А. С. Моляков

KPROCESSOR-CID-TABLE-ФАКТОРИНГ – НОВЫЙ МЕТОД КОНТРОЛЯ, ОБЕСПЕЧЕНИЯ ДОСТОВЕРНОСТИ И ЗАЩИТЫ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ КОМПЬЮТЕРНЫХ СИСТЕМ

Введение

В настоящее время основная проблема в создании защищенных систем нового поколения заключается в том, что современная платформа недалеко ушла от компьютерных архитектур, операционных систем и языков программирования 1960–1980-х годов. Среда вычислений тех времен сильно отличалась от современной. Компьютеры были весьма ограничены в скорости и объеме памяти; они использовались только малыми группами технически грамотных и незлонамеренных пользователей; они редко объединялись в сети или общались с физическими устройствами. Сейчас все не так, но современные архитектуры компьютеров, операционные системы и языки программирования недостаточно изменились для того, чтобы отражать фундаментальные изменения в компьютерах и их использовании. Стремительно развивающиеся аппаратные средства обычно являются движущей силой фундаментальных изменений систем и приложений. Программное обеспечение, развивающееся медленнее, редко создает возможности для фундаментальных усовершенствований. Однако программное обеспечение действительно развивается, и его изменение делает возможным — и необходимым — пересмотр старых подходов. Менее исследовано, как расширенные программные механизмы способствуют глубоким изменениям в системной архитектуре, которая, в свою очередь, могла бы приблизить достижение такой цели, как устранение дефектов ПО и смягчение их последствий [1]. Расширение, как правило, состоит из кода, динамически загружаемого в адресное пространство родительского приложения. Обладая прямым доступом к внутренним интерфейсам и структурам данных родительского приложения, расширение может предоставлять богатую функциональность. Расширения — одна из главных проблем надежности, безопасности и обратной совместимости ПО. Несмотря на то что код расширений зачастую может быть непроверенным или некачественным, поступить из непроверенного источника, а то и просто быть вредоносным, он грузится напрямую в адресное пространство программы без четкого разграничения между кодом родительской программы и расширения. Результат часто бывает плачевным. Например, согласно отчету Swift, 85 % диагностированных падений Windows вызваны плохими драйверами устройств. Более того, поскольку четкой границы нет, расширение может использовать нераскрываемые внутренние аспекты реализации родительского приложения, что может сдерживать эволюцию программы и требовать экстенсивного тестирования для исключения несовместимости. Динамическая загрузка кода взимает еще один, менее очевидный налог на производительность и корректность. ПО, способное загружать код, — это открытая среда, в которой невозможно сделать четкие предположения о состоянии системы или правильности переходов. Рассмотрим виртуальную машину Java (JVM). Прерывание, исключение или переключение потоков могут вызвать код, который загружает новый файл, переписывает тело класса или метода и модифицирует глобальное состояние. В общем, единственный реальный способ анализировать программу, исполняющуюся в таких условиях, — начать с неявного предположения, что среда не может произвольно измениться между двумя любыми операциями. Альтернативой является запрет загрузки кода и изоляция динамически создаваемого кода в его собственной среде. Предыдущие попытки реализации этих процедур не завоевали широкой популярности, так как механизмы изоляции имели проблемы с программируемостью и производительностью, делавшие их менее привлекательными, чем выполнение без изоляции. Наиболее распространенный механизм — это традиционный процесс ОС,

но его высокая стоимость ограничивает его применимость. Аппаратное управления памятью защищает состояние процессора, но также удорожает передачу управления и данных между процессами. Более новые системы, такие как JVM или Microsoft Common Language Runtime (CLR), проектировались с учетом использования расширяемости и языковой безопасности, а не аппаратного обеспечения в качестве механизма изоляции вычислений, выполняемых в одном адресном пространстве. Сами по себе безопасные языки не гарантируют изоляции. Совместно используемые данные могут проложить широкую тропу между объектными пространствами, при этом механизмы рефлексии могут разрушить абстракцию данных и сокрытие информации. Как следствие, подобные системы включают сложные механизмы и политики безопасности, такие как контроль доступа в Java или безопасность доступа к коду в CLR, ограничивающие доступ к механизмам и интерфейсам системы. Эти механизмы сложны в использовании. Более того, в публикациях Научно-исследовательского института Карнеги-Меллона описываются лишь базовые подходы контроля сигнальных событий (мьютексы, спин-блокировки, временные отметки инициализации в списках процессов ядра). В современных научных школах по исследованию вирусного кода и изучению методов маскирования программных модулей используется классический подход — спецификация базовых сервисов ОС, маркерные сигнатуры, динамический анализ исполняемого кода на уровне KOS (Kernel Object Specification). Однако вышеописанные методики не позволяют получать информацию о всех скрытых системных процессах-потоках, поскольку не проводится факторинг «модифицированного микроядра» ОС Windows NT и ОС класса Unix с микроядерной архитектурой. В данной статье рассматривается использование нового фундаментального направления — KPROCESSOR-CID-table-факторинг — при создании операционных систем нового поколения в защищенном исполнении. Рассмотрим кратко основные положения.

