интеллектуальной собственности. Обе модели угроз: с прямым (враждебная программа в легальной среде исполнения) и обратным (легальная программа во враждебной среде исполнения) распределениями неблагонадежности были сформулированы уже больше чем 35 лет назад.

Борьба с кражей интеллектуальной собственности состоит в защите от незаконного копирования и использования программного обеспечения и обеспечение возможности доказать права настоящего создателя программы. Задача защиты от незаконного копирования и использования состоит в том, чтобы гарантировать, что только легальные пользователи могут выполнить программу на своем компьютере. Неправомочным использованием считается любое использования программы, кроме предусмотренного ее законным создателем. Для юридического преследования нарушителя авторских прав, необходимо обеспечить возможность доказать по украденной программе факт кражи. Среди методов, используемых для этой цели, наиболее популярным является использование «водяных знаков» (watermarking). «Водяные знаки» представляют собой спрятанное в программном коде объявление об авторском праве.

В обоих методах предполагается что код защиты или водяные знаки – секретный код – являются частью программы «недоступной» для анализа пользователями. Злоумышленнику необходимо понять функции программы, чтобы выделить секретный код. Анализ функций программы входит основной частью в одну из широко распространенных технологий программирования – перепроектирование (reverse engineering). Для предотвращения применения инструментов технологии перепроектирования секретный код можно зашифровать и таким образом сделать часть программы недоступной для анализа.

Однако стандартная криптографическая парадигма – использование известных криптоалгоритмов, секретных ключей и надежных оконечных точек связи – не обеспечивает безопасности в злонамеренной среде исполнения, поскольку секретные ключи непосредственно видимы в программе. Таким образом, сохранение критических ключей в памяти – это основная уязвимость, которую принципиально нельзя устранить.

В распределенных программах применимы одновременно обе модели угроз. Защита сервера – это первая модель, защита клиента – вторая. Чтобы защитить сервер от злонамеренного клиента, достаточно ограничить действия, которые ему позволяются исполнять. К сожалению, нет способов, чтобы защитить клиента от нападения враждебной среды исполнения. Как только клиентская программа оказывается во враждебной среде, злоумышленник, ее контролирующий, может использовать любую мыслимую методику, чтобы извлечь критические данные или просто нарушить целостность клиентской программы. Единственный ограничивающий фактор – это вычислительные ресурсы, которые враждебная среда исполнения может расходовать на анализ клиентского кода. В общем, считается, что полная защита программы от враждебной среды исполнения является недосягаемой целью, однако, некоторая степень защиты все же может быть достигнута.

В ранних работах (1980 г.) по защите программ исследовалась проблема защиты производителей программных средств, в частности защита программного обеспечения от копирования и модификации (легальная программа во враждебной среде исполнения). Для защиты программного обеспечения предлагался набор инструментальных средств, включающий физические (не программные) стойкие к взлому модули и использование криптографических методов. Подход к защите программы в целом состоял в шифровании кода таким образом, что при исполнении программы команды расшифровывались по одной непосредственно перед выполнением.

Около 1985 г. предлагаются аппаратные устройства защиты (например, аппаратные ключи), методы создания ключевых дискет (магнитный и физический), методы предотвращения анализа (например, методы противодействия работе отладчика, использование контрольных сумм, шифрование кода) и способы замедления интерактивного динамического анализа. Основное внимание уделяется проблеме предотвращении копирования программными средствами. Обращается внимание на то, что возможности противодействия копированию связаны с предотвращением анализа программы (инструментами технологии обратного проектирования – software reengineering). Успешному снятию защиты путем модификации программы обычно предшествует обратное проектирование.

Можно ожидать, что противник может выполнить динамический анализ программы без обнаружения (например, путем использование внутрисхемных эмуляторов и имитаторов). В этом случае, используя возможность многократно успешно выполнять программу, противник сумеет достичь своей цели. Таким образом, практическая цель защиты должна состоять в том, чтобы сделать такие эксперименты «чрезвычайно трудными». Другая практическая рекомендация этого времени состояла в предложение циклически повторять разработку программы, предлагая пользователям новые версии программного обеспечения, причем старая версия через определенное время должна становиться неработоспособной. Это должно делаться так, чтобы противник не успевал выполнять снятие защиты. Для предотвращения пиратства предлагался путь принудительного обновления и «состаривания» программного обеспечения. Этот подход удовлетворителен, если требуется защита программы на ограниченный период.

