Развитие taint-анализа для решения задачи поиска программных закладок

А.Ю.Тихонов, А.И. Аветисян
{fireboo@ispras.ru}, {arut@ispras.ru}

Аннотация. Задачу анализа программного обеспечения (ПО) будем рассматривать как задачу получения некоторых свойств исследуемого ПО. Это могут быть и описания реализуемых алгоритмов, и информация о структурах файлов, данных в памяти или пакетов сетевых протоколов, и информация об имеющихся ошибках. В данной статье будет рассмотрен подход к анализу программного обеспечения, представленного в виде исполняемого кода при отсутствии исходных текстов, с целью выявления некоторых видов программных закладок. Согласно ГОСТ Р 51275-2006 программная закладка – это преднамеренно внесенный в ПО функциональный объект, который при определенных условиях инициирует реализацию недекларированных возможностей ПО. Программная закладка может быть реализована в виде вредоносной программы или программного кода.
Предлагаемый подход к анализу описывается в пятом разделе статьи. Первые четыре раздела кратко описывают принципы работы системы анализа программного обеспечения по трассе исполнения программы TREX, в рамках которой ведется практическая реализация данного подхода.

Ключевые слова: безопасность ПО, недекларированные возможности, вредоносный код, анализ уязвимостей

1. Введение

Система TREX предназначена для анализа ПО, представленного в виде исполняемого кода, при отсутствии исходных текстов [1], [2], [3]. В качестве анализируемого программного обеспечения выступает не какая-то отдельно взятая программа или исполняемый модуль, а вся программная система, код которой выполняется процессором. Предполагается, что анализируемая система может противодействовать анализу путем использования различных методов защиты от статического и динамического анализа — такое предположение верно как по отношению к вредоносному ПО, так и по отношению к обычным программам, разработчики которых хотели защититься, например, от несанкционированной модификации кода. В основе алгоритмов анализа трассы лежат принципы анализа потока управления и потока данных.

Рассматриваемый в работе подход основан на известных методах анализа потока управления и данных. Особое внимание уделено уточнению понятия закладки. Существующие определения даны в нормативной базе, которая создавалась в первую очередь под решение задач сертификации на отсутствие недекларированных возможностей (НДВ). При этом разработчик обязан предоставить исходные тексты, а в некоторых случаях — и проектную документацию, которую можно использовать как основу для построения формальной модели исследуемой системы. В реальной жизни исследуемая программа функционирует в недоверенном программном окружении, запрашивая сервисы со стороны такого окружения и подвергаясь его воздействию. В этом случае анализируемой системой является вся программная система в целом. Перед началом анализа в распоряжении аналитика нет ни исходных текстов, ни проектной документации. Нормативные документы анализ в таких условиях не предусматривают. Кроме того, зарубежные исследователи отмечают принципиальную невозможность проведения анализа программ только на уровне исходных текстов, без анализа исполняемого кода — так называемая проблема WYSINWYX (What You See Is Not What You eXecute) [4]. Важным вопросом является построение формального описания модели исследуемой системы, на основе которого можно было бы различать допустимые и недопустимые пути исполнения или состояния программы. Такое описание можно назвать политикой безопасности, вот только для возможности использования при анализе исполняемого кода объекты и субъекты такой политики должны в итоге сводиться к уровню машинных команд, ячеек памяти и состояний программы. Существующие нормативные документы не описывают как это сделать.

В работе сделана попытка заполнить пробел между существующей нормативной базой и практикой при решении задачи поиска вредоносного кода, реализующего некоторые виды программных закладок. В основе подхода лежат методы слайсинга и динамического taint-анализа. В текущем виде описанный подход является вариантом динамического анализа кода, при котором отслеживаются зависимости между инструкциями по управлению и по данным, фактически возникающие при выполнении программы на конкретном наборе тестовых данных. Вследствие этого основным недостатком является невозможность анализа всех возможных путей исполнения и состояний программы. Преодоление этих недостатков видится в использовании методов символьного исполнения программы (symbolic execution), что является темой дальнейших исследований. Стоит отметить, что комбинация динамического анализа по трассе и символьного исполнения лежит в основе ряда зарубежных исследований, направленных на разработку практических систем анализа программного обеспечения [5].

