Аспектно-ориентированная реализация модели защиты программ на основе “удаленного доверия”[1]

В.А. Десницкий, И.В. Котенко

Аннотация. В работе рассматривается аспектно-ориентированный подход (АОП) к реализации динамического замещения мобильного модуля в модели защиты программного обеспечения от несанкционированных изменений и вмешательств на основе механизма «удаленного доверия». Главной целью исследования является разработка механизма, позволяющего осуществлять изменения защищаемого программного кода динамически, без перезагрузки и приостановки выполняющегося приложения.

Ключевые слова. Защита ПО, атака, аспектно-ориентированное программирование (АОП), АОП-машина.

Введение

Используемое в настоящее время программное обеспечение (ПО) является объектом многочисленных атак, направленных на изменение его поведения со стороны злоумышленников. Как правило, злонамеренный пользователь преследует цель получения каких-либо дополнительных преимуществ, не предусмотренных разработчиком программы. В частности, такая атака может заключаться в формировании работоспособного кода программы с изменением некоторых его функций. Так, например, это может быть попытка реализации доступа пользователей к программе без наличия соответствующего пароля или сертификата.

Разработка методов защиты ПО от несанкционированных изменений, которые были бы адекватны текущим угрозам, является одной из актуальных задач в области компьютерной безопасности. В частности, необходимость разработки адекватных средств защиты ПО, устойчивых к различного рода атакам, определяется потребностью разработчиков обеспечить защиту авторских прав на производимое ПО.

Настоящая статья посвящена разработке и анализу механизма замещения мобильного модуля в рамках общего подхода к защите ПО на основе *механизма «удаленного доверия»*. Цель этого подхода – обнаружение несанкционированных изменений клиентской программы, функционирующей в потенциально враждебном окружении. Данный подход предполагает наличие клиентской программы, требующей защиты и выполняющейся в пределах ненадежного окружения, а также надежного сервера, расположенного на защищенном хосте. Также предполагается постоянное сетевое соединение между клиентской программой и сервером.

В соответствии с механизмом «удаленного доверия» в клиентскую программу должен встраиваться переносимый (мобильный) модуль, содержащий два программных компонента, монитор и генератор цифровых подписей.

[1] Работа выполнена при финансовой поддержке РФФИ (проект №07-01-00547), программы фундаментальных исследований ОИТВС РАН (контракт №3.2/03), Фонда содействия отечественной науке и при частичной финансовой поддержке, осуществляемой в рамках проекта Евросоюза RE-TRUST (контракт № 021186-2).

Монитор ответственен за выполнение разнообразных верификаций программы. Результаты верификаций передаются второму компоненту, который непрерывно генерирует подписи – данные, характеризующие текущее состояние клиентской программы, и отправляет их на надежный сервер. Получив подписи, сервер анализирует их и, в соответствии с этим, принимает решение о том, было ли совершено вмешательство в работу программы или нет. При отсутствии вмешательств клиентская программа признается корректной, после чего ей могут, например, предоставляться дополнительные сервисы и программные обновления.

Одной из задач, связанных с разработкой механизма защиты программного обеспечения от несанкционированных изменений и вмешательств на основе механизма «удаленного доверия» [1, 7], является создание механизма, обеспечивающего возможность динамического замещения мобильного модуля. Цель такого динамического замещения - усложнение возможности успешного выполнения атак на механизм «удаленного доверия». Отличительная особенность замещаемого модуля состоит в том, что он не поставляется вместе с клиентской программой, а загружается отдельно с надежного сервера и периодически полностью обновляется. Регулярное обновление модуля необходимо выполнять для того, чтобы ограничить время, которое мог бы использовать злоумышленник для осуществления атак на модуль.

Существующие и используемые в настоящее время методы защиты ПО достаточно легко нейтрализуются злоумышленниками за относительно небольшое время. Для этого они используют различные программные инструменты: отладчики, дамперы, снифферы, эмуляторы и т.п. Как показывает практика, любая защита, основанная исключительно на повышении сложности декомпиляции программы, а также понимания ее структуры злоумышленником, преодолевается без существенных преград.

На основе подхода, представленного в настоящей статье, предлагается осуществлять защиту программы посредством внедрения в ее код специального компонента защиты, который регулярно обновляется. Это делает недостаточным однократное изменение программы противником с целью дальнейшего ее злонамеренного исполнения.