В 80-е годы прошлого века предлагается решение проблемы защиты программ от копирования, требующее поддержки шифрования в процессоре (что было нереально, когда оно было предложено 30 лет назад, но осуществимо сегодня). Для затруднения анализа программ, непосредственно связанного с защитой программ, предлагался метод применении полиморфности кода. Эта идея вызвала поток публикаций, развивающих метод, и новых интерпретаций задачи. Основной результат состоял в том, что задачу анализа программы можно сделать алгоритмически сложной, применяя специальные формы управляющих структур. Для этого используются методы «запутывания кода» (code obfuscation) или скрытия функции, дополняемые применением специальных аппаратных средств.

Практически эти методы оказались малоэффективными, так как код защиты обладает признаками (например, вызовы определенных функций), которые невозможно замаскировать.

Благонадежность среды исполнения связана с доверием к операционной системе и аппаратуре компьютера. Так как современные ОС являются мультипрограммными и мультипользовательскими, то доверие должно распространяться на весь комплекс программ, выполняющихся одновременно с рассматриваемой. Любая программа выполняет действия, которые предусмотрены спецификацией, только в доверенной среде. Доверенность означает, что никто специально не пытается нарушить работу программы (входные данные корректны и ресурсов системы для работы достаточно) или «обмануть» её, представившись, например, другим пользователем. Программа выполняет непредсказуемые действия при получении некорректных входных данных или при выполнении в нестандартных условиях (нехватка ресурсов, аппаратные сбои и т.п.). При возникновении нештатной ситуации программа может или допустить выполнение опасных команд, или, наоборот, отказаться выполнять свои функции.

Очевидно, в мультипрограммной ОС ее архитектура должна защищать процессы программ друг от друга. Отсутствие влияния на функционирование процесса достигается путем обеспечении конфиденциальности и целостности всей информации процесса. Абсолютная конфиденциальность информации невозможна хотя бы потому, что сама ОС должна запускать и обслуживать процесс программы. «Безопасная система управляет, с помощью соответствующих средств, доступом к информации, так что только должным образом авторизованные лица или процессы, действующие от их имени, получают право читать, записывать, создавать и удалять информацию». Доверенная система определяется как «система, использующая достаточные аппаратные и программные средства, чтобы обеспечить одновременную обработку информации разной степени секретности группой пользователей без нарушения прав доступа». Оценка благонадежности – степени доверия – основная линия в современной технологии информационной безопасности.

*Уязвимость* (vulnerablity) – это особенность компьютерной системы (аппаратуры или программ), которая делает возможным несанкционированный доступ к информации в обход механизмов безопасности или причинение ущерба компьютерной системе. Самую большую и практически важную группу дефектов, приводящих к появлению уязвимости и внесенных по небрежности, составляют ошибки проверки правильности данных, точнее, недостаточный контроль входных данных перед их использованием. Отсюда видно, что уязвимости возникают с легкостью и существуют в системе практически всегда. Разработка методов не допущения или выявления подобных ошибок считается задачей первостепенной практической важности. Очень долго уязвимости могут оставаться ненайденными, но рано или поздно любая уязвимость в достаточно широко используемой программе обнаруживается.

Под *вторжением* в компьютерную систему понимается любая деятельность, нарушающая *целостность, конфиденциальность* или *доступность данных* системы. Использованная для вторжения уязвимость называется атакой. Для предотвращения атак необходимо решение задачи обнаружения постороннего кода (malicious code detection), связанное с выявлением недекларированных возможностей программных продуктов (в т.ч. «троянских коней»), и с обнаружением и устранением *программ-вирусов*. Задача обнаружения постороннего кода в программе *P* состоит в обнаружении ее неэквивалентности заданному (эталонному) поведению *P*. Как известно, задача проверки функциональной эквивалентности программ относится к числу алгоритмически неразрешимых проблем.

Практический подход к решению задачи обнаружения постороннего кода состоит в том, что считаются известными образцы этого постороннего кода – код программ-вирусов. Поэтому для обнаружения вирусов можно выделить характерный фрагмент кода, который может служить признаком (сигнатурой) вируса, и затем проводить поиск сигнатуры вируса в коде программы, используя методы поиска заданного шаблона в тексте (pattern matching). Сигнатурные методы не устойчивы к новым типам вирусов, поскольку на момент атаки база сигнатур вирусов еще не содержат соответствующей сигнатуры.