2. Требования к содержимому трассы

Для обеспечения корректности проведения анализа потока управления и потока данных в трассе должна содержаться информация, полностью определяющая процесс преобразования, выполняемый над ячейками памяти и регистрами процессора:

- информация о непрерывная последовательность выполненных процессором инструкций;
- информация о смене режимов работы процессора и значений системных регистров, влияющих на декодирование инструкций, а также информация, требуемая для корректного дизассемблирования инструкции (см. раздел 2).
- информация о возможность восстановления содержимого любого регистра общего назначения на любом шаге;
- информация о факте прямой модификации содержимого ячейки памяти (запись в режиме DMA), т.е. модификации памяти, вызванной не выполнением процессорной инструкции, а работой других устройств;
- информация о факте асинхронного вызова обработчика прерывания, т.е. вызова обработчика прерывания не по причине выполнения специальной процессорной инструкции, а по причине возникновения аппаратного события;
- информация о содержимом ячеек памяти, определяющих поведение конкретных инструкций. Особенно важно знать содержимое ячейки, если она влияет на зависимости по данным, формируемые инструкцией. В случае системы команд x86 примером такой инструкции является CMPXCHG/CMPXCHG8B/CMPXCHG16B;
- информация о некоторых случаях требуется возможность восстановления содержимого ячеек памяти на момент выполнения конкретного шага трассы (см. раздел 4 поиск закладок);
- информация о случаях, когда процессор различает виртуальную и физическую адресацию памяти, требуется механизм отождествления физически одинаковых ячеек памяти, имеющих различающиеся виртуальные адреса, по которым эти ячейки адресуются процессором;
- информация о случаях, когда процессор и/или операционная система поддерживает многозадачность, требуется механизм идентификации задач для того, чтобы уметь определять, какой из множества выполняемых задач принадлежит текущая инструкция.

Не всю эту информацию необходимо получать непосредственно на этапе получения трассы. Часть информации может быть восстановлена на этапе обработки уже полученной трассы.

3. Декодирование и дизассемблирование инструкций процессора

Инструкции процессора имеют числовое представление и хранятся в памяти компьютера. В случае компьютеров, построенных на базе архитектуры фон-Неймана инструкции хранятся в той же памяти, что и данные, и по сути ничем от данных не отличаются. Так же как и обычные данные, код инструкций может быть прочитан и модифицирован. Это приводит к известной проблеме неразличимости инструкций и данных. Кроме того, в некоторых процессорных архитектурах размер инструкций может различаться, в результате чего даже внутри линейного блока кода невозможно декодировать одну инструкцию, не декодировав все ей предшествующие. В зависимости от режима работы процессора интерпретация одного и того же числового значения может различаться. На интерпретацию инструкции может влиять ее положение в памяти — например в случае, когда код инструкции содержит относительный адрес, т.е. когда адрес ячейки памяти, к которой обращается инструкция, должен быть вычислен относительно положения инструкции в памяти. Еще одна проблема — различение констант и адресов. Одна и та же, корректно декодированная инструкция, с точки зрения представления на языке ассемблера может быть представлена по-разному, например, как инструкция помещения в регистр числовой константы, либо как инструкция помещения в регистр адреса ячейки памяти.

Подводя итог сказанному, будем различать термины декодирование и дизассемблирование инструкции.

Декодирование — процесс интерпретации кода инструкции, при котором мы получаем информацию о функциональности инструкции. Декодирование — неоднозначное преобразование, если нет информации о состоянии процессора. При динамическом анализе декодирование — однозначно, при статическом — неоднозначно.

Дизассемблирование — это получение корректного ассемблерного представления инструкции, при котором выполняется декодирование и получение информации о смысле инструкции. Рассмотрим случай с инструкцией, помещающей константу в регистр. Константа возможно является адресом. Вне зависимости от этого ее значение будет помещено в регистр. В случае корректного декодирования такой инструкции, но некорректного дизассемблирования будут корректно учтены функциональные зависимости по данным, и некорректно — адресные зависимости. Корректность распознавания адресов можно обеспечить только в рамках конкретной трассы, причем в общем случае с ошибками второго рода. Например, с помощью слайсинга (см. раздел 3) можно отследить использование константы для поиска ситуации, когда она явным образом используется в качестве адреса. Но если константа пройдет через серию преобразований, результатом которых будет исходное значение — анализ кода не поможет, потребуется полноценный анализ алгоритмов.

Единственный способ решения — явный контроль обращений по адресам и проверка — совпадает ли значение проверяемой константы с адресом. При таком подходе все константные адреса, к которым шло обращение в трассе, будут обнаружены (нет ошибок первого рода), но могут быть ложные отождествления константы с адресом (возможны ошибки второго рода).

