Метод виртуального процессора в защите программного обеспечения

П.В. Бойко (pavel_bpv@mail.ru)

Аннотация. В статье рассматривается метод защиты программного обеспечения от изучения с помощью переноса защищаемого кода в виртуальную среду исполнения. Проводится анализ эффективности, а так же недостатков метода. Предлагается вариант реализации, позволяющий снизить себестоимость разработки.

1. Введение

Как известно, идеального способа защиты программного обеспечения (ПО) не существует, в связи с этим, разработчики защитных систем не стремятся лишить потенциального взломщика самой возможности нейтрализации защиты, но стараются максимально усложнить этот процесс.

Защита может решать одну или комплекс из множества задач, таких как защита от копирования, нелегального использования, модификации и др., но какая бы конечная цель ни стояла перед таким продуктом, разработчикам каждого из них прежде всего необходимо решить общую для всех проблему – качественной защиты от изучения. Какие бы ни применялись алгоритмы защиты ПО, их стойкость к обратной инженерии определяет стойкость всей системы защиты в целом.

Сегодня на рынке существует большое количество коммерческих защит, однако многие из них, в т.ч. до сих пор популярные, давно взломаны. Зачастую их подводит именно слабая защищенность от изучения. После анализа взломщиком алгоритмов работы защиты, серийные ключи генерируются, аппаратные – успешно эмулируются. Ситуацию могла бы исправить разработка эффективного метода защиты ПО от изучения, применяя который к алгоритмам других защит, позволила бы качественно поднять их уровень.

2. Методы защиты ПО от изучения

Рассмотрим, какие методы существуют для защиты ПО от изучения:

- запутывание – искусственное усложнение кода, с целью затруднить его читабельность и отладку (перемешивание кода, внедрение ложных процедур, передача лишних параметров в процедуры и т.п.);

- мутация – при каждом запуске создаются таблицы соответствия операций, сами операции заменяются на синонимы;
- компрессия, шифрование – изначально программа упаковывается / шифруется, и производит обратный процесс по мере выполнения;
- симуляция процессоров – создается виртуальный процессор; защищаемая программа компилируется под него, и выполняется на целевой машине с помощью симулятора.

Существуют и другие методы, а так же их комбинации и разновидности, однако, нетрудно заметить, что все они основаны на одной простой идее: избыточности. В самом деле, что такое запутывание, как не избыточное кодирование программы? Лишние переходы, лишние параметры, лишние инструкции – ключевое слово метода «лишние». То же касается любого из перечисленных методов, и, вероятно, было бы естественным объединить все эти методы в одну группу «Методов избыточного кодирования». Чем же так хороша избыточность, ведь интуитивно понятно, что она увеличивает размер программы и снижает скорость ее работы? Дело в том, что во всех этих разновидностях защиты используется понимание «человеческого фактора» - человеку тем сложнее понять логику какого-либо процесса, чем больше ресурсов этот процесс использует. Например, функциональность одной простой инструкции загрузки константы на регистр может быть «размазана» на десятки, а то и сотни инструкций, и проследить связь всех используемых ресурсов (регистров, памяти и др.) в этой последовательности человеку довольно сложно. Метод шифрования с этой точки зрения не является чем-то особенным – так же, как и в других методах, для выполнения простой инструкции (или группы) требуется избыточная последовательность команд – в данном случае это операции расшифровки, плюс операции расшифрованного кода.

Однако то, что автоматически «запутано» или усложнено, может быть так же автоматически приведено в первоначальное состояние – разработчики механизмов запутывания обычно параллельно разрабатывают и «распутыватели», а методы мутации и шифрования и вовсе подразумевают содержание обратного механизма в защищенном коде. Особняком в этой группе методов стоит лишь метод симуляции виртуального процессора, который, во-первых, приводит к высокой и неснижаемой степени запутанности результирующего кода, а, во-вторых (при определенном подходе к реализации), защищенный код не содержит в явном виде методов восстановления оригинального кода. Рассмотрим этот метод подробнее.