Вторжения могут быть обнаружены по образцам поведения. Инструменты защиты компьютерных систем, использующие известные образцы поведения, называются *системами обнаружения вторжений* (*intrusion detection systems – IDS*). Основным источником информации для IDS являются системные журналы и протоколы, параметры и содержание сетевого трафика, показатели функционирования системы, такие как объем используемой памяти, число операций ввода-вывода, количество работающих процессов и т.д. В традиционных IDS используется *сигнатурный* подход, похожий на способ обнаружения вирусов. При таком подходе с помощью набора правил или сигнатур, заданных экспертом и хранящихся в базе знаний, описываются характеристики и сценарии возможных атак. Для сложных распределенных атак проверка на соответствие сценарию является нетривиальной задачей. Большинство баз знаний сигнатур и правил общедоступны, поэтому злоумышленник может использовать методы, «маскировки» атаки.

В системах обнаружения вторжений на основе *интеллектуального анализа данных* (Data Mining) в указанные недостатки частично преодолеваются. Основная идея применения этих методов основывается на предположении о том, что активность пользователей и программ в системе может быть отслежена и построена ее математическая модель. Методы интеллектуального анализа данных, применяемые в IDS, делятся на две группы: методы *обнаружения нарушений (misuse detection)*, которые строят модель атаки, а в процессе обнаружения используют методы *классификации*; и методы *обнаружения аномалий (anomaly detection)*, которые строят модель нормальной активности, а в процессе обнаружения используют методы *поиска исключений.*

Как показывает анализ литературы проблема защиты программного обеспечения в большей мере проблема социальная (психологическая, юридическая, организационная) и в меньшей степени проблема теоретического программирования. Неудивительно поэтому, что теоретические результаты мало связаны с успехом практических мероприятий. Например, невозможность определения, является ли программа злонамеренной, не препятствует развитию технологий обнаружения компьютерных вирусов и даже появлению фактически весьма прибыльной антивирусной промышленности. Несмотря на утверждения о трудности извлечения потоков управления и данных, имеются результаты, что для всех опубликованных в открытой печати методов «запутывания» программ существуют достаточно эффективные практически (не требующие полного знания семантики программы), хотя и не полностью автоматические, способы противодействия.

Основная идея «запутывания кода программы» (code obfuscation) состоит в том, чтобы использовать инструментальные средства для преобразования программы с сохранением языка представления (например, в форме исходного текста) к функционально эквивалентному виду с целью создать максимальные препятствия для обратного проектирования. Этот метод предполагает построение специального языкового процессора, который используется как препроцессор стандартных компиляторов. «Запутывание кода» в широком смысле имеет большое значение при распространении программ на языке Java (и на C#). Для большинства языков преобразование обратное компиляции является очень трудным, почти единственное исключение представляет собой языки, использующие смешанную схему компиляции на основе промежуточного байт-кода. Для языков Java и C#, использующих байт-код, это преобразование легко осуществимо. На трудности преобразования обратного компиляции, держится защита авторских прав при распространении программ в виде готовых к исполнению модулей. Для Java защита авторских прав при распространении программ в виде готовых к исполнению модулей не работает из-за простоты восстановления исходного текста.

Формальной моделью «запутывания кода программы» является языковой процессор, который преобразует программу *P* в новую программу ***О***(*P*), функционально эквивалентную *P.* Известно, что статический анализ (раздел 6.3.3) подходящим образом преобразованных программ NP-труден.

Под универсальным понимается такой запутыватель, который для любой программы строит запутанную программу, такую, что определение любого свойства программы, легко определимого по исходной программе, неэффективно по запутанной программе. Проблематика формальной операции «запутывания кода программы» сейчас представляет только исторический интерес. Несмотря на наличие признаков хорошей формализации, исследование этого «метода» показало, что он неосуществим. Польза работ в этом направлении состоит хотя бы в том, что их побочным результатом было открытие криптосистем с публичным ключом. Первооткрыватели криптосистемы с публичным ключом Diffie и Hellman предполагали, что криптосистема с публичным ключом могла бы существовать на основании того, что может быть возможно «запутать» схему шифрования секретного ключа.