4. Принципы анализа трассы. Слайсинг и taint-анализ

В основе существующих подходов к решению задач анализа потока управления и потока данных лежит предположение о том, что для исследуемой программы известны корректный граф потока управления — **CFG** (Control Flow Graph) и граф зависимостей программы — **PDG** (Program Dependence Graph). Неформально — CFG описывает поток управления, т.е. возможный порядок исполнения инструкций, его вершинами являются операторы программы, а ориентированные связи между вершинами показывают допустимый порядок исполнения операторов. PDG описывает отношения зависимостей по данным и управлению между операторами на множестве вершин GFG. Связь первого типа идет от одного оператора к другому, если результаты вычислений одного оператора используются в качестве входных данных для вычислений другого. Связь второго типа идет в том случае, если выполнение одного оператора определяет будет ли выполняться другой оператор. На практике, особенно при работе с исполняемым кодом, зачастую неизвестно ни полное множество инструкций, ни связи между ними по управлению и по данным.

Основой подхода, описываемого в данной работе, является получение знаний об инструкциях исследуемой программы, порядке их выполнения и зависимостях между ними по данным напрямую, путем наблюдения за работой программы в интервале времени, в который укладывается работа интересующего аналитика алгоритма. Результатом такого наблюдения является **трасса исполнения программы**, представляющая собой описание последовательности шагов выполнения программы, для каждого из которых сохранен код выполненной инструкции и состояние всех регистров процессора. Основная идея – свести трассу к представлению в виде CFG и PDG и адаптировать к их обработке известные алгоритмы. Преимуществом данного подхода является то, что он позволяет получать гарантированные и точные результаты. Недостатком — то, что эти результаты относятся только к той части алгоритма, которая была выполнена. Оставшиеся незадействованными ветки алгоритма анализироваться не будут. Тем не менее, предлагаемый подход предпочтителен по сравнению с существующими, поскольку для выполнившейся части алгоритма анализ полон и не содержит неточностей, которые возникают у других подходов. Для оставшейся непроанализированной части алгоритма можно адаптировать существующие подходы к статическому и динамическому анализу, суммарно получив более полные результаты анализа.

В предлагаемом подходе различаются четыре типа **зависимостей**:

- функциональная зависимость — когда результат вычисления на одном шаге напрямую зависит от результата вычисления на другом;
- адресная зависимость — когда выбор ячейки, от которой функционально зависит результат вычисления на одном шаге, зависит от результата вычисления на другом шаге;
- зависимость по управлению;
- зависимость по коду инструкции — когда код инструкции, выполняющейся на одном шаге, зависит от результата вычисления на другом. Рассмотрение данной зависимости важно при анализе динамически изменяемого кода программы.

Суть слайсинга по трассе — отбор из трассы шагов в соответствии с зависимостями, позволяющими отнести эти шаги к интересующему аналитика алгоритму преобразования входных ячеек в выходные. Шаг трассы отбирается в слайс, если выполненная на этом шаге инструкция **зависит** хотя бы от одной указанной аналитиком входной ячейки, или от результата работы инструкции **зависит** хотя бы одна выходная ячейка.

Taint-анализ при анализе трассы по сути мало отличается от слайсинга, только отбираются не шаги (инструкции), а данные (ячейки памяти и регистры). Перед началом анализа аналитик помечает некоторые данные, исходя из своих соображений, после чего осуществляется продвижение пометок (taint propagation) вверх или вниз по трассе в соответствии с зависимостями по помеченных данным: если входом инструкции на некотором шаге являются помеченные данные, то выходы инструкции также помечаются для анализа на следующем шаге. Если выходом инструкции на некотором шаге являются помеченные данные, то входы инструкции помечаются для анализа на предыдущем шаге.

Отличия между слайсингом и taint-анализом проявятся в предлагаемом для поиска закладок подходе, когда с этапом продвижения меток может быть связана дополнительная обработка, рассмотренная в следующем разделе.

5. Поиск закладок

В настоящее время наблюдается "эпидемия" вредоносных программ, ориентированных на кражу конфиденциальных данных и тем самым наносящих вред конкретному пользователю. Под них могут маскироваться и шпионские программы, разработанных спецслужбами различных государств, которые причиняют ущерб уже на государственном уровне. Для выявления вредоносного кода и оценки возможного ущерба иногда полезно выяснить цель атаки — конкретные данные, на добычу которых ориентирована такая программа.

