Федеральное агентство по образованию

Санкт-Петербургский государственный политехнический университет

—

Факультет технической кибернетики

Кафедра «Измерительных информационных технологий»

О Т Ч Ё Т

по научно-исследовательской работе

на тему «Методы защиты программного обеспечения,

исполняемого в среде .Net Framework»

Выполнил

студент гр.6085/12 А.Л. Копнин

Руководитель

доцент, к.т.н. В.Ю. Сальников

Санкт-Петербург

2011

Содержание:

1 Введение.

Информационные технологии являются одной из наиболее быстро развивающихся областей современной жизни. Растущая вычислительная мощность компьютеров и дешевеющие комплектующие, все это позволяет создавать более крупные и ресурсоёмкие программные продукты. Современные разработчики могут использовать сложные и многофункциональные программные пакеты для разработки программного обеспечения. Это позволяют абстрагироваться от низкоуровневых проблем и уделить больше времени на детальную проработку алгоритма. Так же теперь задача разработки кроссплатформенных программ стоит перед программистом в меньшей степени, чем раньше. Языки программирования эволюционировали в целый комплекс взаимосвязанных компонентов, которые включают в себе компиляцию в промежуточный язык – байт код, и виртуальную машину, необходимую для выполнения полученного байт кода независимо от целевой платформы.

Одним из первых подобных программных комплексов был язык Java. С его появлением перед разработчиками встала проблема обеспечения защиты своих алгоритмов [1]. Приложения, написанные с использованием платформы Java или .Net, компилируются в промежуточный язык, который в дальнейшем исполняется на Java-машине или в среде .Net Framework соответственно. Такая схема разработки обладает положительными и отрицательными сторонами. Упрощение процесса написания кода, обеспечение кроссплатформенности, за счет промежуточного байт-кода – приводят к проблемам безопасности интеллектуальной собственности создаваемого программного продукта. Программы, написанные на языке C#, с легкостью дизассемблируются с точным восстановлением исходного кода. Сторонние программисты имеют возможность детально ознакомиться с восстановленным кодом,

модифицировать его, пересобрать, или использовать заинтересовавшие фрагменты в своей реализации.

Каждый разработчик, который создает коммерческий продукт, предусматривает способ лицензирования каждой копии своего приложения. Для того чтобы заказчик мог легально использовать очередную копию программного обеспечения, он должен иметь уникальную лицензию, выдаваемую разработчиком. В силу возможности восстановления исходного кода процедура обхода лицензирования нелегальными средствами многократно возрастает.

Многие эксперты в области компьютерной безопасности активно занимаются разрешением проблемы интеллектуальной собственности. Появилось понятие "обфускация" программного кода - приведение программы к виду, затрудняющему восприятие и модификацию. В последнее время интерес к задаче обфускации резко возрос. Повышенный интерес к проблеме обфускации в значительной степени обусловлен появлением технологии .Net от Microsoft. На сегодняшний день существует несколько десятков программных комплексов, которые позволяют в той или иной степени решить проблемы защищенности кода.

Николай Владимирович Лихачёв – специалист в области информационной безопасности, работающий в подразделении компании McAfee, в своей статье [2] “Обфускация и ее преодоление” дал классификацию существующих типов обфускаторов: “ Существуют различные типы обфускаторов: одни занимаются интерпретируемыми языками, например, Perl или PHP, и модифицируют исходные тексты, удаляют комментарии, дают переменным бессмысленные имена, шифруют строковые константы. Другие преобразуют байт-код виртуальных машин Java и .Net, что технически сделать намного труднее. Самые совершенные обфускаторы внедряются непосредственно в машинный код, "разбавляя" его мусорными инструкциями и выполняя целый ряд структурных или математических преобразований, изменяющих программу до неузнаваемости.”

Нарушение интересов обладателей авторских прав широко распространено во многих странах, в том числе в России, на Украине, в Китае, Казахстане, Бразилии, Мексике и Индонезии. По анализам независимых источников [3], основная причина широкого распространения такого нарушения в этих странах носит экономический характер:

1. Цены на произведения интеллектуальной собственности, распространяемые правообладателями-монополистами, могут быть неоправданно завышенными по сравнению с материальными возможностями или ожиданиями пользователей произведений.
2. Официальные цены на произведения интеллектуальной собственности, возможно, являются нормальными (средними, общепринятыми) в стране, где разработан продукт, но могут быть чрезмерными для некоторых других стран.

Помимо этого, несоблюдение авторских прав может способствовать обеспечению доступа широких слоёв населения к информации, распространение которой по той или иной причине ограничено правообладателями, например, автор программного обеспечения официально не выпускает его в той или иной стране.

На современном рынке информационной безопасности известно достаточно большое количество приложений, предназначенных для обеспечения защиты интеллектуальной собственности, среди которых .Net Reflector, {SmartAssembly}, DotFuscator, dotNetProtector, Salamander.Net и другие. Но обеспечить должный уровень безопасности могут только единицы.

Эта работа направлена на исследование существующих решений, выявление их уязвимых мест, и, исходя из полученных результатов, реализацию методовв, сочетающих в себе максимально возможный уровень безопасности.

2 Обзор современных средств защиты программного обеспечения, исполняемого в среде .Net Framework.

Задача этой главы рассмотреть особенности архитектуры .Net Framework, выявить ее слабые и сильные стороны. Проследить историю развития платформы, и определить существующие средства защиты исходного кода, написанного на языке C#.

2.1 История развития платформы .Net Framework.

.Net Framework – программная платформа, выпущенная компанией Microsoft в 2002 году. Основой платформы является исполняющая среда Common Language Runtime (CLR), способная выполнять как обычные программы, так и серверные веб-приложения. Платформа поддерживает создание программ, написанных на разных языках программирования.

Первый релиз .NET Framework 1.0 вышел 5 января 2002 года для Windows 98, NT 4.0, 2000 и XP. На данный момент она устарела, и ее поддержка прекратилась в 2007 году.

Следующий релиз .NET Framework 1.1 вышел 1 апреля 2003 года. Это первая версия, автоматически устанавливаемая вместе с операционной системой [4]. Для более старых операционных систем платформа доступна в виде отдельного установочного пакета. Безопасность в среде .Net Framework 1.1 подразумевает контроль над доступом к таким ресурсам, как компоненты приложения, данные и устройства компьютера. Microsoft .NET Framework предоставляет управление доступом для кода и безопасность на основе ролей, которые помогают решать проблемы безопасности, связанные с мобильным кодом, и позволяют компонентам определять, какие действия разрешены определенным пользователям. Эти механизмы безопасности построены на основе простой, согласованной модели, что дает возможность разработчикам, знакомым с принципами управления доступом для кода, легко использовать безопасность на основе ролей и наоборот. Управление доступом для кода и безопасность на основе ролей реализованы с использованием общей инфраструктуры, предоставляемой средой CLR.

В версии .NET Framework 2.0 добавлена поддержка анонимных методов и полная поддержка 64-битных платформ. Дальнейшее развитие платформ не

подразумевало серьезных изменений в системе безопасности, вплоть до выхода .Net Framework 4.0. Очередные версии платформы имели значительное расширение наборов классов и инфраструктур.

2.2 Современная модель безопасности .Net Framework.

Модель безопасности, которая развивалась в .Net Framework, начиная с версии 1.1, называется Code Access Security. Это технология обеспечения безопасности, разработанная, чтобы предоставить возможность защиты системных ресурсов при исполнении сборок .NET [4]. Такими системными ресурсами могут быть: локально расположенные файлы, файлы на удаленных системах, ключи реестра, базы данных, принтеры и так далее. Неограниченный доступ к этим типам ресурсам влечет за собой потенциальный риск, так как вредоносный код может выполнить разрушительные операции, например, удалить критически важные файлы, изменить ключи реестра или удалить данные, хранящиеся в БД.

Code Access Security позволяет разработчикам и системным администраторам защищать ресурсы, определяя:

1. Право, которое должны иметь метод или сборка, чтобы обращаться к критическим ресурсам.

2. Набор прав, то есть коллекций из двух или более прав.

3. Свойство сборки, которое работает как своего рода комбинация идентификатора сборки относительно зоны, из которой она пришла (данная машина, интранет и т.д.). Идентификатор издателя сборки, полученный с использованием цифрового сертификата, которым подписана сборка. Идентификатор самой сборки, представленный именем или значением его хэш, а также местоположением сборки.

При загрузке сборки платформа от Microsoft проверяет ее свойства и назначает ей только те разрешения, которые допустимы для набора прав.

Изменения в .Net Framework 4.0 во многом являются реакцией на имеющиеся проблемы, а не реализацией новых возможностей. С годами в модели безопасности, реализованной в предыдущих версиях .Net Framework,

проявились проблемы, которые не так просто решить. Маттео Славиеро в своем исследовании [5] обозначил основные проблемы платформы:

1. При перемещении отдельного приложения на другую систему настройки безопасности новой системы, отличающиеся от использовавшихся ранее, приводят к неверному функционированию самого приложения. Например, если исполняемый файл хорошо работает на машине разработчика, при переносе на рабочий сервер тот же самый исполняемый файл может неожиданно перестать работать.

2. При написании кода не так просто настроить Code Access Security для сборки. Это происходит потому, что администратору требуется создавать на своих машинах совершенно разные политики Code Access Security, а разработчику – учитывать все возможные варианты правил. При этом разработчики никак не могут заранее знать, какие решения будет принимать администратор.

Политики Code Access Security очень полезны, когда администратору нужно контролировать, что может, а что не может делать программное обеспечение [6], но на неуправляемый код политики Code Access Security не действуют никак.

В .NET Framework 4.0 произошли существенные изменения в области безопасности. Политика безопасности для всего компьютера больше не применяется, хотя система разрешений продолжает функционировать.

Появилась Level2 Security Transparent-модель [5]. Она делит весь код на три категории: критическая безопасность (Security Critical), прозрачная безопасность (Security Transparent) и защищенная критическая безопасность (Security Safe Critical).

1. Код из категории критическая безопасность пользуется полным доверием. Такой код может вызываться другим кодом этой категории или кодом из защищенной критической безопасности, но не может вызываться кодом категории прозрачная безопасность.

2. Код из категории прозрачной безопасности имеет ограниченные привилегии доступа к системным ресурсам и не имеет доступа к коду критической безопасности. Кроме того, он не может вызывать native-код и расширять привилегии.