3. Виртуальный процессор в защите ПО

Суть метода такова: некоторые функции, модули, или программа целиком, компилируются под некий виртуальный процессор, с неизвестной потенциальному взломщику системой команд и архитектурой. Выполнение обеспечивает встраиваемый в результирующий код симулятор. Таким образом,

задача реинжиниринга защищенных фрагментов сводится к изучению архитектуры симулятора, симулируемого им процессора, созданию дизассемблера для последнего, и, наконец, анализу дизассемблированого кода. Задача эта нетривиальна даже для специалиста, имеющего хорошие знания и опыт в работе с архитектурой целевой машины. Взломщик же не имеет доступа ни к описанию архитектуры виртуального процессора, ни к информации по организации используемого симулятора. Стоимость взлома существенно возрастает.

Почему же, учитывая высокую теоретическую эффективность, данный метод до сих пор не используется повсеместно? Видимо по двум основным причинам. Во-первых, метод имеет особенности, что сужает области его потенциального применения – об этом будет сказано ниже. Во-вторых, и, возможно, это более серьезная причина, сложность (а следовательно и стоимость) реализации метода весьма высока. Если же учесть принципиальную возможность утечки информации о только что созданной системе, которая моментально приведет к ее неэффективности и обесцениванию, становится понятно, почему фирмы-производители защитного ПО не спешат реализовывать этот метод.

Стоит, однако, отметить, что с теми или иными вариациями и ограничениями данный метод все же реализован в таких новейших продуктах как StarForce3, NeoGuard, VMProtect и др. Видимо таких продуктов будет становиться все больше и больше, а существующие будут развиваться, т.к. появляющиеся реализации подтверждают высокую эффективность метода, хоть и имеют пока слабые стороны.

3.1. Реализация метода

Одним из недостатков метода является высокая стоимость его реализации, однако она может быть заметно снижена.

В основе системы защиты, реализующей данный метод, мог бы лежать компилятор с языка высокого уровня. Необходимо машинно-зависимая фаза в любом компиляторе всего одна – кодогенерация, от зависимостей в других фазах, как правило, можно избавиться.

Если же компилятор изначально разрабатывается как мультиплатформенный, в нем, как правило, максимально упрощен процесс перенастройки на другую целевую платформу. Например, это может быть достигнуто автоматической генерацией кодогенератора по специальному описанию целевой машины. В этом случае разработчикам для смены платформы достаточно лишь изменить это описание. Но даже если собственного компилятора нет, можно воспользоваться свободнораспространяемыми с открытым кодом, например, GCC.

А чтобы максимально упростить для пользователя работу с описываемой системой защиты, ее можно снабдить механизмами встраивания в популярные среды разработки, такие как MSVC. В этом случае схема работы такого комплекса могла бы выглядеть так, как изображено на Рис. 1.