Вначале требуется формально определить, что является закладкой. Без такого определения не ясно, что требуется найти, а потому невозможно предложить методы поиска. В Российских стандартах имеется два сходных и довольно общих определения:

1. Программная закладка — Преднамеренно внесенный в программное обеспечение функциональный объект, который при определенных условиях инициирует реализацию недекларированных возможностей программного обеспечения. Программная закладка может быть реализована в виде вредоносной программы или программного кода — ГОСТ Р 51275-2006 [6].

2. Программная закладка — скрытно внесенный в программное обеспечение функциональный объект, который при определенных условиях способен обеспечить несанкционированное программное воздействие. Программная закладка может быть реализована в виде вредоносной программы или программного кода [7].

Проанализируем эти определения с точки зрения возможности их использования для формирования принципов обнаружения закладок.

Первое определение:

- преднамеренность — методами анализа исполняемого кода доказать проблематично;
- функциональный объект это присуще только модели программной системы, при анализе бинарного кода такого понятия нет;
- недекларированные возможности ПО — в соответствии с ГОСТ Р 51275-2006 [6] — функциональные возможности программного обеспечения, не описанные в документации. При отсутствии модели программной системы, описанной ее разработчиком — понятие бессмысленное.

Второе определение:

- скрытное внесение функциональности — так же как и для преднамеренности трудно доказуемо. Скрытность может являться следствием применения механизмов защиты легального кода от анализа;
- несанкционированное программное воздействие — наиболее близкое определение дано в п.2.6.6. ГОСТ Р 50922-2006 [8]: "несанкционированное воздействие на информацию: воздействие на защищаемую информацию с нарушением установленных прав и (или) правил доступа, приводящее к утечке, искажению, подделке, уничтожению, блокированию доступа к информации, а также к утрате, уничтожению или сбою функционирования носителя информации". Это определение подразумевает наличие модели программы, описывающей правила обработки информации. При анализе бинарного кода такого понятия нет, требуется построение модели программы, отражающей политику безопасности, что само по себе является сложной задачей.

Вывод — данные определения давались под конкретный подход к анализу программного обеспечения по исходным текстам при наличии формального описания модели. Такие данные для анализа бинарного кода использованы быть не могут.

Сформулируем более приемлемое определение на основе стандартизованного понятия безопасности информации.

В настоящее время под безопасностью информации принято понимать состояние защищенности информации, при котором обеспечиваются её конфиденциальность, доступность и целостность [8]. Иногда выделяют и другие признаки безопасности: невозможность отказа от авторства (non-repudiation), подотчётность (accountability), достоверность (reliability), аутентичность или подлинность (authenticity).

В соответствии с п. 2.6.5. ГОСТ Р 50922-2006 [8] вредоносная программа: программа, предназначенная для осуществления несанкционированного доступа к информации и (или) воздействия на информацию или ресурсы информационной системы.

По сути, вредоносная программа ориентирована на нарушение безопасности информации путем устранения одного или нескольких перечисленных выше признаков безопасности.

Наиболее очевидным нарушением безопасности, на которое ориентирована значительная часть вредоносных программ, является нарушение конфиденциальности, а по сути — кража информации.

Конфиденциальность — одно из свойств безопасности информации (свойство обеспечивающее безопасное состояние информации). В соответствии с ГОСТ 17799-2005 [[9]] будем понимать его как "обеспечение доступа к информации только авторизованным пользователям".

В данной работе предлагается подход к поиску вредоносного кода, предназначенного для компрометации секретных данных пользователя, т.е. к нарушению конфиденциальности. Неформально будем называть такой код **закладкой, нарушающей конфиденциальность**, далее по тексту — **закладкой**. Рассматриваемый подход может быть распространен и на поиск других типов закладок, но это тема отдельной работы.

Объектом анализа в случае динамического анализа исполняемого кода является последовательность инструкций, выполняемых процессором. Инструкции выполняют некоторые преобразования данных, содержащихся в процессорных регистрах или ячейках памяти. Каждая инструкция может иметь входные данные, сформированные ранее выполненными инструкциями, и выходные данные, являющиеся результатом выполнения инструкции. Получение данных извне процессора и выдача данных вовне процессора

реализуется специальными инструкциями, ячейками памяти и портами ввода/вывода. Любая информация, обрабатываемая процессором, должна быть введена в него извне, и после обработки выведена вовне. Зависимости между инструкциями по входным/выходным данным образуют поток данных. Поток данных имеет несколько начальных точек — инструкций, вводящих данные в процессор, и несколько конечных точек — инструкций, выводящих данные из процессора. Вычислительную систему можно рассматривать и более широко — не только как процессор, но и как все аппаратное обеспечение, составляющее вычислительную систему (компьютер), или даже вычислительную систему из нескольких взаимодействующих компьютеров. В этом случае требуется обеспечить непрерывность отслеживание потока данных между различными аппаратными компонентами вычислительной системы, например передачу данных из процессора на жесткий диск и обратно.

