Guardant®

Система защиты от компьютерного пиратства

Эффективная
защита приложений

Урок 5.2: стойкость загружаемого кода

Содержание

Общее описание технологии

Технология загружаемого кода состоит в том, что в электронный ключ можно поместить собственный специально подготовленный код и, далее, в процессе работы приложения исполнять его.

Напомним, что подразумевать под ней выгрузку части кода приложения в ключ некорректно. Это бессмысленно с точки зрения безопасности и зачастую просто невозможно из-за наличия в коде внешних обращений.

Загружаемый код позволяет решить несколько важных **задач**:

- Защита от анализа и копирования помещенных в ключ алгоритмов (похожим свойством обладают лишь защиты, основанные на размещении части функционала программы в Интернет);
- Защита от анализа приложений на языках, защита кода которых крайне затруднена или вовсе невозможна;
- Повышение стойкости защиты приложения в целом, в том числе от эмуляции электронного ключа на уровне API.

Технология позволяет реализовать автомат Мили с памятью, в отличие от всех предыдущих поколений ключей, в которых мог быть реализован лишь конечный автомат без памяти (комбинационная схема) или автомат с постоянной памятью.

Если грамотно воспользоваться этим преимуществом, то сложность построения табличного эмулятора злоумышленником существенно возрастет. Число возможных «ответов» на одно и то же обращение к ключу будет не одним, а близким к числу возможных внутренних состояний загружаемого кода (а точнее, состояний памяти, которую он использует для расчета очередного «ответа»).

Имея возможность загружать в ключ собственный код, мы обретаем возможность использовать принципиально новый математический аппарат для реализации собственной защиты и, что еще более важно, возможность изменения логики работы ключа под конкретное приложение.

На протяжении урока будет иметь место разделение на Native- и .NET-приложения. Некоторые технологии защиты неприменимы для приложений, распространяемых в байт-коде высокого уровня организации. Другие являются практически единственным средством гарантированной защиты кода от анализа (к таким как раз относятся ключи с загружаемым кодом). Именно поэтому защиту с их помощью необходимо выстраивать наиболее тщательно. Будем стараться уделять внимание обоим типам приложений.

Организация защиты

Ключевым фактором, определяющим стойкость защиты, является корректность выбора загружаемого кода.

В предыдущем уроке упоминалось о трех классах требований к загружаемому коду (изложенных во 2-й части Руководства пользователя).

Что касается требований **к реализуемости** и **производительности**, то с ними все относительно просто. Загружаемый код не должен содержать определенных конструкций и вызовов (изнутри ключа нет доступа к внешним объектам), а также должен быть относительно простым (выполняться быстро). Второе требование является менее строгим и в определенных случаях может не соблюдаться.

С классом требований **к безопасности** ситуация сложнее. Рассмотрим его описание из документации.

«Загружаемый код должен быть достаточно сложным, чтобы brute force или иные (более эффективные и продвинутые) методы анализа черных ящиков не сделали возможным создание эмулятора в короткое время.

Этот код должен отсутствовать в более ранних версиях приложения. Несоблюдение этого условия делает возможным сравнение версий приложения и нахождение перенесенного кода для внедрения его в эмулятор».

Первая часть описания говорит, что, имея доступ к рабочему электронному ключу, злоумышленник не должен иметь возможность

- Полностью или частично восстановить алгоритм преобразования данных в ключе
- Построить таблицу вопросов-ответов, пригодную для эмуляции ключа

Речь здесь идет о загружаемом коде в целом. Как мы узнаем позже, «полезная» составляющая алгоритма преобразования данных, помещенная в ключ, может оказаться сравнительно простой.

Вторая часть предостерегает от переноса в ключ некоторых алгоритмов или их отдельных частей «как есть» (особенно это касается приложений, код которых легко поддается анализу (.NET, Java и т.д.)).

Приведем несколько простых примеров на соответствие тем или иным требованиям.

Пример. Получение координат курсора мыши

Рассмотрим функцию получения координат курсора мыши. Она удовлетворяет требованиям к производительности и безопасности, так как в результате одного простого действия получаются координаты – случайные величины, которые можно практически напрямую использовать в приложении.

Не удовлетворяет эта функция лишь требованиям к реализуемости. Изнутри ключа нет доступа к аппаратным прерываниям компьютера, поэтому использовать ее не удастся.

Пример. Сложение двух чисел

Функция сложения двух чисел удовлетворяет требованиям к производительности и реализуемости. Что касается требований к безопасности, то в «чистом виде» использовать такую функцию не рекомендуется.