1. Сущность и основные понятия АОП

Аспектно-ориентированное программирование позволяет проводить статические или динамические изменения программы посредством внедрения небольших фрагментов кода, называемых аспектами.

В соответствии с концепцией АОП различные особенности поведения (behavioral features) системы программируются отдельно, в наиболее естественном для них виде, и затем вплетаются в целевой код [10].

В целом, основная польза АОП заключается в возможности выделения отдельного процесса разработки, ответственного за каждую такую особенность, а также в уменьшении количества зависимостей между кодом, реализующим особенности поведения, и основным кодом программы. Все это делает процесс разработки приложений более гибким, причем уменьшается сложность модификации кода и увеличивается возможность повторного использования кода, реализующего конкретную поведенческую особенность.

Для реализации механизма замещения мобильного модуля преимуществом, предоставляемым АОП, является возможность динамически, главным образом, во время выполнения программы, добавлять, изменять и удалять дополнительные фрагменты кода в защищаемой программе.

Базовым понятием АОП является понятие *перекрестной функциональности (crosscutting concern)*. Это – функциональность, обращение к которой и ее реализация разбросаны (scattered) по всему коду программы (или значительной ее части). В некоторых работах [10] данное понятие описывается как программная сущность, которая в рамках данного языка не может естественным образом быть выражена в виде отдельной обобщенной процедуры (generalized-procedure), метода, объекта и т.п. (в зависимости от конкретного языка программирования). Таким образом, реализуемая функциональность распределена (scattered) по программному коду, в результате чего он становится запутанным

(tangled) и трудным для дальнейшей обработки и сопровождения.

В основу АОП заложен *принцип разделения функциональностей (separation of concerns)*, в соответствии с которым осуществляется выделение реализуемых функциональностей поведения в отдельные программные единицы (units). В АОП такие программные единицы называются аспектами.

Аспект представляет собой набор методов, реализующих требуемую функциональность, и описания точек в программе, которые будут ассоциированы с этими методами. При достижении потоком управления этих точек будут срабатывать данные методы.

Аспекты могут внедряться в приложение на разных стадиях работы:

- на стадии компиляции приложения;
- во время загрузки приложения операционной системой;
- во время его выполнения.

Под АОП-машиной (AOP engine) понимается, в зависимости от реализации, интерпретатор или специальная программная оболочка, которые позволяют выполнять аспектный код [8].

Динамические АОП-машины позволяют во время выполнения создавать новые аспекты, внедрять их в работающее приложение и при необходимости удалять их из приложения. Тем самым реализуемая программная функциональность может динамически включаться в приложение, предоставляя все необходимые возможности, а также удаляться из него.

АОП позволяет делать программные зависимости односторонними – аспект зависит от остальной части программы, в то время как программа почти или совсем не зависит от аспекта и может выполняться как с внедренным аспектом, так и без него.

Представим основные понятия АОП:

- *aspect (аспект)* – базовая программная единица, которая является основной конструкцией по внедрению новых функциональностей в код программы;
- *join points* – однозначно определенные точки в коде программы, например, вызовы методов, обращения к полям класса и т.п.;
- *pointcuts* – множество однотипных точек соединения, то есть точек, которые связаны с одним и тем же фрагментом исполняемого кода; pointcut может задаваться явно или как комбинация других pointcut’ов;
- *advice* – вплетаемый в программу фрагмент кода аспекта, который может выполняться до и (или) после каждой точки соединения определенного pointcut’а;
- *weaving mechanism (механизм вплетения)* – механизм вплетения кода аспекта в программу, то есть механизм, связывающий точки соединения из pointcut с advice кодом;
- *inter-type declaration* - внутри-типовое объявление, дающее возможность модифицировать статическую структуру программы, вводя поля и методы в классы и интерфейсы в рамках аспекта.

В различных АОП-подходах используются различные *join point-модели*, которые определяют способы определения точек соединения [17]. Точки соединения могут задаваться лексически как некоторые инструкции, имеющие определенное положение в исходном тексте программы. Применение АОП-машин, основанных на таком задании точек соединения, в случае их совместного использования с некоторыми дополнительными средствами защиты, например, с инструментами обфускации, должно происходить с осторожностью. Действительно, применение обфусцирования программы после задания точек соединения может полностью нарушить механизм вплетения аспектов, сделав его неработоспособным.