Если снабдить функционирующую в вычислительной системе информацию специальной пометкой, отражающей ее конфиденциальность, и контролировать потоки данных, обрабатывающие эту информацию, можно выявить все факты выдачи такой информации вовне вычислительной системы, после чего аналитик должен решить — что из них будет являться нарушением конфиденциальности. На этом принципе основан **метод анализа помеченных данных (taint-анализ)**.

Закладка должна нарушать конфиденциальность некоторых данных, для чего эти данные посредством выполнения процессорных инструкций должны попасть вовне вычислительной системы. Очевидно, что какая-то часть таких инструкций может принадлежать легальному коду операционной системы. Это необходимое, но недостаточное условие для того, чтобы отнести некоторый код к закладке. Например, код отображения конфиденциальных данных на экран может быть вполне легальным, если делает это по запросу пользователя. Легален или нет доступ к конфиденциальным данным, должно описываться политикой безопасности исследуемой системы. При начале анализа исполняемого кода она, как правило, неизвестна аналитику, и его задачей является восстановить ее в ходе анализа, и в соответствии с ней принимать решение о допустимости или недопустимости конкретного обращения к данным. Вопрос восстановления политики безопасности по коду программы является темой отдельных исследований.

С точки зрения стойкости закладки к обнаружению методами проверки работоспособности легального алгоритма на контрольных примерах, закладка не должна влиять на результаты использования компрометируемых секретных данных легальным алгоритмом.

Передача секретных данных вовне вычислительной системы может быть связана с их преобразованием, т.е. вычислением некоторой функции от исходных секретных данных. Чтобы нарушать конфиденциальность, данная функция должна быть обратима при условии контроля атакующей стороной переданных вовне вычислительной системы данных.

Если секретные данные попали вовне вычислительной системы — они скомпрометированы, при условии, что они **доступны**.

Здесь под **доступностью** понимается возможность вычисления обратной функции, т.е. возможность вычисления исходных секретных данных. Если это функция симметричного шифрования с секретным ключом, должен быть доступен секретный ключ и выход прямого алгоритма. При этом ключ шифрования либо был прошит в реализацию закладки, либо получен извне анализируемой системы. Такой ключ может быть извлечен аналитиком посредством анализа потоков данных, после чего ему становятся доступны компрометируемые закладкой секретные данные. Если используется функция асимметричного шифрования — ключ должен принадлежать атакующему, причем извлечение ключа прямого преобразования из программной реализации не позволит вычислять обратное преобразование. Иными словами, компрометируемые секретные данные будут доступны только владельцу ключа (которым автор закладки может даже и не являться). На данном принципе основаны некоторые схемы депонирования ключей.

Имеет также смысл говорить о **частичной доступности**, если компрометируется только часть секретных данных.

Проверка обратимости в общем случае является криптоаналитической задачей, и методами анализа потоков данных решена быть не может. Однако можно сформулировать частный случай, когда такую задачу можно свести к задаче анализа потоков данных — когда функция преобразования является известной криптографической функцией шифрования или комбинацией таких функций. В этом случае для доказательства доступности компрометируемых данных достаточно показать доступность ключевых данных алгоритма шифрования.

Наиболее подходящим подходом для решения задачи поиска рассматриваемого типа закладок является taint-анализ. В известных определениях taint-анализа при продвижении меток обратимость вычислений не учитывается. Для исправления этого недостатка операции, анализируемые алгоритмом продвижения меток, должны быть отнесены к классу прямого или обратного преобразования конкретного алгоритма. При прохождении помеченных данных через такую операцию ее результат также помечается, однако при этом пометка "обертывается" признаком выполненного преобразования (функция + ключ). Если в дальнейшем такой "конверт" попадает на операцию, отнесенную к обратному преобразованию той же функции, при совпадении ключей "обертка" снимается и остаются содержащиеся в ней помеченные данные. Разновидность такого подхода — для некоторых реализаций криптоалгоритмов ключи должны также снабжаться пометками "прямой/обратный ключ".