Защита программного обеспечения может рассматриваться не только как проблема теоретического программирования, но также как проблема *психологии*. Понимание программ – это процесс, результатом которого является некоторое *знание* субъекта, изучающего программу. Методы запутывания применимы (и практически иногда дают хороший результат) как средство воздействия на злоумышленника путем использования свойств (точнее, слабостей) человеческой психики.

Обратное проектирование – естественный первый шаг нападающего для модификации программы. Цель обратного проектирования состоит в том, чтобы понять программу и составить план изменений в тексте программы для нейтрализации защиты. Обнаружение таких нарушений целостности исходного программного обеспечения – это цель защиты программ путем предотвращения изменений. Первые результаты в разработке методов предотвращения изменений кода программы были получены в конце прошлого века. Программа считается стойкой к изменению, если она должным образом может надежно функционировать во враждебной среде. Архитектура таких программ основана на использовании ядра проверки целостности (Integrity Verification Kernel – IVK), которое анализирует целостность критических частей программы. Предполагается, что само ядро IVK защищено от изменений.

В альтернативной архитектуре для самоконтроля кода с целью предотвращения изменений в тело программы внедряется большое количество кодовых фрагментов, которые во время выполнения вызывают специальные функции – тестеры. Каждый тестер проверяет на целостность свою маленькую часть кода (используя линейную хеш-функцию и ее ожидаемое значение). В случае нарушения целостности тестер сигнализирует об изменениях. Для отключения защиты злоумышленнику необходимо найти и отключить каждый тестер. Чем больше тестеров включено в программу, тем выше трудоемкость работы по отключению защиты.

*Полиморфность кода* – это идея, которая является частью программного фольклора, и она только недавно получила существенное внимание в сообществе разработчиков защиты. В общем, идея проста: в природе генетическое разнообразие обеспечивает защиту любого биологического вида, поражаемого вирусом или болезнью. Та же самая идея применяется в разработке программного обеспечения: известно, что представления функции существует неограниченное количество форм реализующей программы. Имея для продажи ряд функционально эквивалентных, но различных форм программы, можно надеяться на усложнение работы злоумышленника при нахождении секретного кода. Полиморфность кода улучшает сопротивление атакам, эксплуатирующим уязвимости программных продуктов, и способствует успеху при нападении с использованием вредоносной программы.

Значение полиморфности кода как механизма защиты против компьютерных вирусов и других программных атак было известно еще в 90-х годах. Чтобы обеспечить полиморфность кода может использоваться слегка измененная архитектура автоматизированных инструментальных средств преобразования, разработанных для «запутывания» кода. В этом случае трудность обратного проектирования представляет собой не единственный способ защиты, более важно то, что программа (exploit), автоматизирующая снятие защиты и разработанная для одного варианта программы не обязательно будет работать на других вариантах. Таким образом, фактор полиморфности кода в задаче защиты имеет значение не как теоретическая проблема, а как практический прием, ориентированный на противодействие определенной технологии реализации атаки. Идея о зависимости характеристики надежности и живучести сетей от полиморфности кода приобрела популярность в связи с инцидентами глобальных сетевых атак на серверы Интернет.

Защита, связанная с использованием трудно решаемых задач, основана на следующем комплексе предположений: 1) атака требует изменения в коде программы; 2) изменения в коде программы невозможны без знания семантики программы; 3) это знание может быть приобретено путем статического анализа программы. Статический анализ программы обычно включает следующие стандартные шаги: лексический, синтаксический, семантический анализ; анализ потока управления (построение графа потока управления и деревьев доминирования и постдоминирования операторов) и консервативный глобальный анализ потоков данных (для разрешения определений переменных и операндов выражений с учётом возможного именования одного объекта несколькими именами).

Реальные программы имеют тенденцию иметь легко извлекаемый поток управления, поскольку это полезно для понимания программы и обусловлено применением конструкций управления в языках высокого уровня. В этом случае извлечение графа потока программы – это операция линейной сложности *O*(*n*), где *n* – число базовых блоков в программе. Однако в общем случае задача извлечения потока управления NP–трудна.

Последний факт используется для защиты программы путем преобразования их графа потока управления так, чтобы сделать его трудноизвлекаемым, например, путем введения *непрозрачных* предикатов (opaque predicates). Переменная *v* является *непрозрачной*, если существует свойство π относительно этой переменной, которое априори известно перед преобразованием, но трудноустанавливаемо после преобразования. Аналогично, предикат *P* называется *непрозрачным*, если его значение известно перед преобразованием программы, но трудноустанавливаемо после преобразования. Метод построения непрозрачных предикатов и переменных основан на «встраивании» в программу вычислительно сложных задач, например, задачи *3-выполнимости КНФ*.