3. Код из категории защищенная критическая безопасность – это своего рода мост между кодом прозрачной безопасности и кодом критической безопасности (рис. 2.1). На самом деле прозрачная безопасность может вызывать код защищенной критической безопасности, который, в свою очередь, может вызывать критическую безопасность. Код этой категории рассматривается как полностью доверенный и имеет те же полномочия, что и код критической безопасности.

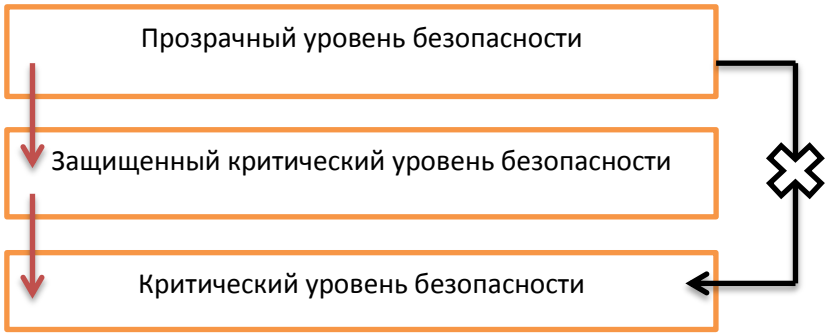

Рис. 2.1. Модель безопасности .Net Framework 4.0

Если сборка создана сторонним разработчиком, то выставить атрибуты безопасности можно, помещая сборку в sandbox, ограничивающую использование ресурсов сборкой, что позволит защитить систему. Level2 Security Transparence пришла на смену Code Access Security, оставив хосту возможность создавать разрешения.

Таким образом, в среде .Net Framework предлагается защита использования критичных ресурсов системы, и безопасность использования сборок, написанных сторонними разработчиками.

2.3 Верификация кода.

При разработке платформы .Net Framework было уделено много внимания обеспечению безопасности выполняемого программного кода [7]. С точки зрения безопасности можно привести следующую классификацию:

1. Недопустимый код.

   Это код, который не может быть обработан JIT-компилятором, то есть не может быть транслирован в машинный код.

2. Допустимый код.

   Это код, который может быть представлен в виде машинного кода. При этом он может содержать вредоносные фрагменты или ошибки, способные нарушить работу не только программы, но и среды выполнения и даже операционной системы.

3. Безопасный код.

   Безопасный код не содержит вредоносных фрагментов и не может повредить ни системе выполнения, ни операционной системе, ни другим выполняемым продуктам.

4. Верифицируемый код.

   Верифицируемый код – это код, безопасность которого может быть строго доказана алгоритмом верификации, встроенным в CLR.

Весь код, который поступает в JIT-компилятор, автоматически подвергается верификации. Верификатор платформы .Net реализует достаточно простой линейный алгоритм проверки правильной работы программного кода. В зависимости от настроек безопасности .Net Framework система выполнения может разрешить или не разрешить выполняться на машине код, отбракованный верификатором.

Таким образов, верификация в платформе .Net Framework, может гарантировать, в большинстве случаев, что код, переданный ей на выполнение, содержит критические, с точки зрения безопасности, фрагменты.

2.4 Обзор архитектуры .Net Framework.

Как утверждает Макаров в своей работе [8], посвященной платформа .Net Framework , она состоит из двух основных компонентов. Это Common Language Runtime и .Net Framework Class Library.

Common Language Runtime можно назвать “двигателем” платформы. Его задача – обеспечить выполнение приложений .Net, которые, как правило, закодированы на языке CIL, рассчитаны на автоматическое управление памятью и вообще требуют гораздо больше заботы, чем обычные приложения Windows. Поэтому CLR занимается управлением памятью, компиляцией и выполнением кода, работая с потоками управления, обеспечением безопасности и прочее.

.Net Framework Class Library – это набор классов на все случаи жизни. На платформе .Net реализованы компиляторы для различных языков программирования, и большинство этих языков позволяют легко использовать одну и ту же библиотеку классов. То есть .Net Framework Class Library – это единая библиотека для всех языков платформы .Net. Использование этой библиотеки позволяет существенно сократить размер библиотеки, что способствует их распространению через Internet.

Программы для платформы .Net Framework распространяются в виде сборок [9]. Каждая сборка представляет собой совокупность метаданных, описывающих типы, и CIL-кода.

Ключевой особенностью выполнения программ в среде .Net является Just In Time-компиляция. Компиляция заключается в том, что CIL-код, находящийся в запускаемой сборке, тут же компилируется в машинный код, на который затем передается управление.

В .Net реализованы два JIT-компилятора: один компилирует сборку непосредственно перед ее выполнением, а другой позволяет откомпилировать ее заранее и поместить в так называемый кэш откомпилированных сборок.

2.5 Защита исходного кода.

Из скомпилированных сборок для платформы .Net Framework может быть легко восстановлен код на языках высокого уровня, в частности C#, VB.net. Это означает, если программный продукт подразумевает систему лицензирования, то этот участок кода может быть легко идентифицирован и модифицирован. Так же злоумышленник, для которого программа представляет коммерческую выгоду, например конкурент, может воспользоваться восстановленным исходным кодом, чтобы перенести его фрагменты в свою реализацию.

Существует несколько классов программ, которые предоставляют защиту исходного кода .Net приложений.

1. Упаковщики.

   Это класс программного обеспечения, который изначально предполагал сокращение размера исполняемых модулей. В дальнейшем, этот класс программ стал обладать функцией защиты, так как после упаковки нельзя непосредственно редактировать файлы: для этого нужно распаковать файл, отредактировать его должным образом, а затем снова упаковать.

   Данный механизм реализует защиту приложения путем модификации сборки таким образом, что на выходе получается WinApi-приложение, которое содержит в себе распаковывающийся модуль и упакованный IL-код.

   Такой способ защиты несет ряд сложностей:

   - Упакованный результат выполняется только под операционной системой Windows.
   - На вход виртуальной машины подается оригинальный код, который содержится в упакованной программе. Следовательно, он без особых усилий перехватывается, и злоумышленник получает исходный код.
   - Упаковщики могут перехватывать вызовы функций из библиотеки виртуальной машины mscoree.dll. В таком случае происходит сильная

привязка к версии .Net Framework, а так же многие антивирусы могут расценивать такое поведение, как попытку атаки.

2. Обфускаторы (Obfuscator).

Обфуска́ция или запутывание кода — приведение исходного текста или исполняемого кода программы к виду, сохраняющему ее функциональность, но затрудняющему анализ [11], понимание алгоритмов работы и модификацию при декомпиляции.

Суть процесса обфускации заключается в том, чтобы запутать программный код и устранить большинство логических связей в нем, то есть трансформировать его так, чтобы он был очень труден для изучения и модификации посторонними лицами.

На вход такого рода программ подается оригинальная .Net-сборка, а результатом является валидная .Net-сборка, которая претерпела ряд модификаций с точки зрения IL кода.

3. Защитники (Protectors).

Этот класс программ помимо функции упаковки, модифицирует исполняемые файлы таким образом, что распаковать их после этого становится намного сложнее, чем при использовании любого другого упаковщика.

Программы защитники, упаковывают, шифруют код, портят таблицу импортов в файле [10]. Некоторые используют защиту против отладки. Могут проверять контрольную сумму, хэш сумму запакованного файла, чтобы предотвратить возможность редактирования файла.

Некоторые программы обеспечивают связь всех уровней защиты, что позволяет своевременно реагировать на любые изменения. Например, при определении, что программу отлаживают, то, не предпринимая никаких видимых действий, программа защитник передает информацию на нижний уровень, где возможна более гибкая обработка действий.

Защитники объединяют в себе два описанных ранее подхода: упаковщики и обфускаторы.

2.6 Базовые методики работы средств, предназначенных для защиты исходного кода.

Основной целью общих методов обфускации кода является запутывание кода таким образом, что было бы сложно понять, как данный код работает.

1. Объединение сборок и пространства имён.

   Методика объединения подразумевает избавление от многомодульного способа программирования. Сама по себе она не задерживает злоумышленника ни на минуту, но механизм полезен для дальнейшего запутывания.

2. Переименование классов, методов и т.д.

   Каждый обфускатор обладает механизмом переименования классов и методов, равно как и деобфускаторы умеют обходить это способ защиты без особых усилий.

   Программа обфускатор удаляет всё, что злоумышленник может использовать для быстрого поиска классов, отвечающих за лицензирование [12]. Этот способ нацелен на то, чтобы осложнить злоумышленнику, при прямом просмотре исходного кода, понимание логики работы приложения. Скрыть для чего создаются классы, вызываются те или иные методы.

   Основные способы реализации – это переименование в непечатные символы, использование коротких, но печатных идентификаторов, использование ключевых слов языка высокого уровня, либо не валидных идентификаторов этого языка. Такие преобразования затрудняют просмотр через Reflector, но на деобфускаторе это никак не сказывается.

   Еще один механизм этой методики – это создание большого количества перегруженных методов, которые до обфускации никак не были связаны.

   Эта разновидность модификации кода может привести к неприятным последствиям, например, если у пользователя конечным продуктом выпадет исключение, то направленный разработчику отчет может быть полностью не информативен.

3. Изменение содержимого классов.

   Некоторые обфускаторы могут объединять разные классы в один, или делать вложенные классы [12]. Чаще всего это ведет к ошибкам и используется редко.

4. Преобразование вычислений.

   Основываясь на работе Коллбегра [13] можно заключить, что существует строгая зависимость между сложностью кода и числом конструкций, которое содержится в коде. С увеличением числа конструкций, добавить дополнительный запутывающий код становится проще. Например, базовый блок кода, B можно разделить на две части путем вставки некоторого запутывающего условия P(t) , которое всегда возвращает true, в середину блока B [14]. После этого для каждого из получившихся блоков можно применять различную технику обфускации.

5. Обфускация абстрактных данных.

   - Модификация связей наследования.

     Метрика Чидамбера [13] определяет сложность класса, основываясь на количестве его методов, глубине дерева наследования, количества прямых производных классов, и прочих классов, с которыми он связан.

     В соответствии с этой метрикой сложность программы увеличивается с увеличением глубины дерева наследования. Таким образом, сложность можно повысить путем разделения класса на несколько частей или путем дополнения фиктивных классов.