Точки соединения могут определяться как некоторые run-time-сущности [12], такие как события, которые происходят во время выполнения программы, например, вызовы методов, исключительные ситуации или другие события, связанные с обработкой потока управления. Существуют АОП-реализации, в которых точки соединения задаются посредством метаданных, например, Java-аннотаций. Однако мета–данные являются легко обнаружимыми злоумышленником посредством рефлексии, что способствует проведению атаки по отделению мобильного модуля и защищаемой программы.

В предлагаемой модели реализации механизма замещения мобильного модуля при помощи аспектов реализуются функции верификации и генератор подписей (тегов) загружаемого моду-

ля, встраиваемые в код клиентской программы. Таким образом, использование АОП обеспечивает возможность динамической загрузки мобильного модуля на клиентскую программу и его выгрузку впоследствии.

Обычно АОП рассматривается как надстройка над концепцией объектно-ориентированного программирования (ООП). АОП добавляет дополнительное измерение, измерение функциональностей, которые независимы от классов и объектов ООП. Тем самым измерение функциональностей является как бы ортогональным к ООП. В результате, в дополнение ко всем преимуществам ООП добавляется возможность выделения поведенческих функциональностей в виде отдельных программных единиц – аспектов (Рис 1).

При реализации механизма замещения важной задачей является правильный выбор подходящей АОП-машины, которая предоставляла бы необходимые средства для построения и функционирования мобильного модуля.

Реализации АОП-машин могут быть основаны как на преобразованиях, выполняемых над исходным кодом, так и на преобразованиях исполняемого кода [13].

В последующих разделах будут рассмотрены различные подходы, в соответствии с которыми может осуществляться механизм “вплетения” (внедрения) аспектов (aspect weaving) при реализации механизма динамического замещения мобильного модуля.

2. Механизм динамического замещения

В работе предполагается, что алгоритмы верификации загружаемого модуля и данные, используемые ими, а также алгоритм генерации подписей должны изменяться от версии к версии (Рис.2). Это позволяет, в случае успешного выполнения злоумышленником атаки на модуль, сделать невозможным применение определенного способа взлома модуля после следующего его обновления [1, 7].

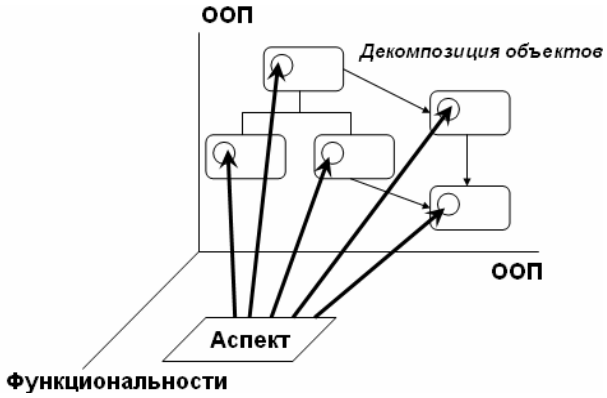

Рис.1. Визуальное представление АОП

Рассмотрим основные виды атак, которые могут осуществляться злоумышленником для компрометации программы, защищаемой при помощи механизма «удаленного доверия». Основной атакой является атака «обратной разработки» (reverse-engineering). Атака такого вида предполагает дизассемблирование бинарного кода клиентской программы и его декомпиляцию. Такая атака позволяет злоумышленнику получить исходный код программы с целью ее дальнейшего исследования и последующего выполнения атак других видов.

В целом, основными возможными атаками, предназначенными для компрометации программы, являются атаки, которые направлены на изменение бинарного кода программы, модификацию среды выполнения программы при помощи эмуляторов и отладчиков, динамическое изменение состояния программы без изменения ее кода, перехват и подмену сетевых сообщений в процессе коммуникации клиент-

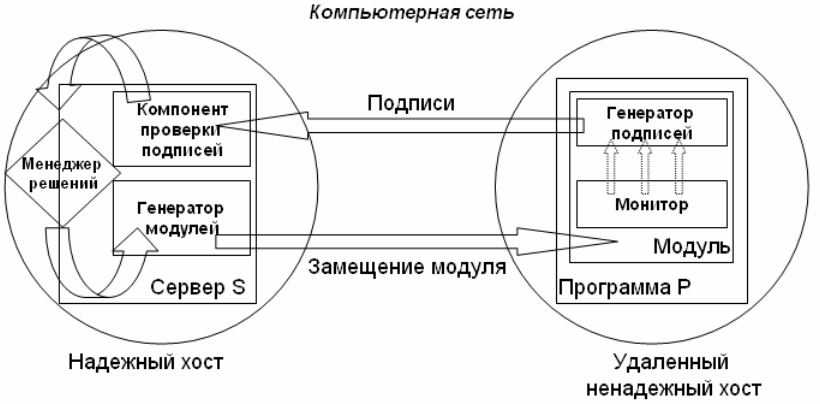

Рис. 2. Схема модели защиты основе механизма “удаленного доверия”