Важным вопросом является проверка совпадения ключей. Если между созданием и снятием "конверта" ключ был преобразован некоторым алгоритмом, необходимо ответить на вопрос — была ли произведена реальная

модификация ключа, либо выполненное преобразование оставило ключ неизменным — что может быть при намеренном запутывании алгоритма для усложнения анализа. В общем случае ответ на такой вопрос не может быть получен анализом потоков данных и управления. Однако при динамическом анализе можно просто сравнить содержимое ключа на момент выполнения прямого и обратного преобразования. При анализе трассы это предъявляет дополнительное требование к системе получения и обработке трассы — необходимо обеспечить **возможность восстановления содержимого памяти на момент выполнения конкретного шага трассы** (см. раздел 1 требования к содержимому трассы).

Рассмотренные выше ситуации позволяют дать определение понятия закладки в терминах, близких к анализу потоков данных и управления.

Определим закладку как обратимый алгоритм преобразования секретных данных, порождающий доступные выходные данные, помимо указанного аналитиком (санкционированного) выхода.

Консервативный анализ считает любой алгоритм преобразования секретных данных обратимым, а любой способ выдачи секретных данных вовне системы — несанкционированным доступом (политика безопасности по умолчанию). По мере выдачи системой анализа ложных, по мнению аналитика, сообщений о найденных закладках, аналитик уточняет политику безопасности, описывая конкретное обращение к данным как разрешенное. Задача восстановления политики безопасности по коду программы является темой отдельного исследования.

Поскольку окончательное решение о том, является найденный алгоритм закладкой или нет, принимает аналитик, процесс такого поиска правильнее называть **поиском мест, подозрительных на наличие закладок**.

Если закладка влияет на выход легального алгоритма — она потенциально **ослабляет этот алгоритм**.

Если одновременно с ослаблением отсутствует компрометация секретных данных — закладки как алгоритма не существует, поскольку не существует выходных данных никакого другого алгоритма, кроме ослабляемого легального алгоритма. В этом случае мы имеем дело со слабостью, присущей самому легальному алгоритму. Слабость являющаяся частью легального алгоритма, методами анализа потоков данных не выявляется, выявляется методами криптоанализа.

Выявление закладки в сформулированном выше смысле средствами только динамического анализа возможно только в том случае, когда закладка функционировала в момент проведения анализа. Более точно — поток данных, образованный кодом закладки, должен "ответвляться" от помеченных секретных данных в интервале времени снятия трассы, и иметь хотя бы один выход вовне контролируемой системы. Во временном интервале с момента ответвления кода закладки от секретных данных до момента вывода

преобразованных секретных данных вовне системы должна быть обеспечена непрерывность контроля над потоком данных. Такая непрерывность естественным образом обеспечивается в случае анализа по трассе исполнения программы. В случае статического анализа машинного кода в силу неполноты CFG и PDG (некорректность и неполнота множества инструкций, множества связей по управлению и множества связей по данным) непрерывность анализа обеспечить очень сложно.

В ситуации, когда аналитик не знает происхождение некоторых входных данных и потому не может выставить для них пометки (сбор трассы начался, когда известные аналитику секретные данные уже были неподконтрольно преобразованы), проведение анализа потока данных невозможно.

В ситуации, когда помеченные данные находятся в обработке, но не попали вовне системы (раннее прекращение контроля), возможно проведение дальнейшего анализа. Подходы к такому анализу могут быть различными, например рестарт вычислительной системы с состояния, сохраненного на момент завершения анализа (возможно на симуляторах), эмуляция или статический анализ незадействованных веток кода на предмет возможного доступа к помеченным данным.

Преимуществами подхода, основанного на taint-анализе, является относительная простота задания политики безопасности в виде пометок входных данных на некотором шаге трассы и ее последующей модификации. Необходимо задать начальную разметку секретных данных, что при известном интерфейсе вычислительной системы с внешним миром не является слишком сложной задачей. Например, если помеченными данными является содержимое файла на жестком диске, началом отслеживаемого потока данных могут являться функции ОС доступа к файлу, или, если интерфейс ОС неизвестен — низкоуровневые процессорные инструкции доступа к диску. Если осуществляется ввод помеченных данных с клавиатуры — началом отслеживаемого потока данных могут являться функции ОС получения данных с клавиатуры, из окна пользовательского интерфейса, или низкоуровневые процессорные инструкции чтения данных с клавиатуры. Точки вывода информации вовне вычислительной системы могут также определяться либо соответствующими функциями ОС, либо процессорными инструкциями, осуществляющими низкоуровневое взаимодействие с периферийным оборудованием.

В случае консервативного анализа требуется только начальная разметка. Для проведения полного анализа потребуется восстановление модели алгоритма, составляющего поток помеченных данных. Даже в этом случае это более простая задача, чем построение модели системы в целом, что необходимо для проведения анализа в соответствии с руководящими документами [7].