   - Реструктуризация массивов.

     В отличие от сокрытия массивов в классических языках программирования, где цель разработчика – это спрятать адрес массива в ассемблерном коде, в языках, компилируемых в байт код, основная задача – это скрыть операции с массивами, или затруднить их понимание.

Среди различных способов сокрытия операций выполняемых над массивами необходимо отметить разбивку массивов на несколько частей, склеивание нескольких массивов, уменьшение или увеличение размерности, путем привидения массивов к плоскому или многомерному виду.

6. Обфускация процедурных абстракций.

Представленная классификация основывается на работе Коллбегра [14], и ее обобщении Калпинским [13].

- Табличная интерпретация.

  Этот механизм представляет один из самых эффективных механизмом обеспечения защиты. Основная идея преобразования заключается в конвертации части кода в другой машинный код. Участок кода должен выполняться новым интерпретатором виртуальной машины, включенным в обфусцированный код. Основным недостатком этого способа является существенное замедление времени исполнения.

- Клонирующие методы.

  При исследовании программного обеспечения, прежде всего, изучается тело метода и ее сигнатура. Таким образом, для усложнения восприятия, можно сделать код более запутанным, заменив вызовы реальных методов, другими методами.

7. Обфускация встроенных типов данных.

- Разделение переменных.

  Идея метода заключается в представлении некоторой переменной X типа T в переменные a и b типа U.

  Для возможности такого преобразования, необходимо обеспечить следующее [13]:

  1) F(a, b), которая отображает a и b на соответствующее значение типа U.
  2) F(X), которая отображает X на a и b.

Чем больше деление переменной тем эффективность метода выше.

- Конвертация статических и процедурных данных.

  Статические переменные – это очень важная информация для злоумышленника. Чтобы усложнить анализ константных данных разработчик может использовать динамическое вычисление. Например, использовать ряд функций, которые вычисляют константные данные.

2.7 Обфускация, как средство против статического анализа программ.

В этом разделе рассматривается модель системы, которая представлена на рис 2.2. Определим критерии рассматриваемой модели:

1. Программа отправляется от доверенного сервера.
2. Сетевое взаимодействие считается безопасным. Предполагается использование криптографических протоколов для обеспечения сетевого взаимодействия. Сетевые атаки, как внедрение, подмена и другие считаются невозможными.
3. Исследуется защита против сложно контролируемых атак, которые попадают в категории интеллектуальной собственности.

Атака интеллектуальной собственности подразумевает, что злоумышленник изменяет программный продукт в направлении, которое позволяет программе выполняться корректно, но с неправильными данными или находиться в неверном состоянии.

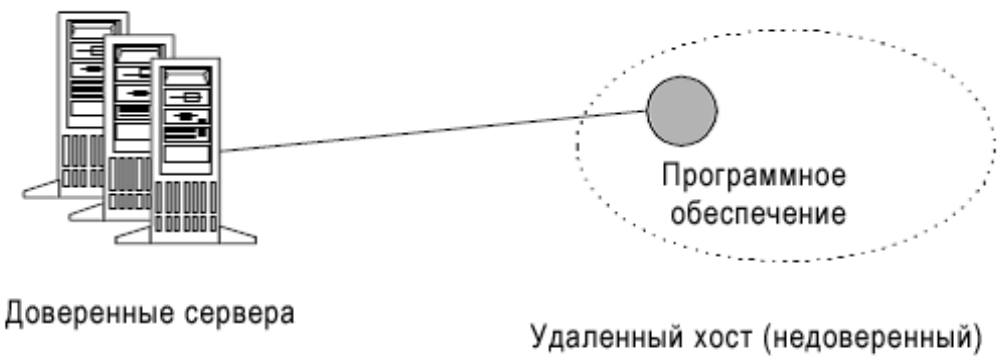

Рис. 2.2. Схема распространения и выполнения ПО.

Для успешного проведения атаки злоумышленник должно повредить механизм проверки легальности копии. Для этого требуется информация о семантике программы, и поведении механизма проверки.

Статический анализ основывается на технических методах извлечения информации из статического образа программ. Традиционно, статический анализ используется для начального сбора информации.

Статический анализ предполагает анализ потока управления (control-flow) и потока данных (data-flow). Анализ потока управления включает следующие шаги:

1. Анализ потока данных для построения графа потока программы. Граф состоит из узлов, которые представляют блоки и ребра, описывающие передачу управления между блоками.
2. Определение проблемы сбора информации, как проблему потока данных и провести анализ потока данных на графе потока

Анализ потока управления – это первая стадия исследования. Без этой информации анализ потока данных будет ограничен только базовым блоком и будет фундаментально не эффективен. Созависимость control-flow и data-flow анализов дает значительное повышение сложности проведения обоих анализов, а также приводит к уменьшению точности полученных данных.

2.8 Преобразование потока управления.

Построение графа потока – это набор простых операций, когда инструкции и их назначение легко идентифицируемы. Операция построения графа потока является операция линейной сложности Q(n), где n – количество блоков в программе. Тенденция современных программ – это наличие потока управления, который может быть легко различимым.

Для повышения сложности построения графа потока управления, необходимо изменить высокоуровневое управление. Это преобразование можно реализовать в несколько проходов. На первом этапе высокоуровневое управление структурируется и переписывается через if-then-go конструкции (рис. 2.3).

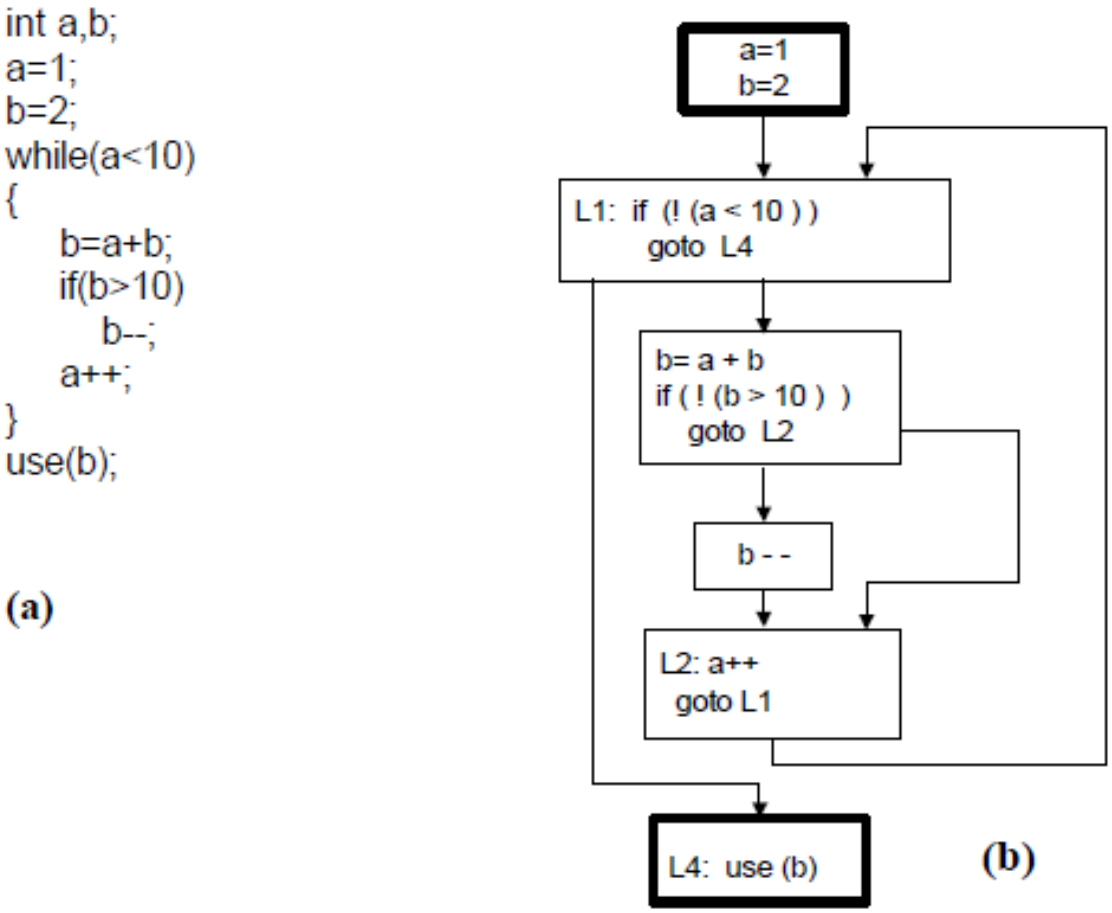

Рис. 2.3 Преобразование логики вызова.

При следующем проходе модифицируется статический адрес на динамически вычисляемый. Примером может сложить перезапись goto на switch, где значение switch переменной вычисляется на в каждом блоке кода, тем самым определяя какой блок будет выполняться следующим (рис. 2.4).

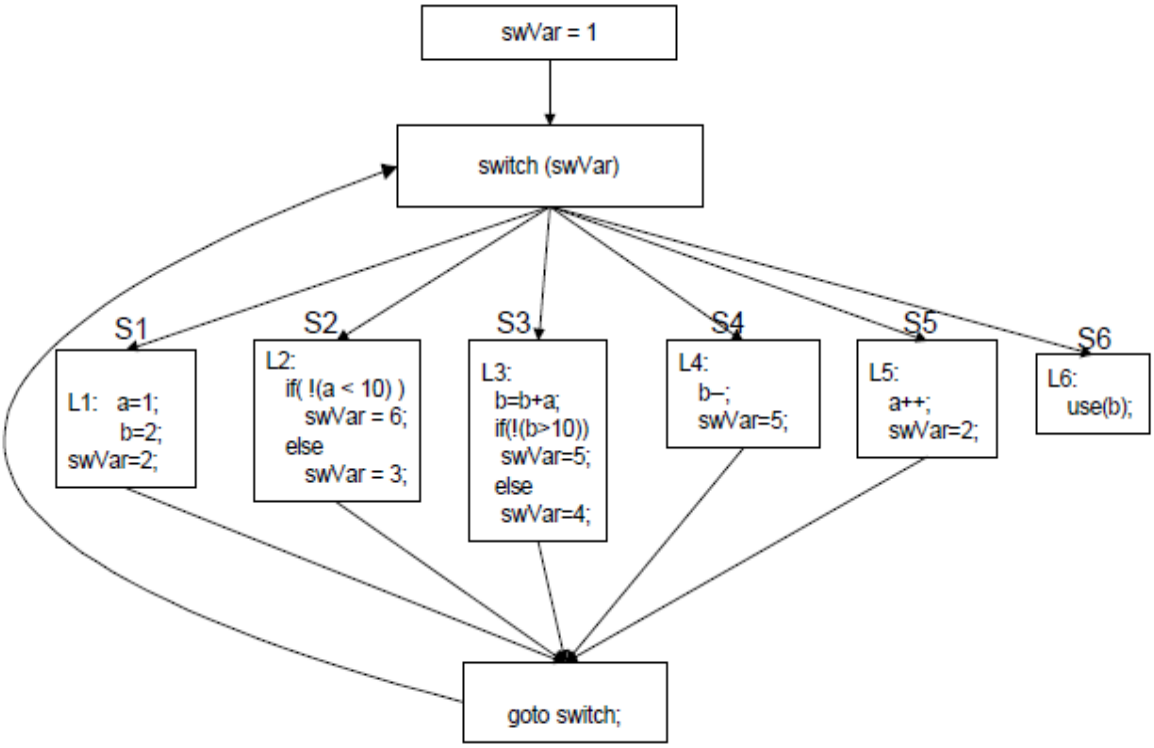

Рис. 2.4 Преобразование потока управления.