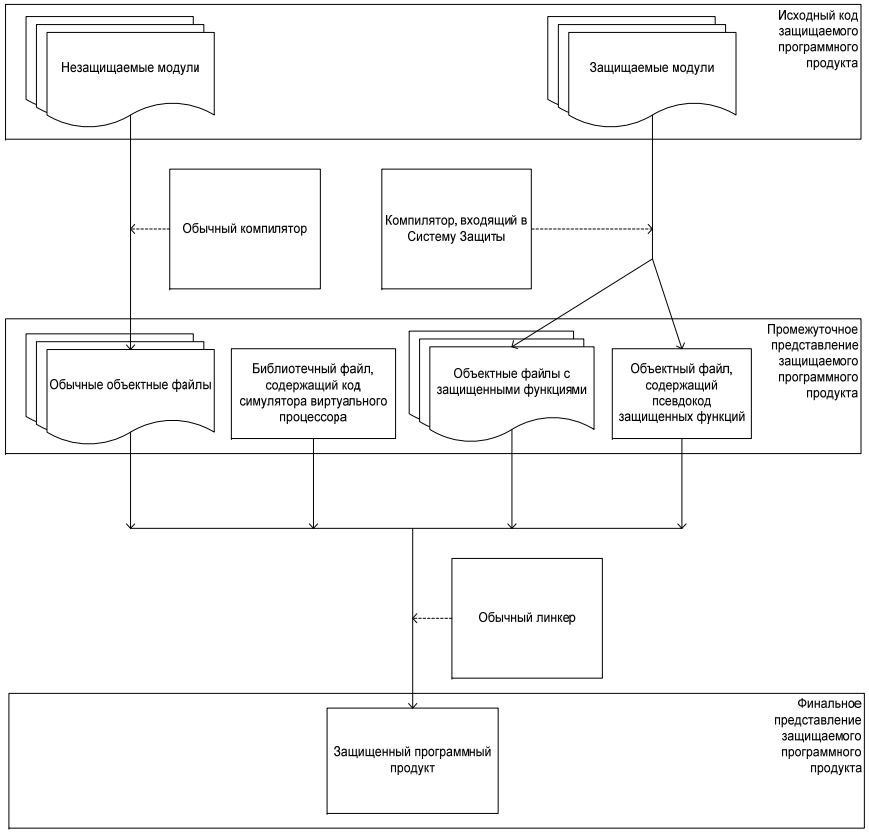

Рис. 1. Схема защиты ПО

Соответственно, функционирование защищенного таким способом образом продукта происходило бы по схеме на Рис. 2.

Специфика использования компилятора налагает ряд особых требований к виртуальному процессору, тем не менее все они могут быть легко реализованы. Требований немного – нужно лишь обеспечить возможность доступа к внешней, относительно виртуальной машины, памяти, а так же возможность вызова внешних функций – это необходимо для взаимодействия защищенного и незащищенного кода. В остальном архитектура виртуального процессора может быть совершенно произвольной, и чем запутаннее и оригинальней она будет, тем более высокий уровень защиты будет достигнут.

Сам компилятор, кроме изменения кодогенерационной фазы, нужно доработать для приобретения им возможностей:

- различать обращения к внутренней и внешней памяти относительно виртуального процессора (в том числе и вызовы функций);
- создавать для каждой защищаемой функции так называемую оболочку, выполняющуюся на реальном процессоре, с вызовом защищенной функции через симулятор.