Очевидными подходами к затруднению анализа на наличие закладок являются:

- порождение большого числа выходных данных, зависящих от секретной информации, кроме существенных выходных данных;
- использование нестандартных криптографических преобразований, чтобы вынудить проводить криптоанализ для определения доступности выходных секретных данных;
- смягчение требования к обратимости алгоритма — алгоритм может быть необратим, но иметь ограниченное количество эффективно вычисляемых обратных значений;
- расширение границ анализируемой системы.

Рассмотрим подробнее последний подход. Предположим, анализ потока данных осуществляется в пределах процессора. Если помеченные данные будут сохранены на диск, а затем заново прочитаны с него, то в рамках анализа зависимостей между процессорными инструкциями сохранить пометку у прочитанных с диска данных не получится. Потребуется формализовать взаимодействие с диском. Например, определить правила распространения пометок при прохождении потока управления через функции чтения/записи с диска. Проблема — при использовании различных способов чтения/записи. Например, запись на файловом уровне, а чтение — на секторном. Другой пример — передача помеченных данных "в конверте", ключ которого не доступен вовне системы, на другой компьютер, а затем возврат таких данных обратно, "снятие конверта" и передача результата вовне системы. Для анализа такой ситуации требуется проводить "сквозной" анализ потоков данных через вычислительную систему из двух компьютеров. При анализе только одного из них, имеем незамкнутую систему — наблюдаем только часть потока управления.

В качестве примера рассмотрим схему, используемую в некоторых реальных системах — см. рис. 1.

В принципе, анализ мог быть начат с любой части потока данных, изображенного на рисунке — с записи в файл, генерации ключа или чтения открытого текста из файла — в зависимости от того, какую из этих точек аналитик обнаружит первой. Предположим, анализ начался с момента, когда аналитик обнаружил функцию AES-шифрования, преобразующую открытый текст *ОТ* в шифр-текст *ШТ* на основе ключа *Ключ1*. Шифр-текст впоследствии сохраняется в файле на диске, т.е. в нашем понимании поступает вовне анализируемой системы. Аналитик объявляет *ОТ* и *Ключ1* секретными данными (помечает их), и описывает модель функции AES (входные параметры — шифруемый буфер и ключ, выходные параметры — зашифрованный буфер, функция обратима при известных ключе и зашифрованном буфере). После этого аналитик выполняет анализ потока данных, состоящий из комбинации восходящего и нисходящего анализа. При восходящем анализе определяется, как были получены данные, при нисходящем — как они будут обработаны.

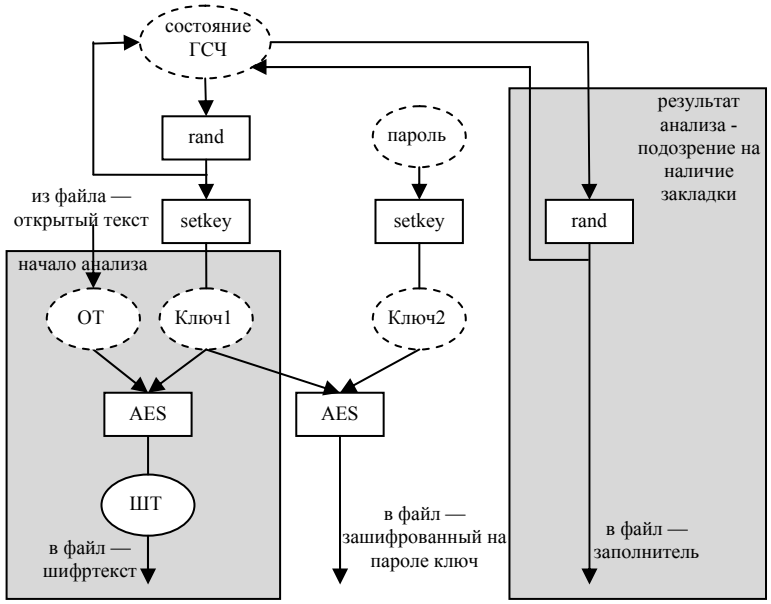

Рис. 1. Пример анализа алгоритма на наличие закладок

Анализ начинается с проверки преобразований над секретными данными (*ОТ* и *Ключ1*):

- восходящий анализ по трассе для *ОТ* показывает, что данные были прочитаны из файла, по всем промежуточным результатам чтения нисходящим анализом проверяется, что ни в каких других вычислениях они участия не принимали
- нисходящий анализ для *ОТ* показывает, что больше ни в каких вычислениях данные участие не принимали
- восходящий анализ по трассе для *Ключ1* показывает, что данные были получены как результат вызова функции setkey (инициализация ключа из ключевого материала), в свою очередь функция setkey получает данные из псевдослучайного генератора rand.
- нисходящий анализ для *Ключ1* показывает, что он шифруется на ключе *Ключ2* алгоритмом AES, результат сохраняется в файл.