Такое преобразование внедряет в поток управления зависимость от данных. Сложность статического построения графа потока зависит от сложности вычисления switch переменной. Таким образом, построение графа управления зависит от потока данных, что представляет дополнительную сложность построения графа потока, а следовательно и усложнение статического анализа.

3 Защита программного обеспечения, исполняемого в среде .Net Framework.

Глава содержит анализ методических рекомендаций, которые автор исследовал, разработал в рамках магистерской диссертации. Набор ниже приведенных правил, позволяет разработчику защитить программный продукт от нелегального использования. Использование описанных методик ведет к усложнению процесса обратной инженерии программного обеспечения, что обеспечивает более высокий уровень безопасности кода.

3.1 Направления атак киберпреступников.

Как было описано в предыдущих главах диссертации, основная цель злоумышленника – это полное или частичное изменение фрагмента кода, отвечающего за лицензирование программного продукта. В случае использования виртуальной машины .Net Framework, векторы нападения киберпреступника можно условно разделить на три направления:

1) Восстановления исходного кода программного продукта по промежуточному представлению, в котором распространятся сборка, написанная на .Net Framework.
2) Анализ, статическая отладка и модификация промежуточного представления программного продукта.
3) Анализ, динамическая отладка и модификация ассемблерного представления программы.

Три направления действий злоумышленника являются ключевыми в вопросах безопасности. Исходя из этого, можно заключить, что для увеличения сложности взлома, необходимо обеспечить защиту этих направлений.

3.2 Архитектура .Net Framework. Компиляция.

Программа, написанная на платформе .Net Framework компилируется в промежуточное представление, которое есть низкоуровневый язык виртуальной машины MSIL (CIL, IL). Его можно рассматривать, как ассемблер некоторой виртуальной машины.

Это нетипичный ассемблер, так как он обладает многими конструкциями, характерными для языков более высокого уровня: например, в нем есть инструкции для описания пространств имен, классов, вызовов методов, свойств, событий и исключительных ситуаций. Кроме того, MSIL является стековой машиной со статической проверкой типов; это позволяет отслеживать некоторые типичные ошибки. MSIL представляет собой дополнительный уровень абстракции, позволяющий легко справляться с переносом кода с одной платформы на другую, в том числе, и с изменением разрядности платформы.

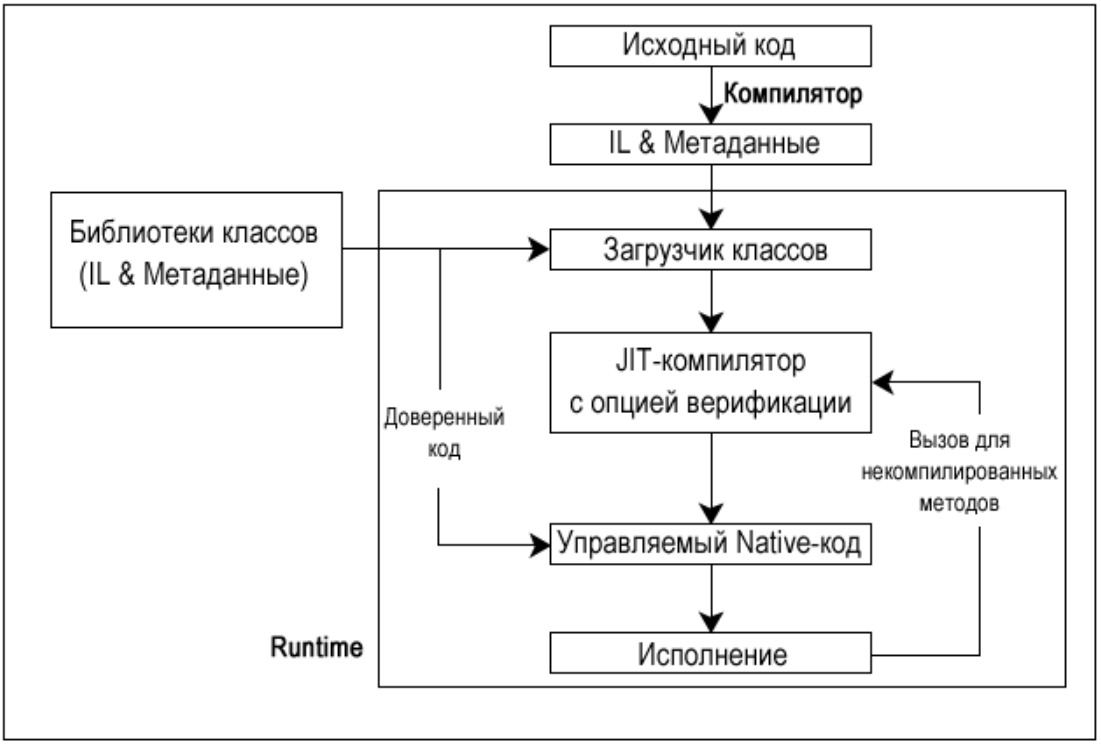

Рис 3.1. Схема архитектуры .Net Framework.

Исходный код компилируется в промежуточный байт-код и только перед выполнением преобразуется, с помощью JIT-компилятора, в родной код платформы, на которой он исполняется (рис. 3.1). MSIL сохраняет достаточно

много информации об именах, использованных в исходной программе: имена классов, методов и исключительных ситуаций. Они могут быть извлечены при обратном ассемблировании.

Ниже приведен пример кода, который написан на языке MC++ и его представление в байт-коде.

MC++ представление:

```
void main( int p_Password )
{
        TopSecurity Obj;
        int SecLevel = Obj.GetSecurityLevel( p_Password );
}
```

Представление в байт-коде MSIL (некоторый код опущен):

```
 .locals ( [0] int32 SecLevel,
           [1] valuetype TopSecurity Obj )

 IL_0000:  ldloca.s   Obj
 IL_0002:  call  valuetype TopSecurity*… 'TopSecurity.{ctor}'…
 IL_0008:  ldloca.s   Obj
 IL_000a:  ldarg.0
 IL_000b:  call  int32 … TopSecurity.GetSecurityLevel(valuetype TopSecurity*…
```

Объект TopSecurity принимает секретные данные, затем проверяет их корректность и если они валидны, то возвращает уровень секретности, в противном случае -1 . На этом примере, видно, что в байт-коде сохраняются исходные названия переменных и легко прослеживается последовательность вызова тех или иных функций. На данный момент, существует множество программных решений, которые восстанавливают исходный код в полном соответствии оригиналу. Самой базовой методикой защиты является применение техник обфусцирования, описанных в главе 2. К сожалению, если злоумышленнику известен используемый механизм обфускации, то для него не составит большой сложности обход этой защиты, и спустя некоторое время, исходный MCIL код будет восстановлен.

3.3 Защита промежуточного представления программного продукта.

Следующим этапом защиты является защита байт-кода, используя особенности объектно-ориентированного программирования. После того, как злоумышленнику удалось обойти механизм обфускации, он столкнется с MSIL кодом, который представляет прямое отображение исходного кода. Для того, чтобы осложнить киберпреступнику статическую отладку кода, необходимо защитить фрагмент кода, который представляет интерес злоумышленника.

Платформа .Net Framework является уникальной с точки зрения взаимодействия разных языков программирования. Платформа является единой моделью, позволяющей на равных пользоваться различными языками для создания приложений. Так как MSIL не зависит от исходного языка программирования или от целевой платформы, в рамках .NET становится возможным развивать новые программы на базе старых программ - причем и первый, и второй языки программирования не важны. Благодаря такой универсальности, платформа наследует сложность указателей из языка программирования C++.

3.3.1 Указатели в платформе .Net Framework.

Использование указателей в платформе .Net Framework не рекомендуется, так как это нарушает принцип виртуальной машины, которая возлагает любое взаимодействие с памятью на себя. Но для совместимости с C++ были введены 2 категории указателей: управляемые и неуправляемые указатели.

Для того чтобы программы оставались безопасными, на использование управляемых указателей наложен целый ряд ограничений:

1. Управляемые указатели могут содержать только адреса ячеек, то есть они могут указывать исключительно только на глобальные и локальные переменные, параметры методов, поля объектов и ячейки массивов.
2. За каждым указателем закреплен тип ячейки, на которую он может указывать. Другими словами, запрещены void-указатели.
3. Указатели могут храниться только в локальных переменных и параметрах методов.

4. Запрещены указатели на массивы.

На использование неуправляемых указателей никаких ограничений не накладывается, то есть они могут содержать абсолютно любой адрес. Программа, в которой используются неуправляемые указатели, автоматически считается небезопасной.

Для усложнения процесса статического анализа программы, разработчику необходимо написать фрагмент кода, который будет заведомо небезопасным с точки зрения платформы .Net Framework. Последовательно модифицируем код, рассматриваемый в главе 3.2, и проследим эволюцию защиты промежуточного представления программы.

3.3.2 Запутывающие методы.

Во-первых, добавим уровень абстракции в исходный код программы.