Описание применения направления KPROCESSOR_CID-table-факторинг

Ранее мы уже использовали предложенный автором прием изучения динамики дискретной по времени и уровням системы посредством рассмотрения конечной таблицы, дискретного точечного отображения пространства параметров в автоструктуры фазового пространства ДДС с разным качественным поведением [5]. Предложенные математические модели позволили сводить задачи идентификации и восстановления алгоритмов к идентификации дискретных динамических систем с использованием формальных методов и соответствующего математического аппарата, что может стать основой при разработке средств автоматизированного анализа программ в промышленном масштабе. В настоящей работе приведена таблица динамических дискретных систем, которая не только играет роль разбиения пространства параметров с разным качественным поведением, но, скорее, выступает как таблица эффективного анализа возможного кодирования и распознавания в выбранной предметной области. Проверка адекватности рассмотренной модели исследования объектов проводилась на основе имитационной модели построения защищенных операционных систем нового поколения. Ядро любой операционной системы — это привилегированный компонент системы, управляющий доступом к аппаратным ресурсам, выделяющий и освобождающий память, создающий потоки и управляющий ими, осуществляющий внутрипроцессную синхронизацию потоков и управляющий вводом-выводом. Научные школы Массачусетского технологического института и Института Беркли [3] предложили новые технологии для разработки защищенных операционных систем. Кроме обычных каналов передачи сообщений, процессы будут обмениваться информацией с ядром через верифицированный бинарный интерфейс приложений (application binary interface, ABI), вызывающий статические методы в коде ядра. Этот интерфейс следует дизайну остальной системы и изолирует объектные пространства ядра и процесса. Все параметры ABI — значения, а не указатели, поэтому координация сборщиков мусора ядра и процессов не нужна. ABI обеспечивает неизменность