ской программы и сервера. Актуальной также является атака «дублирования», которая предполагает наличие нескольких работающих одновременно экземпляров клиентской программы, причем один из них является корректным, не измененным злоумышленником экземпляром программы, а другой – несанкционированно модифицированным. Сложность защиты от подобного вида атак заключается в практической невозможности гарантировать, что модифицированный экземпляр программы не будет использовать все дополнительные программные сервисы, предоставляемые законным образом первому экземпляру программы надежным сервером.

Важным требованием является превентивная защита модуля от возможных атак. В частности, предполагается использование разнообразных методов обфускации с целью препятствования осуществлению атак. Применение обфускации позволяет существенно усложнить понимание злоумышленником структуры программы и ее функциональностей. Тем не менее, достаточно непростой задачей является оценивание сложности, вносимой конкретным методом обфускации в общую совокупность используемых защитных механизмов.

Двумя дополнительными требованиями, которые делают клиентскую программу более устойчивой к взломам, являются реализация взаимопроникновения кода модуля и остальной части кода клиентской программы, а также сокрытие кода модуля, направленное на максимально возможное усложнение выполнения «обратной разработки» злоумышленником.

Таким образом, в рамках механизма замещения монитора и генератора подписей в клиентской программе требуется периодически во время выполнения добавлять и удалять методы, ответственные за верификацию, а также структуры данных, связанные с ними. Для реализации этих целей, то есть построения и обеспечения функционирования мобильного кода, предлагается использовать средства аспектно-ориентированного программирования.

Атаки, целью которых является компрометация механизма динамического замещения, могут осуществляться как на стадии установки модуля в программу, так и в процессе его функционирования. Одной из наиболее интересных является атака по выделению модуля из программы, направленная на получение его кода с целью дальнейшего изучения [7]. В частности, такая атака может включать в себя обнаружение точек в программе, из которых происходит обращение к методам модуля.

Исследование злоумышленником структуры модуля является необходимым условием проведения другого класса атак на мобильный модуль. Данный класс атак может выполняться злоумышленником с использованием методов статического и динамического анализа модуля в контексте защищаемой программы.

Следует также отметить атаку на модуль, целью которой является несанкционированное изменение его кода. Атака такого класса может включать в себя как изменение положения точек в программе, из которых вызываются методы модуля, так и непосредственно изменение кода методов модуля. Для осуществления такого рода действий злоумышленник может использовать отладчики бинарного кода, эмуляторы, декомпиляторы и дизассемблеры.

Для защиты модуля от описанных выше атак могут использоваться любые традиционные методы защиты кода. В частности, возможно применение разнообразных анти-отладочных приемов, а также приемов, направленных на препятствование дизассемблированию, методов обфускации [11] и т.п.

В настоящее время предметом многочисленных исследований являются различные технологии динамического изменения кода, а также методики само-модификации программ. Однако в настоящей работе исследуется только метод динамического замещения кода на основе парадигмы АОП. В работе [7] была предложена постановка данной задачи и приведен ряд начальных требований к необходимой для ее решения АОП-реализации. В то же время данная работа не дает ясного понимания, какие АОП-реализации из всего разнообразия существующих могли бы быть использованы, а какие являются непригодными для поставленной задачи.

Далее в работе представлен всеобъемлющий анализ существующих на данный момент АОП-подходов на предмет их возможного использо-

вания для реализации динамического замещения мобильного модуля. Рассматриваемые АОП-подходы снабжены ссылками на их конкретные программные реализации. Выработан набор критериев, которым должна удовлетворять наиболее приемлемая с точки зрения безопасности и возможности использования АОП-реализация.

Использование АОП для реализации мобильного модуля должно также способствовать его сокрытию в рамках кода программы. Это содействует реализации требуемого принципа сокрытия модуля (module hiding) и усложняет возможность выполнения атак по выделению кода модуля и его модификации, основанных на обратной разработке. Однако степень сокрытия зависит, главным образом, от свойств конкретной реализации АОП.