MC++ представление:

```
class Base
{
protected:
        int SecurityLevel;

public:
   virtual int GetSecurityLevel( const int p_Password ) = 0;
};

class TopSecurity : public Base
{
public:
   TopSecurity() { SecurityLevel = 0x7FFFFFFF; }

   int GetSecurityLevel( int p_Password )
   {
      if( 1 != p_Password )
     {
         return 0xffffffff;
     }
     return SecurityLevel;
   }
};
```

```
void main( int Password )
{
        Base *Obj = new TopSecurity();
        int SecLevel = Obj->GetSecurityLevel( Password );
}
```

Представление в байт-коде MSIL (некоторый код опущен):

```
.locals ( [0] valuetype TopSecurity* V_0,
          [1] valuetype Base* Obj,
          [2] valuetype TopSecurity* V_2,
          [3] int32 SecLevel )

 /* проверка возможности выделить память */

IL_0008:  ldloc.0
IL_0009:  brfalse.s  IL_0013
IL_000b:  ldloc.0
IL_000c:  call       valuetype TopSecurity* … 'TopSecurity.{ctor}' …
IL_0011:  br.s       IL_0014
IL_0013:  ldc.i4.0
IL_0014:  stloc.2
IL_0015:  leave.s    IL_001e

/* загрузка на вершину стека аргумента и локальной переменной [1] */

IL_0025:  calli      unmanaged thiscall int32
modopt([mscorlib]System.Runtime.CompilerServices.CallConvThiscall)(native
int,int32 modopt([mscorlib]System.Runtime.CompilerServices.IsConst))
```

Как видно из выше приведенного примера, внедрение небольшого уровня абстракции и инстанцирование объекта через указатель, скрывает прямое упоминание вызываемых методов класса (IL_0025). Этот прием усложняет процесс восстановления логики работы приложения при непосредственном анализе байт-кода.

Пример восстановленного кода через .Net Reflector:

```
internal static unsafe int modopt(CallConvCdecl) main(int Password)
{
        TopSecurity* securityPtr2;
        TopSecurity* securityPtr = @new(12);
        try
        {
            securityPtr2 = (securityPtr == null)
                        ? null : TopSecurity.{ctor}(securityPtr);
        }
        fault
        {
            delete((void*) securityPtr);
        }

        Base* Obj = (Base*) securityPtr2;
        int SecLevel = **(*((int*) Obj))(Obj, Password);

        return 0;
}
```

Как видно методы вызываются через указатели, получив восстановленный код в .Net Reflector, злоумышленнику не составит большого труда разобраться в логике приложения.

Таким образом, можно утверждать, что наличие у класса множества одинаковых по параметрам методов, и создание объекта через указатель повышает уровень противодействия кода статическому анализу, при условии прямой работы с MSIL кодом.

```
class TopSecurity : public Base
{
public:
        TopSecurity() { SecurityLevel = 0x7FFFFFFF; }

        int GetSecurityLevel( int p_Password )
        {
             if( 1 != p_Password )
            {
                return 0xffffffff;
            }
            return SecurityLevel;
         }
```

```
        int CheckSecurityLevel( int p_Password )
        {
            if( 0x43211 == MD5( p_Password )  )
            {
                return 0xffffffff;
            }
            return 0x0101;
        }
};
```

Добавление запутывающих методов, усложняет исследование кода. В случае если злоумышленник добрался до фрагмента кода, который отвечает за обеспечение безопасности, он вынужден более детально анализировать логику вызова, так как явный вызов в коде отсутствует.

3.3.3 Дополнительный уровень абстракции.

Опираясь на прошлый раздел, определим, что уровень абстракции позволяет скрыть вызовы тех или иных методов. Для того чтобы повысить безопасность, можно скрыть операции взаимодействия с защищаемым объектом под дополнительным уровнем абстракции.

Перед анализом этого метода, опишем исходные условия. Защищаемый объект наследуется от интерфейса Base и имеет некоторое количество реализаций TopSecurtity, NoneSecurity и другие, из которых только один реализует правильный алгоритм обработки входных данных. Базовый интерфейс подразумевает реализацию у классов наследников нескольких методов обработки входных данных, которые имеют одинаковую семантику.

```
class Base
{
    virtual int CheckSecurityLevel( int p_pPassword ) = 0;
    virtual int AnalizeSecurityLevel( int p_pPassword ) = 0;
    virtual int GetSecurityLevel( int p_pPassword ) = 0;
};
```

Чтобы скрыть работу с защищаемым объектом, воспользуемся классом-оберткой, который на этапе инициализации создает экземпляр корректного

класса и в дальнейшем обработка входных данных осуществляется через созданный нами класс-посредник.

```
class WrapperBase
{
protected:
   Base *pSecurity;

public:
   virtual int Operation( int p_Val ) = 0;
};

class Wrapper : public WrapperBase
{
public:
   Wrapper( Base *p_pSecurity ) { pSecurity = p_pSecurity; }

public:
   int Operation( int p_Val )
   {
      return pSecurity->GetSecurityLevel( p_Val );
   }
};
```

Таким образом, в методе Operation мы имеет следующий код в промежуточном представлении (некоторый код опущен):

```
 .locals ( [0] int32 V_0 )

  IL_0000:  ldarg.0
  IL_0001:  ldc.i4.4
  IL_0002:  add
  IL_0003:  ldind.i4
  IL_0004:  ldarg.1
  IL_0005:  ldarg.0
  IL_0006:  ldc.i4.4
  IL_0007:  add
  IL_0008:  ldind.i4
  IL_0009:  ldind.i4
  IL_000a:  ldind.i4
```

```
IL_000b:  calli   unmanaged thiscall int32 … (native int,int32) …
```

.Net Reflector восстанавливает исходный код следующим образом:

```
internal static unsafe Wrapper* modopt(CallConvThiscall) Wrapper.{ctor}
(Wrapper*   modopt(IsConst) modopt(IsConst) local1, Base* p_pSecurity)
{
   WrapperBase.{ctor}(local1);
   local1[0] = &??_7Wrapper@@6B@;
   local1[4] = (Wrapper* modopt(IsConst) modopt(IsConst)) p_pSecurity;
   return local1;
}

internal static unsafe int modopt(CallConvThiscall) Wrapper.Operation(Wrapper*
modopt(IsConst) modopt(IsConst) local1, int p_Val)
{
        return **(*(local1[4]))(local1[4], p_Val);
}
```

Деликатным местом в описанном механизме защиты является знание класса-обертки о базовом классе защищаемого объекта. Чтобы повысить уровень сложности, попробуем описать защищаемое поле класса-посредника Wrapper через неуправляемый указатель void*. Таким образом, при статическом анализе кода предназначение Wrapper будет невозможно идентифицировать, так как в нем не останется ни описания инкапсулируемого объекта, ни методов, для которых он является посредником.

Таким образом, заменим тип указателя в поля protected базового класса-посредника WrapperBase на void *pSecurity и конструктор класса Wrapper заменим на Wrapper(void *p_Security). Исходя из этих изменений, необходимо модифицировать и вызов операций в классе-посреднике:

```
int Operation( int p_Val )
{
     return ((Base *)pSecurity)->GetSecurityLevel( p_Val );
}
```

Эти изменения в исходном коде повлияли только на описание поля pSecurity, при этом оставив содержимое методов класса-обертки без изменения.

```
.method assembly static valuetype Wrapper*
modopt([mscorlib]System.Runtime.CompilerServices.CallConvThiscall)
       'Wrapper.{ctor}'(valuetype Wrapper*
modopt([mscorlib]System.Runtime.CompilerServices.IsConst)
modopt([mscorlib]System.Runtime.CompilerServices.IsConst) A_0,
                                void* p_pSecurity) cil managed
```

Как видно из фрагмента кода p_pSecurity стал указывать на абстрактные данные, тем самым скрыв реальный тип объекта.

Таким образом, получаем, что дополнительный уровень абстракции скрывает все операции с защищаемым объектом. При попытке злоумышленника провести статический анализ кода, он сталкивается с защищенными вызовами функций обработки секретных данных, через неявные объекты. Для увеличения уровня сложности анализа, необходимо применить методику запутывающих методов к классу-посреднику Wrapper.

Проанализируем вызов класса Wrapper.

```
void main( int Password )
{
   WrapperBase *Obj = new Wrapper( new TopSecurity() );
   int SecLevel = Obj->Operation( Password );
}
```

В промежуточном представлении код имеет следующий вид (некоторый код опущен):

```
  .locals ( [0] valuetype TopSecurity* V_0,
            [1] valuetype Wrapper* V_1,
            [2] valuetype WrapperBase* Obj,
            [3] valuetype Wrapper*
            [4] valuetype Wrapper* V_4,
            [5] valuetype TopSecurity* V_5,
```

```
            [6] int32 SecLevel )
/* проверка возможности выделить память под объект Base */
/* проверка возможности выделить память под объект Wrapper */
IL_0016:  call  valuetype TopSecurity* … 'TopSecurity.{ctor}' …
IL_001b:  br.s       IL_001e
IL_001d:  ldc.i4.0
IL_001e:  stloc.s    V_5
IL_0020:  leave.s    IL_0029
…
IL_0029:  ldloc.1
IL_002a:  ldloc.s    V_5
IL_002c:  call       valuetype Wrapper* … 'Wrapper.{ctor}' …
…
IL_0031:  stloc.3
IL_0032:  br.s       IL_0036
IL_0034:  ldc.i4.0
IL_0035:  stloc.3
IL_0036:  ldloc.3
IL_0037:  stloc.s    V_4
IL_0039:  leave.s    IL_0042
…
/* загрузка объекта Wrapper и аргумента функции на стек для вызова */
IL_004a:  calli  unmanaged thiscall int32 modopt ( [mscorlib]
System.Runtime.CompilerServices.CallConvThiscall ) (native int,int32)
```

Восстановленный исходный код через .Net Reflector:

```
internal static unsafe int modopt(CallConvCdecl) main(int Password)
{
   Wrapper* wrapperPtr3;
   Wrapper* wrapperPtr = @new(8);
   try
```

```
    {
        Wrapper* modopt(IsConst) modopt(IsConst) wrapperPtr2;
        if (wrapperPtr != null)
        {
            TopSecurity* securityPtr2;
            TopSecurity* securityPtr = @new(12);
            try
            {
                securityPtr2 = (securityPtr == null) ? null : TopSecurity.{ctor}(securityPtr);
            }
            fault
            {
                delete((void*) securityPtr);
            }
            wrapperPtr2 = Wrapper.{ctor}(wrapperPtr, (void*) securityPtr2);
        }
        else
        {
            wrapperPtr2 = 0;
        }
        wrapperPtr3 = wrapperPtr2;
    }
    fault
    {
        delete((void*) wrapperPtr);
    }
    WrapperBase* Obj = (WrapperBase*) wrapperPtr3;
    int SecLevel = **(*((int*) Obj))(Obj, Password);
    return 0;
}
```

3.3.4 Неявное инстанцирование объекта.