Фрагмент программы, который никогда не выполняется, называется *недостижимым* кодом (unreachable code). Общая задача обнаружения недостижимого кода алгоритмически неразрешима. Поскольку недостижимый код никогда не выполняется, преобразование **внесения недостижимого кода** влияет только на размер программы, но не на скорость её выполнения. Для выявления недостижимого кода бесполезно использовать отладчик, поэтому должны применяться методы, основанные на статистическом анализе программы. В отличие от недостижимого кода, *мёртвый* код в программе выполняется, но его выполнение никак не влияет на результат работы программы. Задача выявления мёртвого кода в программе, состоящей из одного базового блока, NP-трудна, и, более того, для любого *p* > 1 не существует *p*-приближённый полиномиальный алгоритм выявления мёртвого кода.

Большую проблему представляет то, что для методов защиты, основанных на сложности решения трудно решаемых задач, часто неясно, нужно ли нападающим в действительности решать эти трудные задачи, чтобы достигнуть своей цели. В случае «Легальная программа во враждебной среде исполнения» невозможно исключить другие условия атаки (например, атаку можно выполнить в условиях отсутствия знаний смысла всей программы). Следовательно, в этой ситуации нужно предполагать, что злоумышленник имеет возможность не решать эти трудные задачи.

Внутренняя ограниченность защиты исключительно на основе программных решений поставила задачу разработки дешевой и универсальной аппаратно защищенной платформы. В январе 1999 усилиями Compaq, HEWLETT-PACKARD, IBM, Intel и Microsoft был образован косорциум «Защищенная Вычислительная Платформа» (Trusted Computing Platform Alliance – TCPA), потом переименованный в группу «Защищенные Вычисления» – TCG (Trusted Computing Group). Основная функциональность защиты защищенной подсистемы TCG состоит в том, чтобы «создать базу защиты для программных процессов, основанную на аппаратных средствах небольшого объема». Консорциум предлагает стандарт «более безопасного» персонального компьютера. Определение «защиты» этого консорциума спорно; машины, построенные согласно их спецификации, будут заслуживать большего доверия с точки зрения продавцов программных средств и поставщиков информации, но заслуживают меньше доверия с точки зрения их владельцев. Спецификация TCG перемещает действительное управление компьютером от пользователя к тому, кто написал программное обеспечение, которое выполняется на этом компьютере.

Идея создания такой платформы основана на надежном процессе первоначальной загрузки компьютера. В основе создания благонадежной среды исполнения лежит процесс инициализации компьютерной системы так, что в ходе начальной загрузки последовательно подтверждается целостность каждого уровня операционной системы. Это обеспечивает цепочку гарантий целостности, основанных на начальном доверии к аппаратному уровню.

Надежный процесс начальной загрузки в безопасном компьютере стандарта TCG обеспечивают smart-карта или аппаратный ключ, помещенный на системную плату, и «секретная память» в центральном процессоре. В текущей версии TCG аппаратный ключ является пассивной компонентой контроля, который сохраняет состояние компьютера на запуске. Состояние хранится сжато в виде «контрольной суммы» (hash). Эта контрольная сумма вычисляется на основе особенностей аппаратных средств (звуковая плата, видео плата и т.д.) и программного обеспечения (OC, драйверы, и т.д.). Если машина оказывается в предусмотренном состоянии, аппаратный ключ предоставляет операционной системе криптографические ключи, чтобы расшифровать программы приложений и данные. Если процесс загрузки заканчивается неправильным состоянием (не та контрольная сумма), аппаратный ключ блокирует доступ к криптографическим ключам. В этом случае компьютер может выполнять незащищенные приложения и обращаться к незащищенным данным, но защищенная информация будет недоступна.

В последнее время все большую популярность при обеспечении защиты программного обеспечения от несанкционированного использования приобретают программно-аппаратные средства защиты, известные как «аппаратные (электронные) ключи».