3. Внедрение аспектов при компиляции

Внедрение аспектов на стадии компиляции (compile-time-подход) является наиболее широко распространенным и изученным подходом. При данном подходе исходный код приложения "сливается" с кодом аспекта до или во время компиляции. В результате получается новая измененная версия кода приложения, включающая функциональность аспекта.

Такой подход может быть основан на использовании специального препроцессора, который внедряет код аспектов в исходный код перед стадией компиляции. В этом случае АОП-машина во время компиляции должна производить семантически корректную программу, так как она должна соответствовать используемому компилятору [13].

Данный подход может также предполагать интеграцию АОП-машины и компилятора с языка исходного кода, когда измененный код используется в процессе сборки, в результате которой и формируется целевое приложение.

Примером АОП-машины с compile-time-подходом является система AspectJ. Это - базирующаяся на Java реализация, имеющая препроцессор ajc. Другим примером может служить система ASPECT.NET, представляющая собой .NET реализацию, в которой на уровне исходного кода аспекты задаются при помощи специального мета-языка – Aspect.NET.ML, а на уровне IL.NET (Intermediate Language – аналог java байт-кода) аспекты представляются при помощи атрибутов (*custom attributes*), внедряемых в целевые .NET сборки (assemblies) [14]. В результате в обоих случаях выполняемый код (java байт-код и .NET IL, соответственно) не требует никаких дополнительных расширений для поддержки возможностей АОП.

Следует отметить, что compile-time подход не пригоден для реализации механизма динамического замещения модуля, так как он не позволяет осуществлять необходимые для изменений мобильного модуля модификации кода во время выполнения.

4. Внедрение аспектов во время запуска

Среди динамических АОП-подходов необходимо выделить подход, в соответствии с которым аспекты внедряются в приложение во время его запуска (load-time).

Для этого используется специализированный программный инструментарий. В частности, для Java-программ такими инструментами являются BCA (Binary Component Adaptation) [9], JOIE (Java Object Instrumentation Environment) [5] и Javassist [16].

Построенные на основе такого инструментария АОП-машины часто используются для адаптации программ под специфические требования конкретных пользователей, а также для решения задач, связанных с эволюцией и интеграцией программных компонентов [13].

Недостатком данного подхода для реализации механизма динамического замещения является то, что изменение кода мобильного модуля происходит единожды – во время запуска программы, тогда как, в соответствии с требованиями механизма защиты, такие изменения должны происходить регулярно, через определенные промежутки времени.

Среди других динамических АОП-подходов следует упомянуть так называемый «hook-based»-подход. Он основан на внедрении в код программы специальных «вплетаемых» заглушек (weaving hooks) – элементов программного кода, которые служат реализацией точек соединения. Предполагается, что заглушки вне-

дряются не позже, чем на стадии запуска программы, тогда как присоединение к ним advice-кода происходит на стадии выполнения программы [6, 8]. Таким образом, отличие этого подхода состоит в том, что в нем происходит вплетение элементов кода, определяющих места вызова advice-методов, а не вплетение самих advice-методов.

Выделяют две различные стратегии, согласно которым может осуществляться механизм вплетения аспектов при «hook-based»-подходе.

Первая стратегия, называемая «total hook weaving» [4], основывается на снабжении заглушкой каждой возможной точки в программе. Данная стратегия обеспечивает возможность присоединения advice-кода в любом заранее не определенном месте в программе.С точки зрения реализации механизма динамического замещения это, несомненно, является преимуществом. Действительно, в такой ситуации программные методы каждой новой версии переносимого модуля могут прикрепляться к защищаемой программе в различных участках ее кода. Это препятствует выполнению злоумышленником атак, направленных на выделение модуля и последующего блокирования его работы.

Однако ощутимым недостатком данной стратегии является существенное снижение производительности всей программы. Возникает так называемая «проблема пустых заглушек» (the empty hook problem), состоящая в том, что подавляющее большинство заглушек, как правило, оказываются пустыми, то есть они никогда не срабатывают. А так как перед выполнением каждой инструкции проверяется прикреплен ли к данной точке какой-либо advice-код, этот метод оказывается неэффективным.