Предыдущий метод защиты маскирует работу с защищаемым объектом. Злоумышленнику усложняется процесс определения, как обрабатываются данные, подаваемые на вход класса Wrapper. Однако киберпреступнику доступен вызов объекта Wrapper с явно указанным защищаемым объектом. Таким образом, злоумышленник, в случае успешного определения защищаемого объекта, может начать исследовать все его методы и пытаться выявить тот алгоритм, который представляет для него интерес. Для увеличения сложности статического анализа необходимо использовать метод неявного инстанцирования объекта.

Перед применением схемы защиты, предполагается, что предыдущие методики используются в исходном коде. Перед анализом этого механизма опишем начальные условия. Защищаемый объект представлен наследником базового класса Base, у которого есть некоторое число объектов, аналогичных классу TopSecurity. К защищаемому объекту применена методика запутывающих методов и написан дополнительный уровень абстракции, который принимает указатель на абстрактные данные. Целью данной методики является сокрытия имени объекта, который оборачивает класс-посредник Wrapper.

Для начала, реализуем схему передачи объекта в конструктор класса Wrapper, через специализированную функцию, которая возвращает void*.

```
void* Instance( void );
```

В соответствии с этим перепишем создание объекта Wrapper через функцию Instance:

```
WrapperBase *Obj = new Wrapper( Instance() );
```

Таким образом, в функции, создающей объект Wrapper и выполняющей действия с секретными данными, название объекта никак не фигурирует.

Определим способ создания объекта через уровень абстракции, который будет проверять наличие указателя на защищенный объект и если он есть, то

возвращать его, иначе создавать. Реализация этого класса подразумевает наличие базового класса:

```
class AbstractBase
{
public:
        virtual void* GetTopSecurity()   = 0;
        virtual void* GetNoneSecurity() = 0;
        ….
};

class AbstractFactory : public AbstractBase
{
        static TopSecurity *pTop;
        static TopSecurity *pNone;

public:
        void* GetTopSecurity()
        {
                if( nullptr == pTop )
                        pTop = new TopSecurity();
                return pTop;
        }

        void* GetNoneSecurity()
        {
                if( nullptr == pNone )
                        pNone = new NoneSecurity();
                return pNone;
        }
        …
};
```

Исходя из этого, можно определить содержимое функции Instance:

```
void* Instance( void )
{
   AbstractBase* Factory = new AbstractFactory();
   return  Factory->GetTopSecurity();
}
```

Так как уровень абстракции обеспечивает сокрытие вызываемых методов, то можно утверждать, что в конечной версии промежуточного кода тип

создаваемого объекта в явном виде присутствовать не будет. Вызывающая функция будет иметь следующий вид:

```
void main( int Password )
{
   WrapperBase *Obj = new Wrapper( Instance() );
   int SecLevel = Obj->Operation( Password );
}
```

Представление функции Instance в промежуточном коде (некоторый код опущен):

```
  .locals ( [0] valuetype AbstractFactory* V_0,
            [1] void* V_1,
            [2] valuetype AbstractBase* Factory )
/* проверка возможности выделить память под объект AbstractFactory, создание объекта */
  IL_0018:  calli  unmanaged thiscall void* … (native int)
```

Класс AbstractFactory может реализовывать создание большого количества объектов разного типа, поэтому определить тип создаваемого объекта, для класса-посредника Wrapper, злоумышленнику буду сделать сложнее и займет больше количество ресурсов.

3.4 Защита программы от специализированных инструментов отладки

Приведенные выше средства для увеличения сложности анализа кода теряют смысл, когда злоумышленник видит восстановленный код через инструменты дизассемблирования. Анализ логики приложения через байт-код, может вызвать ряд сложностей, который ведут к увеличению времени реверс инжениринга.

Вопрос защиты байт-кода от реверс инжениринга интересовал разработчиков на протяжении длительного времени. Программы для статического анализа, имеют несколько разновидностей:

1. Просмотр промежуточного байт-кода IL. Основной представитель в этой категории продукт Ildasm от компании Microsoft. Предназначение этой утилиты – анализа кода, порождаемого после компиляция.
2. Просмотр промежуточного байт-кода IL с попыткой восстановления исходного кода. Утилиты этого типа широко применяются злоумышленниками для анализа и изменения логики чужого программного продукта.

Для предотвращения использования программ восстанавливающих исходный код, приложение должно содержать механизмы, которые выводят из строя инструменты дизассемблирования, но при этом сохраняют работоспособность программы и не нарушают логику.

3.4.1 Встроенные механизмы защиты среды .Net Framework.

Утилита Ildasm используется для низкоуровневой отладки приложения, модификации и пересборки приложений. Чтобы оградить круг лиц, которым доступен низкоуровневый код приложения, компания Microsoft добавила дополнительный атрибут сборки. SuppressIldasm – этот атрибут предотвращает отладку приложения средствами утилиты Ildasm.

При сборке приложения с SuppressIldasm, изменяется содержимое PE-заголовка. Выставляется дополнительный флаг в секции метаданных (рис. 3.2).

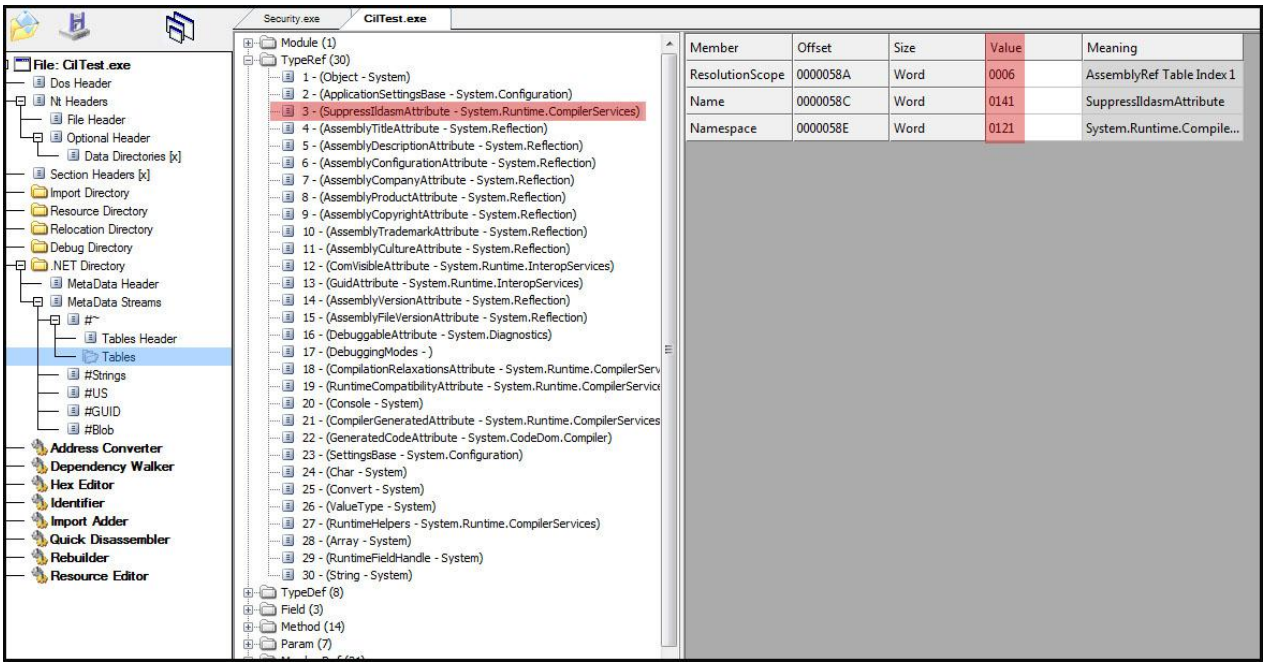

Рис. 3.2 Защита от дизассемблирования утилитой Ildasm.

Каждая запись в таблице TypeRef соответствует одному из импортируемых типов и содержит следующие поля:

1) ResolutionScope – Специальным образом закодированная информация о том, какой сборке или какому модулю принадлежит данный тип.
2) Name – Индекс в массиве строк, по которому храниться имя импортируемого типа.
3) Namespace – Индекс в массиве строк, по которому храниться имя пространства имен; данному пространству принадлежит импортируемый тип.

Для обхода описанной выше защиты злоумышленнику достаточно обнулить значение TypeRef для типа SuppressIldasmAttribute.

Так же компания Microsoft предлагает разработчикам защищать программный продукт путем подписания сборки по технологии Strong Name Signature (SNS). Цифровая подпись используется, чтобы проверить целостность данных, которые отправляются от отправителя к получателю. Подпись формируется и проверяется, используя публичный криптографический ключ. Автор подписи имеет два типа ключей: публичный, который знают все пользователи, и приватный – сохраняемый в секрете. Технология Strong-Name Signing представляет механизм подписания .Net сборок.