Под термином «электронный ключ» понимается аппаратная реализация системы защиты и соответствующего программного обеспечения. Сам ключ представляет собой электронное устройство, которое может подключаться к LPT, COM, PCMCIA, USB портам компьютера (рисунок 8.2). В его программное обеспечение входит модуль, который встраивается в защищаемую программу. Таким образом, данное программное обеспечение «привязывается» к ключу, а не к конкретному компьютеру.

Рисунок 8.2 – Электронный ключ, подсоединяемый к порту принтера

В последнее время в качестве электронного ключа широко используются смарт-карты. Носителем информации в них является микросхема. Наиболее предпочтительными для защиты программного обеспечения являются смарт-карты, в которых реализованы криптографические алгоритмы (поддержка алгоритмов DES, RSA и других) (рисунок 8.3). Смарт-карты представляют собой пластиковые карточки, внутри которых встроена интегральная микросхема (smart – умный), благодаря которой смарт-карты используют в различных технических приложениях. Эти карты имеет стандартные размеры 85,6мм × 53,98мм × 0,76мм - такие же, как и повсеместно используемые банковские карты с магнитной полосой. Самое широкие применение смарт-карты нашли в качестве электронного платежного средства и средства идентификацию владельца в системах контроля доступа.

Рисунок 8.3 – Смарт-карта для порта USB

На сегодняшний день самый распространенный класс ключей изготавливается на основе специализированной СБИС. Эта СБИС аппаратно реализует один или несколько алгоритмов криптографической обработки данных. Все ключи одной модели работают по одинаковому алгоритму или алгоритмам (т.е. в них содержаться функции одинакового вида). Такая особенность может неблагоприятно сказываться на степени стойкости системы защиты. СБИС, использующиеся в электронных ключах, имеют небольшую схемотехническую сложность (порядка тысяч логических модулей) и могут обрабатывать данные длиной в несколько байт.

В общих чертах работу системы защиты посредством электронного ключа можно представить следующим образом.

1. В процессе работы защищенная программа передает электронному ключу информацию, так называемый «вопрос».
2. Электронный ключ ее обрабатывает и возвращает обратно – «отвечает».
3. Программа на основе возвращенных данных идентифицирует ключ. Если он имеет верные параметры, программа продолжает работать. Если же параметры ключа не подходят, либо он не подсоединен, то программа прекращает свою работу или переходит в демонстрационный режим.

**Изготовление аппаратной копии ключа –** этот метод заключается в считывании специальными программными и аппаратными средствами содержимого памяти ключа. Затем данные переносятся в микросхему другого ключа. Этот метод может применяться, если память ключа не защищена от считывания информации. В ключах на основе специализированной СБИС память может быть реализована как состояния автомата. Считывание данных из такого ключа эквивалентно эксперименту по распознаванию автомата с очень большим числом состояний. Эта задача имеет большую сложность, экспоненциально зависящую от числа состояний автомата. Реально изготовление копии состоит в перепроектировании СБИС (для этого нужно иметь формальное описание поведения аппаратуры ключа). К тому же, создание аппаратной копии ключа не решает проблему тиражирования программы, ведь она все равно остается «привязанной», но только к другому ключу.

Эмуляторы могут воспроизводить работу ключей определенной модели, или ключей, поставляемых с какой-то программой, или одного конкретного ключа.

По организации их можно разделить на эмуляторы структуры и эмуляторы функциональности. Первые моделируют работу аппаратуры ключа, вторые работают на основе таблицы вопросов и ответов конкретного ключа. Чтобы построить эмулятор первого типа нужно знать описание СБИС на каком-либо языке описания аппаратуры (VHDL или Verilog). Для создания второго эмулятора нужно составить таблицу, в которой указаны все возможные верные вопросы к ключу и сопоставленные им ответы.

Конструктивно наиболее защищены от эмуляции ключи на основе специализированной СБИС. Для этих ключей разработка программного эмулятора требует решения тех же задач, что и в атаке методом изготовления аппаратной копии ключа.

Целью подмены драйвера является перехват потока данных между портом и аппаратурой ключа. Современные ключи обладают целым набором средств, предотвращающих перехват потока данных. Прежде всего, это различные варианты усложнения протокола обмена ключа и драйвера, а также кодирование передаваемых данных. Используются следующие основные виды защищенных протоколов обмена или их сочетания:

1. плавающий протокол – вместе с реальными данными передается «мусор», причем со временем порядок чередования и характер, как реальных, так и ненужных данных, изменяется хаотическим образом
2. кодированный протокол – все передаваемые данные кодируются. Расшифровка принимаемых данных осуществляется аппаратурой ключа. Передаваемые ключом данные тоже зашифровываются. Расшифрование выполняет драйвер.