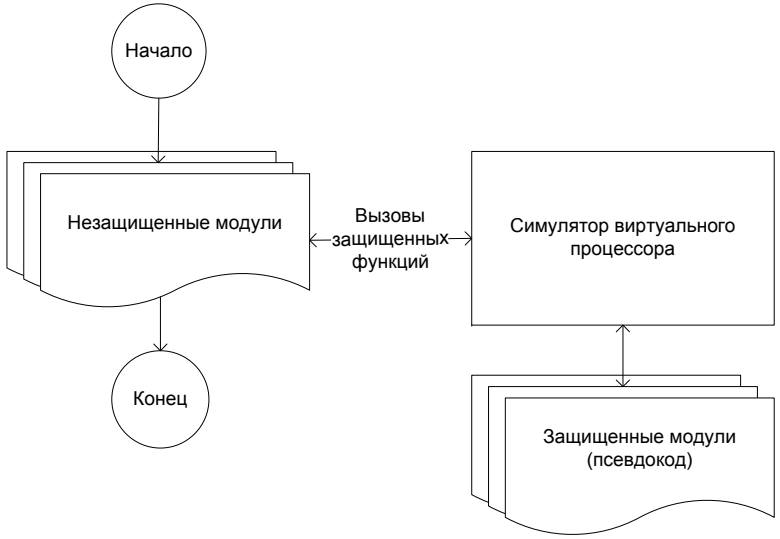

Рис. 2. Схема работы защищенного программного продукта

Последнюю возможность следует описать подробнее. Как было отражено в схеме на Рис. 2, функционал защищенных функций будет реализован через вызов симулятора, с указанием, какую из защищенных функций нужно интерпретировать. Однако, до этого, необходимо выполнить специальный код, подготавливающий для защищенной функции параметры - "переместить" их с реальных регистров и памяти на виртуальные, способом, соответствующим архитектуре виртуального процессора. Всем этим будут заниматься специальные функции, сгенерированные нашим компилятором - "оболочки". Оболочки, в свою очередь, будут использовать специальные функции симулятора для доступа к виртуальным регистрам и памяти. Характерно, что наш компилятор будет генерировать оболочки на языке высокого уровня, которые, в свою очередь, будут компилироваться стандартным компилятором, использующимся пользователем для сборки незащищенной части своего проекта. Итак, вызов защищенной функции из незащищенного модуля может выглядеть так, как изображено на Рис. 3.

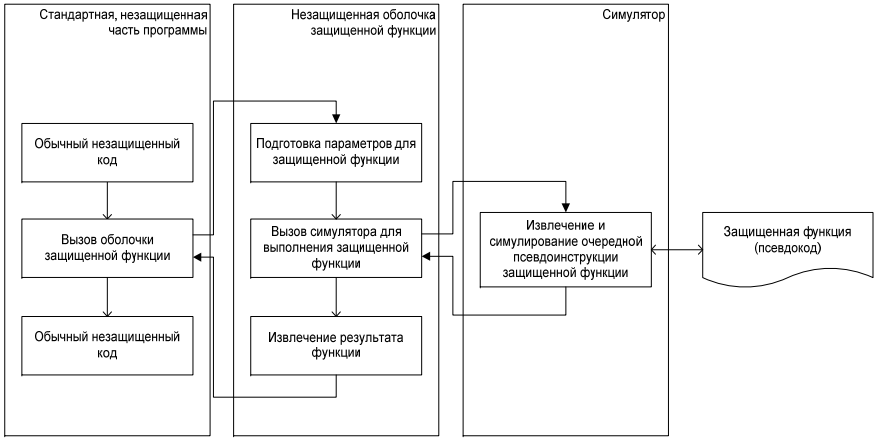

Рис. 3. Схема вызова защищенной функции

Конечно, конкретная реализация метода виртуального процессора может быть несколько отличной от описываемой, как и схема работы защищенного им продукта. Тем не менее, описанный вариант вполне жизнеспособен, и, кроме того, относительно прост.

Сердце защищенного продукта – симулятор. Он будет включаться в любую сборку защищенного продукта. Однако не будем подробно рассматривать его реализацию, т.к. специальных требований к нему практически не предъявляется – он должен лишь симулировать архитектуру нашего виртуального процессора, включая операции доступа к внешней памяти. Стоит, однако, отметить, что с учетом специфики его применения, необходимо максимально автоматизировать процесс перенастройки симулятора на новые виртуальные архитектуры.

Недостатки же метода – следствие его достоинств:

- скорость работы перенесенного в виртуальную среду кода в разы (ориентировочно в 10-50, в зависимости от архитектуры виртуального процессора и симулятора) ниже, чем кода оригинального;
- объем защищенной программы, как правило, будет несколько выше, чем незащищенной.

Впрочем, последний недостаток несущественен, т.к. размер увеличится незначительно, а в некоторых случаях может даже снижаться. Первый же недостаток принципиален, и налагает некоторые очевидные ограничения на использование метода.

4. Заключение

Рассмотренный метод защиты ПО весьма эффективен, учитывая то, что затраты на его разработку можно существенно сократить. Однако, особенности метода не позволяют рекомендовать его для защиты программ полностью. Так, метод не может применяться для защиты функций, критичных ко времени выполнения, а так же функций, замедление работы которых может заметно снизить эффективность использования программы пользователем. Тем не менее, аккуратное применение данного метода позволяет добиться очень высокого уровня защиты от изучения. В связи с этим, основной областью его применения видится повышение стойкости к изучению отдельных алгоритмов других систем защиты ПО. Кроме того, метод применим для защиты нересурсоемких алгоритмов ноу-хау, а так же для сокрытия содержания в защищаемой программе некоторых специальных данных, например, сведений об авторстве.