Когда платформа .Net Framework загружает подписанную сборку, она проверяет, что загружаемая сборка использует подписанные библиотеки. Если Strong-Name Singing не проходит проверку, то сборка не будет загружена. Таким образом, SNS обеспечивает целостность данных, в рамках, используемых в сборке компонентов. Для проверки подписи в одиночном модуле необходимо организовать верификацию в коде посредством публичного токена ключа, а так же вызова API для проверки корректности сборки.

```
[DllImport("mscoree.dll", CharSet = CharSet.Unicode)]
static extern bool StrongNameSignatureVerificationEx( string wszFilePath,
                                                      bool fForceVerification,
                                                      ref bool pfWasVerified );
private static bool IsValidSignature ()
{
   bool NotForced = false;

   /* Токена отклытого ключа, нашей ключевой пары */
   byte[] PublicKey = { 0x67, 0xd1, 0x3a, 0x85, 0xdf, 0x57, 0x45, 0x09 };

   /* Значение токена отклытого ключа, с использованием
      которой подписана сборка */
   byte[] ActualKey = System.Reflection.Assembly.GetExecutingAssembly().
                           GetName().GetPublicKeyToken();

   /* Если сборка не подписана, то это попытка модификации кода */
   if (ActualKey == null)  return false;
   if (ActualKey.Length != PublicKey.Length) return false;

   for( int i = 0; i < PublicKey.Length; i++ )
      /* Если токены не совподают, то это попытка модификации кода */
      if (ActualKey[i] != PublicKey[i])  return false;
```

```
    System.Console.WriteLine("Token have been verified!" );

    /* проверка подписи сборки */
    return StrongNameSignatureVerificationEx( System.Reflection.Assembly.
                    GetExecutingAssembly().Location, false, ref NotForced );
}
```

3.4.2 Сокрытие тела методов.

Для сокрытия дизассемблеров с восстановлением исходного кода необходимы более интеллектуальные решения, которые инициируют непредвиденное поведение средств отладки.

Каждый код IL инструкции представляется одним – двумя байтами. Большинство из пространства инструкций зарезервировано для дальнейшего использования. Переход JIT компилятора на инструкцию из области зарезервированных команд, приводит к аварийному завершению программы-анализатора. Таким образом, если отладчик встречает не валидную инструкцию, то дальнейший разбор кода прекращается. В этом случае программа-анализатор проинформирует злоумышленника о не возможности разбора.

3.4.3 Редактирование PE-заголовка сборки.

Исполняемый файл – это файл, который может быть загружен в память загрузчиком операционной системы и затем исполнен. Для того чтобы загрузчик операционной системы мог правильно загрузить исполняемый файл в память, содержимо этого файла должно соответствовать принятому в данной операционной системе формату исполняемых файлов. Формат PE – это основной формат для хранения исполняемых файлов в операционной системе Windows. Сборки .Net тоже хранятся в этом формате.

PE-формат сборки почти полностью повторяет стандартный формат PE-файла. В PE-формате сборки также присутствуют основной и дополнительный

заголовки, секции, которые представляют некоторые данные или код. Особенность сборок – это наличие метаданных. Метаданные служат для описания типов, находящихся в сборке .Net, и хранятся в исполняемых файлах.

Метаданные начинаются с заголовка, называемого корнем метаданных. В конце заголовка метаданных идет 16-разрядное целое число, содержащее количество потоков метаданных.

Поток метаданных предназначен для хранения информации определенного вида. В спецификации определено пять видов потоков метаданных. Четыре потока представляют хранилища однородных объектов, и один – таблицы метаданных. Возможно наличие десятков таблиц метаданных.

Поток метаданных содержит полную информацию о сборке, в частности описание методов, объявленных в сборке. Каждая запись в таблице методов содержит информацию об одном методе, представленную в полях:

1. RVA тела метода в исполняемом файле.
2. Индекс в куче строк, по которому храниться имя метода.
3. Индекс двоичных данных, по которому расположена сигнатура метода
4. Индекс в таблице описателей параметров метода

Статические анализаторы используют метаданные PE-формата сборки для построения графа вызова методов и сбора информации о них. JIT-компилятор в процессе исполнения сборки, компилирует ее на лету в native код, так как исполнение происходит на платформе x86, то результатом работы JIT-компилятора является ассемблер x86. В ассемблере информация о классах, методах, переменных отсутствует. Используется только адреса. На основании этого, можно заключить, что JIT компилятор минимально обращается к секции метаданных, преимущественно он использует секцию кода. Исходя из этого, можно предположить, что частичная модификация секции метаданных приведет к некорректному поведению статических анализаторов, но при этом сохранит валидное поведение сборки.

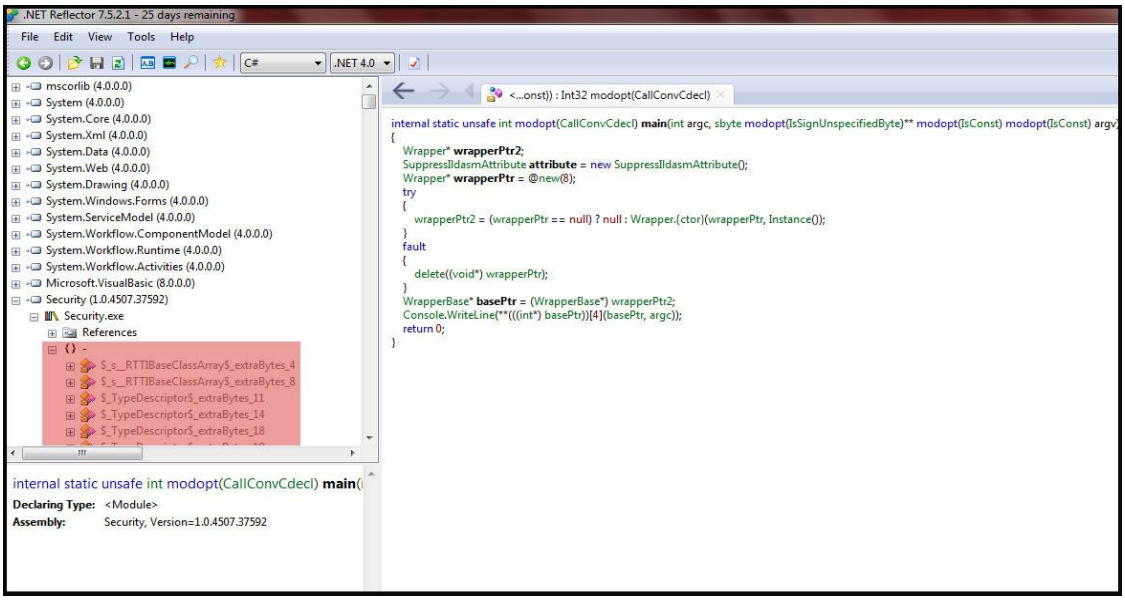

Рис. 3.3 Восстановление кода сборки статическим анализатором.

На рис. 3.3 приведен фрагмент корректной работы статического анализатора кода. Изучив формат таблицы методов в секции метаданных, автор провел исследования и определил, модификация каких полей может привести к желаемому результату.

Результат исследования – это новый метод защиты сборок .Net Framework от статического анализатора с восстановление исходного кода. От разработчика требуется завести избыточный метод, вызов которого добавить в защищаемый объект. После пересборки программы, необходимо вручную отредактировать PE-файл, а именно изменить поле ParamList, в секции метаданных, который отвечает за хранения индекса в таблице описателей параметров метода. На рис. 3.4 изображено поведение статического анализатора при попытке восстановить исходный код защищенной сборки.

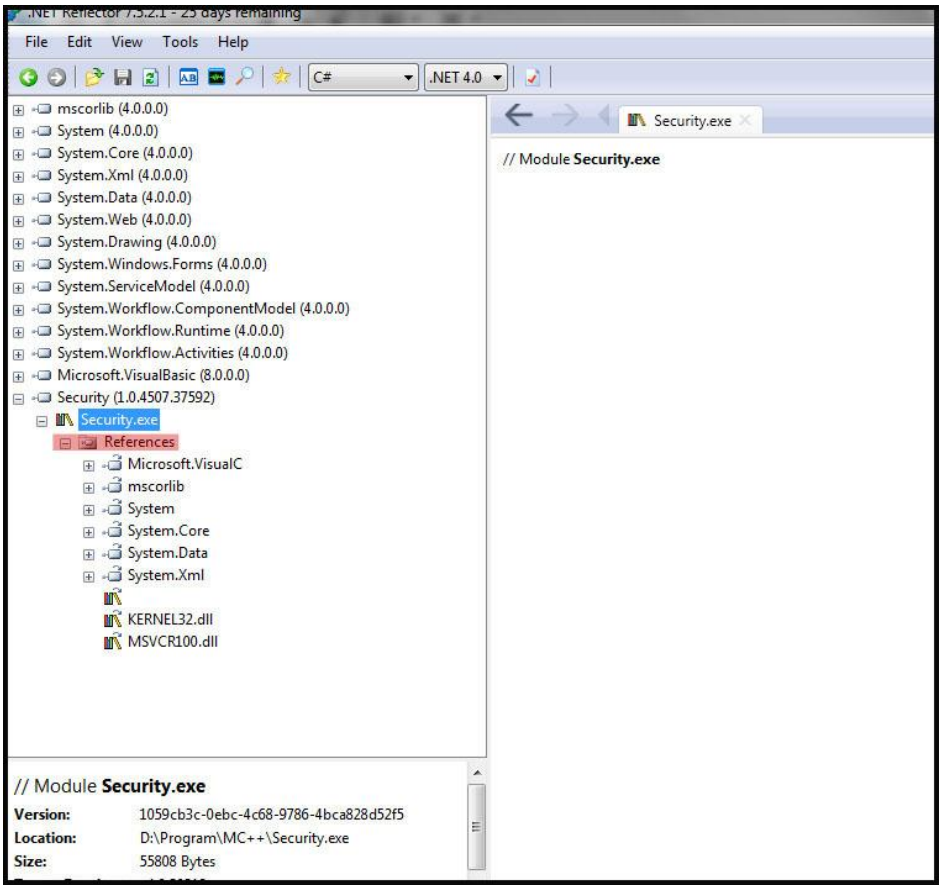

Рис 3.4 Восстановление кода защищенной сборки статическим анализатором

Как можно заметить, статический анализатор не определяет ни один метод защищенной сборки.

3.5 Защита ассемблерного представления программы.

Завершающим этапом защиты является ассемблерное представление программного продукта. Если затраты времени на статический анализ превышают разумные пределы, то злоумышленник прибегнет к динамическому анализу. JIT компилятор компилирует сборку на лету, таким образом, при запуске программы можно пытаться отладить ее ассемблерное представление. Платформа .Net Framework предполагает возможность запуска сборки на любом типе процессоров и на любой целевой машине, главное условие – наличие виртуальной машины. Эта особенность накладывает сильные ограничения, разработчик не может использовать ассемблер из сборки, написанной на .Net Framework. Если разработчик определил какой сегмент процессоров он поддерживает в своей сборке, то можно применить некоторые

техники дающие возможность внедрить ассемблерные вставки в исходный код программы.