Во время выполнения ключ получает зашифрованные данные от драйвера и расшифровывает их таким способом, который нельзя подделать злоумышленнику. Расшифрованные данные, возвращенные драйвером в программу, используются так, чтобы выбрать один из режимов исполнения программы: программа загружается и выполняется, программа выполняет только некоторые модули или компоненты, или программа не выполняется вообще.

При удачной атаке этого типа драйвер подменяется программой, эмулирующей работу комплекса: драйвер – ключ. Стойкость защиты по этому виду атаки зависит от реализации драйвера. Для взлома защиты требуется нейтрализовать эффект шифрования. При использовании криптостойких алгоритмов шифрования в драйвере и ключе раскрытие защиты возможно только путем компрометации ключа шифрования. Как показывает практика взломов, драйверы современных ключей реализованы адекватно степени угрозы. Известные взломы драйвера ключей осуществлялись только после утечки информации от производителя ключа.

**З**ащищенная программа содержит ряд программных блокировок, которые и обеспечивают предотвращение несанкционированного доступа. Каждая программная блокировка состоит из трех компонентов: вызов функции обращения к драйверу (функции API ключа), оценка возвращаемого значения и выполнение некоторого действия по результатам оценки. Удаление программной блокировки снимает защиту программы. Программные блокировки могут быть обнаружены в коде программы по вызовам функции API ключа.

Обнаружение вызовов функций API ключа может выполняться статически: путем дизассемблирования, или динамически: при выполнении программы с помощью отладчика. Современные дизассемблеры позволяют указать набор сигнатур для определенных вызовов функций, что позволяет легко обнаруживать места программных блокировок в распечатке результатов работы дизассемблера. Сигнатуры вызовов функций API для наиболее используемых ключей можно найти в интернете. Современные отладчики позволяют поставить точку прерывания на вызов любой нелокальной (расположенной в другой программной компоненте) функции. Это возможность позволяет легко обнаруживать обращения к вызовам функции API ключа, происходящие в ходе прогона программы в сеансе отладки.

Итак, предположим, что сам ключ и реализация протокола обмена между ключом и программой надежны. Задача состоит в том, чтобы построить схему защиты в предположении, что злоумышленник обладает не меньшими знаниями в программировании, чем самый лучший современный программист.

Противодействие методам статического обнаружения вызовов функций API неэффективно при использовании следующих приемов.

1. Использование «безумного кода»: при создании функций API их команды перемешиваются с «мусором» – ненужными командами, т.о. код сильно «зашумляется», что затрудняет исследование логики работы функций. Обнаружение выполняется по сигнатурам вызовов и не зависит от сложности или логичности структуры программы.
2. Использование множества точек входа в API. Трудоемкость обнаружения повышается за счет необходимости использования большего количества сигнатур. Практически это не влияет на успех обнаружения вызовов. Вызовы API могут быть легко обнаружены подключение довольно простого аппаратного дополнения.

Эффективным противодействием методу статического анализа программы является шифрование всего исполняемого кода или участков, содержащих блокировки. Однако в этом случае вызовы функции API ключа, происходящие в ходе прогона программы, могут быть обнаружены в ходе сеансе отладки при использовании современных отладчиков. Причем не существует реальных (не мифических) приемов помешать отладчику обнаружить эти вызовы.

Тем не менее, шифрование исполняемого кода обеспечивает надежную защиту от атаки злоумышленника имеющего код программы, но не имеющего ключа. Динамически в отсутствие ключа можно обнаруживать блокировки только по одной за сеанс отладки, что существенно повышает трудоемкость взлома, но не снимает принципиальной возможности нейтрализации защиты. Шифрование кода в этом случае не является принципиальным препятствием: вызовы обнаруживаются в тот момент, когда выполняемый участок кода расшифрован. Шифрование может технически очень усложнить удаление обнаруженной блокировки, но принципиальная возможность обхода защиты остается.

Таким образом, приходится признать, что злоумышленник всегда имеет возможность обнаружить вызов функции API ключа. Однако он не всегда имеет возможность удалить обнаруженную программную блокировку. Эта возможность существенно зависит от реализации двух остальных компонентов блокировки: оценки возвращаемого значения и выполнения некоторого действия по результатам оценки.