Однако она имеет серьезное преимущество. Результат ее выполнения можно легко использовать в приложении. Поэтому загружать в ключ такую функцию можно, несмотря на ее простоту. Необходимо лишь выполнить некоторые действия для защиты ее логики от анализа (об этом см. ниже).

Пример. Операции с точками на эллиптической кривой

Может показаться, что любые функции со сложными зависимостями между входными и выходными параметрами (к примеру, выполнение операций с точками эллиптической кривой) идеально подходят для загрузки в ключ.

В плане требований к безопасности это действительно так. Однако важно понимать, как именно та или иная функция будет использоваться при организации защиты приложения.

Если функциональность приложения такова, что результат столь сложного загружаемого кода будет использоваться «напрямую», то указанный код можно и нужно помещать в ключ. В противном случае ситуация не будет отличаться от защиты при помощи симметричного или асимметричного алгоритма, и загрузка такого кода не имеет большого смысла.

Технология загружаемого кода дает серьезное преимущество: если поместить в ключ «полезные» алгоритмы и использовать результаты их выполнения «напрямую», то можно отказаться от «искусственных» проверок, к которым приходилось прибегать при использовании встроенных криптографических алгоритмов ключа.

Защита загружаемого кода (как и защита приложения в целом) состоит из двух неотделимо связанных между собой аспектов:

- Защита от анализа самого загружаемого кода
- Защита логики вызывающего кода

Защита загружаемого кода от анализа

Загружаемый код не зависит от представления защищаемого приложения. Однако в случае с .NET, спектр доступных технологий защиты вызывающего кода несколько уже, чем для Native. Поэтому особое внимание необходимо уделить стойкости загружаемого кода, как основной линии обороны.

Если рассматривать загружаемый код как «черный ящик», то единственный доступный злоумышленнику интерфейс – это вызов функции **GrdCodeRun()** и передаваемые ей параметры. Теоретическая сложность анализа загружаемого кода определяется количеством этих параметров (или формально мощностью множеств возможных входных и выходных параметров).

Поэтому чем больше параметров передается загружаемому коду, тем больше ресурсов понадобится злоумышленнику для анализа. В то же время, длина передаваемого буфера существенно влияет на время вызова загружаемого кода (на передачу 1 КБ в одну сторону уходит порядка 40 мс). Необходимо соблюсти баланс между количеством параметров и скоростью вызова. Здравый смысл подсказывает, что длина буфера в 256 байт уже является серьезным препятствием для криптографического анализа межбитовых зависимостей.

Если говорить о практической сложности, то не менее важным является изменение этих параметров от вызова к вызову, а также отсутствие простых зависимостей как внутри входного/выходного буфера, так и между ними.

Для соблюдения указанных условий параметры можно шифровать. Приведем несколько примеров.

Пример. Использование аппаратного алгоритма AES128

Пирожком с первой полочки является шифрование передаваемого буфера на аппаратном алгоритме ключа (AES128). Однако такой подход практически неприменим (кроме случая, когда требуется обеспечить конфиденциальность параметров при их передаче на расстояние, а принимающая сторона будет также расшифровывать их при помощи электронного ключа).

Проблема здесь заключается в следующем:

- Попытка расшифровать результат выполнения загружаемого кода в приложении при помощи **GrdTransform()**, может быть обнаружена злоумышленником, что даст ему информацию о структуре буфера ввода/вывода;
- Встраивание в приложение исходного кода AES128 потенциально могло бы решить проблему, но чаще всего существенного эффекта достигнуть не удастся, так как большинство известных реализаций AES обнаруживается в приложении по сигнатурам, а вероятность, что разработчик напишет собственную, пренебрежительно мала. В то же время этого и не требуется.

Это одна из тех редких ситуаций, когда реализация собственного, пусть слабого с точки зрения криптографии, алгоритма шифрования лучше использования существующего.

Пример. Математические операции

Для видоизменения параметров можно выполнять обратимые математические операции. Операции могут быть выбраны любые. Однако лучше всего подойдут те же, что уже используются в алгоритмической части приложения. Так злоумышленнику будет сложнее отделить в вызывающем коде алгоритм «расшифрования» от «полезной» функциональности (см. след. раздел).

Следует понимать, что для расшифрования данных последовательность, использованную в качестве ключа шифрования, необходимо будет получить в приложении. В связи с этим возникает необходимость получения и согласования ключей шифрования.

Ключ шифрования

Ключ шифрования должен как можно сильнее отличаться от вызова к вызову. За это отвечает в большей степени алгоритм его генерации и в меньшей – источник энтропии (случайности), используемый для получения ключевого материала.