3.5.1 Инъекция.

Один из способов получить возможность сделать ассемблерную вставку – это использование инъекции. Эта техника подразумевает знание об основных принципах формирования ассемблерного кода под процессор.

Для большинства языков высокого уровня свойственно использовать регистр EBP для доступа к аргументам функции и локальным переменным. Кадр стека внутри функции представлен на рис. 3.5.

Содержание стека	Адресация FLAT MODEL
Аргумент ...	...
Аргумент 2	[EBP+A]
Аргумент 1	[EBP+8]
Адр.возвр.(смещ.)	[EBP+4]
EBP при входе	[EBP]
Лок.перем. 1	[EBP-4]
Лок.перем. 2	[EBP-8]
Лок.перем. ...	...

Рис. 3.5 Кадр стека внутри функции.

Для сохранения значения EBP неизменным внутри функции используются следующие команды при входе в функцию:

push ebp

mov ebp, esp.

При выходе из функции значение EBP восстанавливается из стека. Исходя из того, что значение регистра EBP остается неизменным, относительного него всегда можно определить местоположение адреса возврата из функции.

Таким образом, на основании этих знаний можно сформировать ассемблерную инъекцию в код. Для этого необходимо сформировать массив байт, который содержит ассемблерные команды необходимые разработчику.

Затем создать функцию, и определив адреса возврата из функции, заменить на адрес массива с ассемблерными командами. Таким образов, при выходе из функции, управление получит массив с командами разработчика. В операционной системе Windows, массив находится в памяти, которая имеет права только на чтение и запись. Следовательно, в функции, которая подменяет адрес возврата, необходимо дать изменить права на область памяти, где находиться массив с ассемблерными командами. Для того, чтобы выполнение прошло успешно и после вызова функции, выполняющей инъекцию, в массиве ассемблерных команд необходимо гарантировать корректный возврат в вызывающую функцию. Для этого необходимо чтобы вызываемая функция возвращала корректный адрес возврата. При выполнении этого условия, перед выходом из функции адрес возврата будет размещен в регистре общего назначения EAX. Следовательно, последними командами ассемблера должны быть восстановления адреса возврата из EAX и выполнение команды ret.

Пример кода, реализующий вызов массива с ассемблером

```
static void Main(string[] args)
{
   int b = AsmHelper.CallAssembler(AsmInject);

   Console.WriteLine( "All was fine!" );
   Console.ReadLine();
}

unsafe public static class AsmHelper
{
   const uint OFFSET = 0x40;
   const uint PAGE_EXECUTE_READWRITE = 0x40;

   [DllImport("kernel32.dll")]
   static extern bool VirtualProtect( int  *p_pAddress,
                                      uint   p_Size,
                                      uint   p_NewProtect,
                                      uint *p_pOldProtect  );

   public static int CallAssembler(byte[] AsmArray)
   {
      int Ret;
```

```
        int* pRet = &Ret;

        /* определяем адрес возврата */
        pRet = pRet + (OFFSET + 4) / sizeof(int);
        Ret = *pRet;

        fixed (byte* AsmByte = &AsmArray[0])
        {
           bool Res;
           uint PreviousRights;

            /* Заменяем адрес возврата на массив с ассемблером */
           *pRet = (int)AsmByte;

           /* разрешить изменить код в функции CallAssembler */
           int* EBP = (int*)new IntPtr(*(pRet - 1)).ToPointer();
           Res = VirtualProtect( EBP - 1, sizeof(int),
                                 PAGE_EXECUTE_READWRITE,
                                 &PreviousRights );

           /* разрешить массиву исполнять код */
           int* pAsmArray = (int*)new IntPtr(AsmByte).ToPointer();

           /* [ptr-2]<array type> [ptr-1]<array size> [0]<element> */
           uint AsmArraySize = (uint)*--pAsmArray;

           Res = VirtualProtect(  pAsmArray,
                                 (uint)AsmArraySize,
                                 PAGE_EXECUTE_READWRITE,
                                 &PreviousRights );
        }

        return *pRet;
     }
}
```

Ассемблерную инъекцию можно использовать в качестве защиты от динамической отладки. Для этого нужно в ассемблерной вставке реализовать логику отслеживания отладчика и алгоритм противодействия ему. Для детектирования отладчика можно использовать Process Environment Block, и в случае выявления отладчика организовать запутывающий код, который не приводит к защищаемому блоку. Так же пользуясь инъекцией можно реализовать алгоритма шифрования входных данных перед проверкой.

3.5.2 DLL-защита.

Альтернативным способом инъекции является использование динамически подгружаемых библиотек. Этот способ хорош тем, что он может содержать всю логику проверки легальности действий пользователя в отдельно выделенной сущности. В библиотеке можно использовать родной код для определенной архитектуры. Она может концентрировать в себе любые классические подходы защиты, применяемые для обычных приложений.

Для обхода такой зашиты злоумышленнику достаточно определить библиотеку реализующую проверки легальности и заменить ее на свою реализацию.

4 Заключение.

Анализ существующих технологий защиты программного обеспечения, написанного посредством виртульных машин, привел к выводу о том, что эта область информационной безопасности имеет ряд недостатков. Процесс написания программ сильно упростился, а вместе с тем, возросла эффективность мошенников, применяющих различные методики взлома. Реверс инжениринг стал более доступен в силу появления нового уровня абстракции – промежуточного байт кода. Анализ проблем безопастности для приложений, исполняемых в среде .Net Framework, показал не состоятельность большинства существующих методик защиты. Это обстоятельство потребовало постановки научной задачи по разработке методик повышения безопасности, в ходе решения которой получены следующие наиболее существенные результаты:

1. Определены наиболее состоятельные решения, на основании которых, сформулированы и оптимизированы методики по реализации элементов защиты на основе техники обфусцирования.
2. Исследован промежуточный код виртуальной машины .Net Framework. Выявлены уязвимые места, требующие дополнительного внимания при написании программы. Разработаны рекомендации, на основании объектно-ориентированного подхода, которые обеспечивают повышение уровня сложности обратной инженерии.
3. Проанализирован принцип работы инструментов дизассемблирования, применяемые злоумышленниками для исследования программ. На основании полученных знаний, разработаны и описаны техники вывода из строя этих программных комплексов.
4. Разработан методов инъекции кода на языке низного уровня, реализующий защиту от динамической отладки приложения.

Научная новизна проделанной работы заключается в том, что разработаны новые механизмы для обеспечения защиты программных комплексов с предусмотренной системой лицензирования.

Теоретическая значимость полученных результатов определяется, разработкой методик противостояния атакам киберпреступников, позволяющая повысить уровень защищености правообладателя продукта.

Практическая значимость работы заключается в разработке концептуального подхода к организации механизма защиты программы. Набор разработанных методик может быть использован в практической деятельности каждого разработчика, особенно при планировании дальнейшего распространения продукта на недоверенные хосты. Обобщенные и разработанные методики являются универсальными, предоставляют мощный механизм защиты фрагментов интеллектуальной собственности.

5 Список используемой литературы.

1. Ильдар Инкуартрос Защита .NET приложений от просмотра [Электронный ресурс] / Инкуартрос Ильдар – Режим доступа: http://netobf.com/Protection-dotNET-applications. Дата обращения 05.01.2012.
2. Николай Владимирович Лихачёв Обфускация и ее преодоление / Лихачёв Николай Владимирович - Режим доступа: http://www.insidepro.com/kk/080/080r.shtml. Дата обращения 04.01.2012.
3. Сергей Александрович Середа Проблема теневого рынка программных продуктов и пути ее разрешения [Электронный ресурс] / Середа Сергей Александрович – Режим доступа: http://consumer.stormway.ru/piracy.htm. Дата обращения 10.11.2011.
4. Microsoft Основные понятия безопасности [Электронный ресурс] / Microsoft – Режим доступа: http://msdn.microsoft.com/ru-ru/library/z164t8hs(v=VS.90).aspx. Дата обращения 28.10.2011.
5. Matteo Slaviero Что нового в Code Access Security в .NET Framework 4.0 [Электронный ресурс] / Matteo Slaviero – Режим доступа: http://www.k-press.ru/cs/2010/3/CAS/CAS.asp. Дата обращения 30.11.2011.
6. Microsoft Прозрачный для системы безопасности код [Электронный ресурс] / Microsoft – Режим доступа: http://msdn.microsoft.com/ru-ru/library/ee191569.aspx. Дата обращения 28.10.2011.
7. Макаров, А.В. Common Intermediant Language и системное программирование в Microsoft .Net : учебное пособие / А.В. Макаров, С.Ю. Скоробогатов, А.М. Чеповский – М. : БИНОМ. Лаборатория знаний. – 2007. – С. 9.
8. Макаров, А.В. Common Intermediant Language и системное программирование в Microsoft .Net : учебное пособие / А.В. Макаров, С.Ю. Скоробогатов, А.М. Чеповский – М.: БИНОМ. Лаборатория знаний. – 2007. – С. 5.
9. Макаров, А.В. Common Intermediant Language и системное программирование в Microsoft .Net : учебное пособие / А.В. Макаров, С.Ю.

Скоробогатов, А.М. Чеповский – М. : БИНОМ. Лаборатория знаний. – 2007. – С. 7.

10. Bagie Жизнь после компиляции [Электронный ресурс] / Bagie – Режим доступа: http://www.xakep.ru/magazine/xs/057/074/1.asp. Дата обращения 10.11.2011.
11. Николай Владимирович Лихачёв Обфускация и защита программных продуктов [Электронный ресурс] / Лихачёв Николай Владимирович – Режим доступа: http://citforum.ru/security/articles/obfus/#5. Дата обращения 11.11.2011.
12. Антон Димитров Обфускаторы и деобфускаторы для .Net [Электронный ресурс] / Димитров Антон – Режим доступа: http://habrahabr.ru/blogs/net/74463/. Дата обращения: 07.01.2012.
13. Микалай Калпински Обфускация взгляд изнутри [Электронный ресурс] / Калпински Микалай – Режим доступа: http://sharcus.blogspot.com/2011/06/blog-post.html. Дата обращения: 07.12.2012.
14. Christian S. Collberg Breaking abstractions and unstructuring data structures. In International Conference on Computer Languages / Christian S. Collberg, Clark D. Thomborson, and Douglas Low. – 1998. – С. 28–38.