Если эти компоненты реализованы как операции проверки значения и разветвления: знаменитое “if (test) good boys else bad boys”, то удаление блокировки не составляет труда. Рекомендации производителей использовать в блокировке конструкцию *y = f(test)*, где *f* реализует некоторые вычисления, корректные только при правильном значении *test*, затрудняют распознавание блокировки. По крайней мере, это делает невозможным создание полностью автоматических средств устранения защиты. Следует заметить, что если функция *f* оформлена как программная единица, которой *test* передается как параметр, то блокировка легко снимается заменой параметра *test* на константу. Функция должна быть выбрана так, чтобы по ее виду нельзя было предсказать возможные значения параметра. Примером таких функций являются функции расшифрования криптостойких алгоритмов шифрования, параметром которых является ключ шифрования.

Эффективный прием усложнения логики защиты – откладывание реакции программы на коды возврата функций API. В этом случае программа принимает решение о дальнейшей работе спустя какое-то время после получения кодов возврата. Для нахождения части блокировки, связанной с анализом возвращенного значения, злоумышленник может воспользоваться средствами построения таблиц перекрестных ссылок или применить отладчик в комплексе со специальной отладочной платой (позволяющей поставить точку прерывания отладчика на обращение к заданному участку памяти).

Применение электронного ключа позволяет построить надежную схему защиты от злоумышленника, не имеющего ключа. Часто поставщик программного обеспечения должен предусмотреть защиту и от легальных покупателей, которые имеют ключи и могут использовать эту возможность для взлома программы. В этой ситуации рекомендации предыдущего параграфа в известной степени обесцениваются. Например, значение параметра функции *f* может быть определено при прогоне программы с помощью отладчика. После чего программная блокировка снимается заменой этого параметра на соответствующую константу и удалением обращения к ключу. Резко падает также ценность метода шифрования исполняемого кода: пароль расшифрования, хранящийся в ключе, может быть перехвачен при прогоне программы, и взлом выполняется заменой обращения к ключу соответствующей константой.

Надежным способом защиты программы и данных от несанкционированного использования является перенос части действий из персонального компьютера в ключ. Очевидно, этот способ возможен, если ключ способен реализовать нетривиальные операции. Если вместо сложения двух чисел в компьютере вызвать соответствующую функцию ключа и использовать ее результат в дальнейших вычислениях, то злоумышленник по виду аргумента и результата может легко угадать операцию. Примером нетривиальной операции может быть операция вычисления контрольной суммы блока данных. Эта операция должна быть дополнена операцией проверки, реализованной так, что контрольные суммы не выходит из ключа при проверке. Функции зашифрования и расшифрования, реализованные аппаратно в ключе, полезны, если пароль шифрования тоже никогда не покидает аппаратуру электронного ключа.

Ключи на основе специализированной СБИС первого поколения, как правило, не способны реализовать такие операции по причине недостаточности ресурсов. Для защиты от злоумышленника, обладающего ключом, необходимо использовать сложные (и дорогие) ключи последних поколений или использовать в качестве ключа смарт-карты, в которых реализованы криптографические алгоритмы. Хранение наиболее ценной информации внутри ключа, организованное таким образом, что эта информация никогда не покидает ключ, гарантирует, что владение ключом не даст дополнительных возможностей по взлому защиты.

Следует отметить, что для анализа смарт-карты могут использоваться побочные каналы, в которых в роли (невольных) передатчиков выступают штатные компоненты информационных систем, а в роли приемников – внешние наблюдатели, применяющие соответствующее оборудование. Чаще всего с помощью побочных каналов измеряется время видимых операций (временные атаки на RSA стали общим местом), их энергопотребление и/или побочные электромагнитные излучения и наводки.

Весьма эффективны атаки, основанные на дифференциальном анализе энергопотребления. В 1998 году Брюс Шнейер писал, что в галактике не хватит кремния, а у Солнца – времени жизни для реализации атаки методом грубой силы на секретный ключ (112 бит) алгоритма 3DES. Минимальная длина ключа в алгоритме AES – 128 бит, но успешная атака методом дифференциального анализа энергопотребления на незащищенную микросхему, реализующую AES, может быть проведена менее чем за три минуты – от начала измерений до завершения анализа.