По аналогии восстанавливается полная схема.

Обнаружив, что значение имеющего пометку ключа *Ключ1* зависит от *состояния генератора случайных чисел (ГСЧ)*, система анализа пометит и *состояние ГСЧ*. Поскольку по умолчанию система анализа рассматривает

любой алгоритм как обратимый (для консервативности анализа), а на помеченное *состояние ГСЧ* влияли все предыдущие состояния, все они также будут помечены. Поскольку часть из этих состояний в итоге попадают в виде заполнителя в файл на диске, соответствующий код объявляется как алгоритм, подозреваемый на наличие закладки. В дальнейшем аналитик должен выполнить анализ алгоритма ГСЧ, и если он будет обратим — найденное место действительно является закладкой.

Анализируя часть потока данных, где помеченный *Ключ1* шифруется с помощью алгоритма AES на ключе *Ключ2*, а результат попадает в файл, в итоге помечаются *Ключ2* и *пароль*. В ходе дальнейшего анализа проверяется, что никаких результатов преобразований над помеченными данными вовне системы не попадает.

Приведенный пример сильно упрощен — в случае реального анализа реализации сходного алгоритма количество проверочных точек (фактически — вершин графа потока данных), в которых осуществляется восходящий-нисходящий анализ, может достигать десятков и сотен тысяч.

6. Заключение

В работе предложен подход к поиску вредоносного кода, предназначенного для компрометации секретных данных пользователя, т.е. к нарушению конфиденциальности. В работе такой код назван закладкой, нарушающей конфиденциальность, или просто — закладкой. Необходимость в таком определении закладки возникла в силу малопригодности существующих в ГОСТ определений к формулированию подходов к ее поиску средствами анализа потоков управления и данных. Показано, что в общем случае задача поиска закладок средствами анализа потоков данных сводится к алгоритмически неразрешимым задачам. Однако в случаях, когда аналитику известны применяемые в программе алгоритмы преобразования данных, такая задача разрешима. Подобный подход может представлять практическую ценность, поскольку многие разработчики ПО при разработке используют стандартные криптоалгоритмы, которые достаточно просто распознаются в коде. Рассматриваемый подход может быть распространен и на поиск некоторых других типов закладок.

Предложенный в работе подход является развитием динамического taint-анализа и реализуется в рамках системы динамического анализа программного обеспечения по трассе исполнения программы TREX.

Литература

[1] А. Ю. Тихонов. Ключевые технологии, способствующие решению задач восстановления и анализа алгоритмов. Журнал Безопасность Информационных Технологий № 2008-1.

[2] Тихонов А.Ю., Аветисян А.И., Падарян В.А. Извлечение алгоритма из бинарного кода на основе динамического анализа. // Труды XVII общероссийской научно-технической конференции «Методы и технические средства обеспечения безопасности информации». Санкт-Петербург, 07-11 июля 2008 г. Стр. 109.

[3] В.А. Падарян, А.И. Гетьман, М.А. Соловьев. Программная среда для динамического анализа бинарного кода. Труды Института системного программирования РАН, том 17, 2009 г. Стр. 51-72.

[4] Balakrishnan, G. and Reps, T., WYSINWYX: What You See Is Not What You eXecute. In ACM Transactions on Programming Languages and Systems, vol.32 Issue 6, August 2010.

[5] Edward J. Schwartz, Thanassis Avgerinos, David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask), Proceedings of the 2010 IEEE Symposium on Security and Privacy.

[6] ГОСТ Р 51275-2006 — Защита информации. Объект информатизации. Факторы, воздействующие на информацию.

[7] Руководящий документ "Защита от несанкционированного доступа к информации. Часть 1. Программное обеспечение средств защиты информации. Классификация по уровню контроля отсутствия недекларированных возможностей". Утверждено решением председателя Государственной технической комиссии при Президенте Российской Федерации от 4 июня 1999 г. № 114.

[8] ГОСТ Р 50922-2006 Защита информации Основные термины и определения.

[9] ГОСТ Р 17799-2005 Информационная технология. Практические правила управления информационной безопасностью.