изоляции состояния: с помощью ABI процесс не может изменить состояние другого процесса. За двумя исключениями вызовы ABI влияют только на состояние вызывающего процесса. Эти два исключения — изменение состояния дочернего процесса до или после его исполнения, но не во время исполнения. Первое — это вызов, создающий дочерний процесс и указывающий код, загружаемый в дочерний процесс до его исполнения. Второе — это вызов, останавливающий дочерний процесс, что освобождает ресурсы после завершения исполнения всех потоков. Данный подход не решает проблему гарантированной безопасности. Изоляция состояния не обеспечивает единоличный контроль процессов над своим состоянием. Ядро экспортирует конструкции синхронизации — мьютексы, события с автоматическим и ручным сбросом — для координации потоков в процессе. Поток манипулирует этими конструкциями через строго типизированные непрозрачные дескрипторы, указывающие на таблицу дескрипторов ядра. Слоты в таблице дескрипторов освобождаются только после завершения процесса, чтобы не дать процессу освободить мьютекс. В данном случае удерживание при десинхронизации событий дескриптора вызовет критические ошибки внутри процесса при выполнении кода в нулевом кольце защиты. Увы, предложенный «каркас решений» не разрешает коллизии на уровне реализации проектов нижнего уровня. Каковы технологические решения крупных коммерческих производителей США, Канады и Австралийской ассоциации? Главное решение в дизайне для операционных систем нового поколения от компаний Sun, IBM, Microsoft [4] — инвариант независимости памяти: программные ссылки между объектными пространствами указывают только на Exchange Heap. В частности, ядро не имеет указателей на объектное пространство процесса, и ни один процесс не имеет указателя на объекты другого процесса. Это гарантирует, что каждый процесс может подвергнуться сборке мусора и быть выгруженным без взаимодействия с другими процессами. Операционная система использует связанные (составные) стеки для уменьшения количества памяти, используемой потоком. Компилятор выполняет статический межпроцедурный анализ для оптимизации размещения тестов переполнения. Процесс может передать любую из конечных точек (или обе) другим процессам по существующим каналам. Процесс, получающий конечную точку, имеет канал к процессу, владеющему другой конечной точкой. Например, если процесс приложения хочет связаться с системной службой, приложение создает две конечные точки и отправляет запрос, содержащий одну из точек, серверу имен системы, который перенаправляет конечную точку службе, тем самым создавая канал между процессом и службой. Этот инвариант владения обеспечивается языком и исполняющей системой и служит трем целям. Первая — предотвратить совместное использование сообщений разными процессами. Вторая — способствовать статическому анализу программы, предотвращая множественные ссылки (pointer aliasing) на сообщение. Третья — обеспечить гибкость реализаций, предоставив семантику передачи сообщений, которая может быть реализована копированием или передачей указателя. Мы предлагаем новую модель для безопасного расширения функциональности системы или приложения. В этой модели расширения не могут обратиться к коду или структурам данных родительской программы. Они являются замкнутыми программами, выполняемыми независимо. Однако такой подход увеличивает сложность создания расширения, поскольку разработчик родительской программы должен определить надлежащий интерфейс, который не полагается на общие структуры данных, а разработчик расширения должен обращаться к этому интерфейсу и, возможно, повторно реализовать функциональные возможности, имеющиеся в родительском приложении. Широко распространенные проблемы, присущие динамической загрузке кода, являются доводом в пользу альтернатив, увеличивающих изоляцию расширений. Механизм подходит как для приложений, так и для системного кода; он не зависит от семантики API, в отличие от таких подходов, как Nooks [4], и обеспечивает простые семантические гарантии, которые могут быть поняты

программистами и использоваться инструментальными средствами. Принципиальные аргументы против модели расширения, предложенной специалистами MIT, исследовательских лабораторий Microsoft, IBM, Sun, группируются вокруг сложности написания кода передачи сообщений. DMA (Direct Memory Access — технология прямого доступа к памяти аппаратных устройств) сейчас по сути своей небезопасен и из-за различий в интерфейсах устройств не может быть инкапсулирован или виртуализован системой. Только аппаратная поддержка сегментированных стеков может снизить сложность компилятора и накладные расходы на этот механизм во время исполнения. В любой операционной системе при сопряжении кода ядра с оборудованием на физическом уровне возникают запрещенные состояния — нулевой кортеж данных, к которым процессор запрещает обращаться даже программам в нулевом кольце защиты. Более того, переключение контекста активных задач в защищенном режиме может выполнять только процессорный модуль, поскольку при теневом копировании данных исполняемого кода программист не может получить прямой доступ к информации. Данные коллизии разрешает новое фундаментальное направление — KPROCESSOR-CID-table-факторинг, которое было подробно описано в статье [5]. В качестве параметров (a1,a2,a3,a4) указываются 6-разрядные кортежи (первые два бита — значения поля arp (поле привилегий доступа), третий бит — бит системности, четвертый бит — признак исполнимого формата кода, пятый бит — бит подчинения, шестой бит — атрибут на право чтения). Переменная X принимает любые булевы значения {0;1}. Вместо классического сигнатурного распознавания кода используются генеративные таблицы на основе авторских математических моделей [2, 3]. Ord — поле, которое инициализирует статус потока ядра. Нами введено данное обозначение маркерного сканирования процессов-потоков ядра исходя из значения английского слова ordinance (подчинение, приказ). Таким образом, поле Ord определяет, будет ли запущенный процесс-поток базовым или порожденным (значение бита подчинения сегментов кода). Чтобы получить полный доступ к защищенной таблице процессов, необходимо загрузить модуль драйвера, получить точку входа и по базовому указателю прочитать значения кортежа {s, Ord}. На уровне объектов микроядра каждый процесс-поток идентифицируется значением кортежа данных {s, Ord}. Если полю Ord присваивается значение 0, то объект микроядра описывает базовый (родительский) поток, если значение 1, то в системе запускаются дочерние (порожденные) потоки. Бит s не просто идентифицирует специальные дескрипторы ОС, но и определяет контекст процессов-потоков операционных систем класса Windows NT и Unix, ОС семейства JX, Cedar, RMox, VINO на уровне процессорного модуля. Если бит s = 0, то процесс получает статус System, маркер описывает системный объект с делегированием максимального уровня полномочий (возможность выполнять привилегированные команды процессора и изменять контекст функциональных вызовов), если бит s = 1, то id (идентификатору процесса) приписывается статус Application. Помимо кодирования адресного разделения колец защиты памяти, в ОС разных классов реализован строго типизированный интерфейс сопряжения с аппаратным ядром процессора и управления Контекстом исполнения бинарного кода с учетом профиля компиляции и сборки — использование маркеров системных объектов. Массив данных кортежа {s, Ord} передается в исполнительный регион процессора загрузчиком бинарного кода и инкапсулируется в виде расширенного кортежа {s , Ord, ContextID_}. Параметр ContextID однозначно определяет локальное пространство исполнительного слоя ядра (ContextID_>=0) или адреса верхнего регистра системной памяти, выделенной для критически важных устройств ОС (ContextID_<0). В «теневом» регистре содержатся «снимки» активных процессов-потоков ядра. Таким образом, просматривая значения идентификаторов Context_ID, можно получить информацию обо всех процессах (потоках), что делает ОС гибкой и «прозрачной» для программистов и вирусных аналитиков.