Вторая стратегия, «actual hook weaving» [4], предлагает внедрять заглушки только в действительно «интересные» точки в программе, а не в каждую возможную, потенциально нужную точку. Другими словами, предлагается существенно ограничить выбор местоположения точек соединения в программе, тем не менее, оставляя возможность отыскать точку соединения в любом фрагменте кода определенного размера. Поэтому данная стратегия представляет собой некоторый компромисс между безопасностью и производительностью программы.

«Hook-based»-подход может быть применим как к управляемым, так и неуправляемым средам выполнения.

Примером реализации такого подхода для платформы Java является Jac AOP Framework [15]. В Jac точки соединения, соответствующие некоторому pointcut, задаются в рамках специального выражения сопоставления (matching expression), которое представляет собой регулярное выражение. Такое определение точек соединения происходит на основе соглашений именования классов, объектов и методов посредством анализа байт-кода на стадии загрузки программы, а также при помощи средств рефлексии (Java Reflection) во время выполнения.

DAO C++ (Dynamic Aspect Oriented C++) – реализация hook-based подхода для C++ на основе MOP (Meta-object Protocol). Здесь для задания точек соединения вводится небольшое расширение исходного языка, для обработки которого используется специальный препроцессор [3]. В архитектуре DAO C++ препроцессор действует перед стадией компиляции и используется для генерации мета-объектных (meta-object) данных, которые будут использоваться АОП-машиной во время выполнения для нахождения нужных классов и методов. Препроцессор изменяет исходный код и вставляет заглушки (hooks), необходимые для поддержки динамического присоединения advice-кода. После компиляции и загрузки выполняющаяся программа будет содержать meta-object-данные о классах и методах программы. Аналогично Jac, на основе механизма выражений сопоставления (matching expression) эта информация используется для обеспечения динамической привязки и удаления аспектов АОП-машиной.

В рамках разрабатываемой модели защиты с hook-based-реализацией механизма мобильного модуля злоумышленник может произвести анализ кода поставляемого модуля и получить выражение сопоставления некоторого аспекта, являющегося составной частью модуля. Имея выражение сопоставления, злоумышленник способен провести статический анализ кода программы и таким образом выполнить атаку по обнаружению точек соединения, соответствующих данному аспекту.

5. Внедрение аспектов во время выполнения

В соответствии с требованиями runtime-подхода (во время выполнения приложения), механизм вплетения и выполнения аспектов встраивается непосредственно в среду выполнения.

Таким образом, среда выполнения, снабженная механизмом внедрения и обработки аспектов, будет иметь отличающуюся от обычной модель выполнения программ. А именно, для каждой выполняющейся инструкции, если счетчик команд указывает на точку соединения, выполняется определенное дополнительное действие, представляющее соответствующий код аспекта. Для этого используемая среда выполнения должна предоставлять интерфейс (Aspect-интерфейс) для привязывания аспектов во время выполнения. В отличие от compile-time-подхода в данном подходе аспекты и программа располагаются, как правило, отдельно друг от друга и функционируют независимо. Экземпляры аспектов хранятся вне целевого приложения и способны своевременно вплетаться в него в любой точке программы. Основная операция Aspect-интерфейса – это привязка экземпляра аспекта к программе. Этот интерфейс определяет способ создания, регистрации и де-регистрации точек соединения и соответствующие действия [13].

Среди АОП-runtime-подходов можно выделить *подход, основанный на использовании средств отладчика*. В этом случае реализация Aspect-интерфейса использует интерфейс отладчика (Debugger-интерфейс), который предоставляет среда выполнения, работающая в связке с отладчиком.

В частности, в рамках платформы Java существующий интерфейс JVMDI (JVM Debugger Interface) [18] дает возможность управлять работающей Java-программой [8, 13]. Debugger-интерфейс позволяет устанавливать контрольные точки (breakpoints), регистрировать запросы на выполнение событий, происходящих в программе, и получать уведомления в случае их осуществления. Таким образом, при выполнении определенных условий приостанавливается процесс выполнения программы, совершаются определенные действия, после чего выполнение программы возобновляется. В результате, в данном подходе точки соединения могут задаваться при помощи контрольных точек, которые инициируются определенными событиями, определяемыми как условия достижения инструкции определенного вида, имеющей определенные параметры и модификаторы.