Заключение

В операционных системах класса Unix, как и в ОС Windows NT, модулем самого низкого уровня спецификации является таблица данных исполнительного региона процессорного модуля (KPROCESSOR-CID-таблица), при исследовании программных кодов выделяются существенные параметры (доминанты), что обеспечивает масштабируемость и кросс-платформенность без привязки к конкретным аппаратным вычислительным платформам. Каждое микропроцессорное устройство (в зависимости от типа архитектуры) реализует свой специфический механизм формата приведения типов, кодирования разрядности, работы с регистрами, спецификациями ассемблерного кода. Компиляторы средств разработки системного ПО ориентированы по профилированию и режиму оптимизации, реализации механизмов защиты при автогенерации кода (предпроцессорной обработки или в динамическом режиме Runtime) на определенный класс платформ. Авторское решение создает шаблон — активационную матрицу — для анализа и разработки различных вычислительных систем с помощью «математической свертки и кодировочных таблиц», что снижает время оценки уровня защищенности систем и оптимизирует реализацию проектов нижнего уровня при разработке операционных систем нового поколения в защищенном исполнении. Авторское научное направление в теории анализа кода является универсальным для различных вычислительных систем. В ходе исследований удалось создать математическую модель, оригинальные подходы и методы для анализа программного кода с целью создания защищенных систем, а также предложить и разработать новое научное направление в области компьютерной безопасности [1, 2, 3, 5]. Результаты проведенных исследований использованы при создании СЗИ ViPNet Office Firewall и программного комплекса Monitor. Эффективность полученных результатов подтверждена документально. Научные материалы исследований опубликованы в Центре защиты информации «Нева» Санкт-Петербургского политехнического университета и ВИМИ Министерства обороны РФ.

СПИСОК ЛИТЕРАТУРЫ:

1. *Моляков А. С.* Новый метод систематического поиска недекларированных возможностей ядра Windows NT 5.1 с введением контроля ContextHooking и PspCidHooking // Вопросы защиты информации. 2008. № 1.

2. *Моляков А. С.* Исследование ядра Windows NT на платформе Intel 3000: систематический поиск недекларированных возможностей. М.: Спутник+, 2007.

3. *Моляков А. С.* Исследование скрытых механизмов управления ОС на различных платформах // Известия Южного федерального университета. Сер. «Технические науки». 2007. № 1.

4. *Viega J., Bloch J. T., Kohno T., McGraw G.* ITS4: A Static Vulnerability Scanner for C and C++ Code, 2000.

5. *Моляков А. С.* KPROCESSOR-CID-TABLE-факторинг — новое направление в теории компьютерного анализа вирусного кода и программных закладок // Проблемы информационной безопасности. Компьютерные системы. 2008. № 4.