С точки зрения задачи построения механизма динамического замещения, преимуществом данного подхода является то, что точки соединения не устанавливаются заранее, а задаются в процессе выполнения, что существенно затрудняет их обнаружение злоумышленником. К одному из недостатков подхода можно отнести достаточно слабую степень взаимопроникновения advice-кода и кода приложения, тем самым требуемое свойство сокрытия модуля будет выполняться достаточно слабо. Другой недостаток данного подхода состоит в том, что в соответствии с принципом работы отладчика для выполнения любого аспекта необходимо приостанавливать все процессы данного приложения.

Рассмотрим так называемый *JIT (Just-in-time)-подход*, в соответствии с которым реализуются некоторые runtime АОП-машины для управляемого кода. Все программные изменения, связанные с вплетением аспектов, происходят во время выполнения программы в те промежутки времени, когда JIT-компилятор [18] преобразует объектный код в native-код.

В соответствии с JIT-подходом преобразовываться в native-код могут как классы, так и отдельные методы, и даже произвольные фрагменты программного кода. Одним из условий механизма защиты является требование взаимопроникновения кода модуля и кода защищаемой программы. Для этого нужно, чтобы преобразуемый к native-виду фрагмент кода содержал не только код модуля, но и некоторую часть кода программы. Таким образом это обеспечивает определенную степень интеграции модуля и приложения, тем самым препятствуя выполнению атак по выделению модуля из программы и блокирования его работы.

6. Реализация динамического замещения

Сформулируем общие критерии, позволяющие определить, может ли некоторая АОП-

машина быть использована для реализации механизма динамического замещения мобильного модуля. Знание этих критериев является важным как для поиска подходящей АОП-реализации в рамках конкретной платформы или языка, так и для формирования требований к АОП-машинам, которые могут быть созданы в будущем. Эта информация также может способствовать выполнению анализа надежности разрабатываемого механизма защиты, в частности, оценке его стойкости к соответствующим атакам.

Перечислим вербально основные критерии возможности применения АОП-машин:

- возможность задания новых точек соединения во время выполнения программы;
- возможность вплетения, изменения и (или) удаления advice-кода во время выполнения программы;
- предотвращение обнаружения и блокирования злоумышленником процесса выполнения аспектного кода, для чего точки соединения должны быть скрыты в программе; во время работы код аспектов должен располагаться в пределах защищаемой программы и быть скрыт в ней; аспекты должны выполняться в пределах основного процесса приложения, а не в виде отдельного процесса;
- с точки зрения приложения операция вплетения должна выглядеть как один атомарный шаг; если она не является атомарной, то за один шаг advice добавляется только в одну точку соединения, а это может привести к ситуации, когда в половине случаев выполняется дополнительная функциональность, тогда как в другой половине все еще используется старый код.

Сформулированные критерии в наиболее полном виде реализуются в runtime АОП-подходах: *Debugger*-подходе и *JIT*-подходе. Необходимость использования именно runtime-подходов для реализации механизма динамического замещения обуславливается, главным образом, первыми двумя критериями.

Таким образом, программные методы мобильного модуля, в том числе методы верификации, должны реализовываться в виде advice-кода аспектов, которые могут динамически создаваться и удаляться во время выполнения программы.

В результате, в случае Debugger-подхода аспекты привязываются к программному приложению при помощи средств отладчика, поэтому программа должна запускаться в отладочном режиме.

Стоит отметить, что Debugger-подход не является оптимальным, для него третий критерий реализуется лишь частично – аспекты выполняются в виде отдельного процесса; поэтому возможной атакой является блокировка работы данного процесса с целью препятствования функционированию модуля.

JIT-подход применим исключительно в случае управляемых средств выполнения, позволяющих компилировать программный код «на лету».

Заключение

В данной работе рассмотрены основные АОП-подходы к реализации механизма динамического замещения мобильного модуля и связанные с ними понятия. В работе сформулирован список критериев, которые позволяют оценить конкретную АОП-машину с точки зрения требований защиты и реализации динамического замещения мобильного модуля в модели защиты программного обеспечения от несанкционированных изменений и вмешательств на основе механизма «удаленного доверия». Наиболее полно поставленным требованиям удовлетворяют два runtime-подхода, рассмотренные выше – подход, основанный на использовании средств отладчика, и JIT-подход.

Основным результатом работы являются представление подхода по реализации механизма динамического замещения мобильного модуля при помощи АОП-концепции, а также анализ различных АОП-подходов и выбор двух из них (*Debugger*-подхода и *JIT*-подхода), наиболее полным образом удовлетворяющих предъявляемым критериям.

Помимо формального выполнения четырех представленных выше критериев предельно важной также является степень сокрытия advice-кода в программе. Сложность обнаружения кода аспектов злоумышленником будет гарантировать сложность осуществления атак на модуль. С теоретической точки зрения выработка метрик по определению величины сокры-

тия модуля и взаимопроникновения его кода и кода программы представляется достаточно сложной задачей, тем не менее, различные АОП-подходы могут сравниваться также на основе эмпирических методов.

Литература

1. Десницкий В.А., Котенко И.В. Модели удаленной аутентификации для защиты программ // Труды Международных научно-технических конференций “Интеллектуальные системы (AIS'07)” и “Интеллектуальные САПР (CAD-2007)”. М.: Физматлит, 2007. С.43-50.
2. Десницкий В.А. Реализация механизма замещения мобильного модуля на основе парадигмы аспектно-ориентированного программирования // V межрегиональная конференция «Информационная безопасность регионов России (ИБРР-2007). Материалы конференции. СПб, 2007. С.49-50.
3. Almajali S., Elrad T. A Dynamic Aspect Oriented C++ using MOP with Minimal Hook Weaving Approach // Proceedings of the 2004 Dynamic Aspects Workshop (DAW04) as part of AOSD'04. Lancaster, UK, March 2004.
4. Chitchyan R., Sommerville I. Comparing Dynamic AO Systems // Proceedings of the 2004 Dynamic Aspects Workshop. Technical Report RIACS Technical Report No.04.01, RIACS, 2004.
5. Cohen G., Chase J., Kaminsky D. Automatic program transformation with JOIE // 1998 USENIX Annual Technical Symposium, 1998. P.167-178.
6. Eaddy M., Aho A., Hu W., McDonald P., Burger J. Debugging Aspect-Enabled Programs // International Symposium on Software Composition (SC 2007), Braga, Portugal, 2007.
7. Falcarin P., Baldi M., Mazzocchi D. Software Tampering Detection using AOP and mobile code // In Workshop on AOSD Technology for Application level security (AOSDSEC), Lancaster, UK, 2004.
8. Frei A., Grawehr P., Alonso G. A Dynamic AOP-Engine for .NET // Tech Rep 445. Dept. of CS, ETH Zurich, 2004.
9. Keller R., Holzle U. Binary Component Adaptation // Lecture Notes in Computer Science, 1998.
10. Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C.V., Loingtier J-M., Irwin J. Aspect-Oriented Programming. Springer-Verlag, 1997.
11. Linn C., Debray S. Obfuscation of Executable Code to Improve Resistance to Static Disassembly // In Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS) , October 2003. P.290-299.
12. Masuhara K., Kiczales G., Dutchyn C. Compilation semantics of aspect-oriented programs // In FOAL 2002 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2002, G. T. Leavens and R. Cytron, Eds. Tech. Rep. 02-06. Department of Computer Science, Iowa State University, 2002. P.17-26.
13. Popovici A., Gross T., Alonso G. Dynamic Weaving for Aspect Oriented Programming // 1st International Conference on Aspect-Oriented Software Development (AOSD), Enschede, The Netherlands, 2002.
14. Safonov V.O., Gratchev M.K., Grigoryev D.A., Maslennikov A.I. Aspect.NET — aspect-oriented toolkit for Microsoft.NET based on Phoenix and Whidbey // “.NET Technologies 2006” International Conference Proceedings, Pilsen, Czechia, 2006.
15. Jac AOP Framework. http://jac.objectweb.org .
16. Javassist AOP Framework. http://www.csg.is.titech.ac.jp/~chiba/javassist .
17. The Aspect-Oriented Software Association. http://www.aosd.net/ .
18. Java Technolog. http://www.java.sun.com .

Десницкий Василий Алексеевич. Аспирант Санкт-Петербургского института информатики и автоматизации РАН. Окончил Санкт-Петербургский государственный университет в 2006 году. Имеет 15 печатных работ. Область научных интересов: методы защиты программного обеспечения, политики безопасности, объектно-ориентированные паттерны.
E-mail: desnitsky@comsec.spb.ru.

Котенко Игорь Витальевич. Руководитель НИГ компьютерной безопасности Санкт-Петербургского института информатики и автоматизации РАН. Доктор технических наук, профессор. Имеет более 450 печатных работ. Область научных интересов: защита информации, искусственный интеллект, телекоммуникационные системы.
E-mail: ivkote@ comsec.spb.ru.