Аппаратный генератор случайных чисел для получения ключа использовать нежелательно: результат его работы необходимо будет передать в приложение, и он совершенно не удовлетворяет требованиям к производительности (так как, вообще говоря, не предназначен для генерации подобных «одноразовых» ключей).

Для электронных ключей с таймером в качестве источника энтропии можно использовать время в ключе. Однако в этом случае необходимо дополнительно реализовать алгоритм синхронизации времени электронного ключа и рабочей станции, а также ограничить поток запросов - не более одного в секунду, чтобы обеспечить уникальность ключа шифрования.

Одним из предпочтительных вариантов является получение ключевого материала из передаваемых параметров (к примеру, путем свертки входного буфера). Также с успехом может использоваться история вызовов (если она сохраняется внутри электронного ключа в том или ином виде).

Что касается алгоритма генерации ключей, то основным критерием его выбора является его неприметность в коде защищаемого приложения. Лучше не использовать готовый алгоритм, а реализовать его самостоятельно. Достаточно выполнения нескольких простых математических операций над ключевым материалом (преимущественно из тех, что встречаются в приложении наиболее часто).

Защита от перебора параметров

Полезной будет реализация в загружаемом коде проверки передаваемых параметров на перебор.

Для этого необходимо в том или ином виде сохранять историю вызовов загружаемого кода в ключе. Это можно сделать с использованием защищенной ячейки. Нет необходимости хранить все параметры. Достаточной может оказаться проверка только тех из них, которые меняются наиболее часто.

Простой перебор параметров (с целью анализа или построения таблицы вопросов-ответов) часто реализуется «по порядку». То есть отличия между параметрами «соседних» вызовов минимальны. Таким образом, в качестве критерия срабатывания можно использовать сохранение небольшой разницы (в одном или нескольких битах) на длинной последовательности вызовов.

В случае срабатывания защиты работу загружаемого кода можно блокировать, а информацию о его состоянии хранить в специально отведенной защищенной ячейке энергонезависимой памяти ключа.

Для электронных ключей с таймером возможен также подсчет частоты вызовов (их количества в единицу времени) и, в случае превышения порогового значения, блокировка работы ключа или внесение искусственных задержек.

Следует помнить, что любые проверки подобного рода не должны создавать помех работе легальных пользователей и, по возможности, не иметь ложных срабатываний. Для этого их логика должна быть организована с расчетом на предполагаемую частоту и характер вызовов загружаемого кода.

Защита логики вызывающего кода

После того, как злоумышленник обнаружит практическую невозможность построения табличного эмулятора, он обратится к анализу логики вызова загружаемого кода и обработки результатов его выполнения. Поэтому другим аспектом защиты загружаемого кода является защита от анализа кода, его вызывающего.

Вначале остановимся на алгоритмической части вызывающего кода, затем рассмотрим технические средства «навесной» защиты кода для различных типов приложений.

Прежде всего, код приложения не должен быть полным. Часть «полезной» нагрузки обязательно должна располагаться в ключе.

В большинстве случаев обфускация параметров (о которой мы говорили в предыдущей части), передаваемых в/из загружаемого кода обязательна. Поэтому важно позаботиться о том, чтобы алгоритм расшифрования параметров в приложении не выделялся среди прочей функциональности приложения.

В нем рекомендуется использовать те же математические операции, которые встречаются в данном участке приложения наиболее часто. По возможности, алгоритм не должен содержать «необычных» внешних вызовов (к примеру, обращения к таймеру). Аналогично и для алгоритма генерации ключей (если такой используется).

Проверки наличия ключа желательно установить в нескольких местах программы. При этом рекомендуется объединять несколько вызовов загружаемого кода в одно обращение к электронному ключу. Использование результатов его выполнения в различных участках приложения позволит как усложнить зависимость между параметрами, так и затруднить анализ логики обработки результатов для злоумышленника.

Также при организации логики проверок можно следовать рекомендациям по работе со встроенными криптографическими алгоритмами ключа.

Как только организация логики будет завершена, рекомендуется дополнительно установить «навесную» защиту на приложение.

Защита кода Native приложений

Код Native-приложений ориентирован на исполнение непосредственно в аппаратной среде. Он не содержит высокоуровневых конструкций, и для его исполнения не требуются метаданные (информация о самом коде). Ограничения на передачу управления отсутствуют.

Все это, с одной стороны, способствует развитию «навесных» защит, осуществляющих различные низкоуровневые преобразования:

- Трансляцию кода в байт-код близкой по архитектуре виртуальной машины
- Запутывание кода на уровне инструкций и блоков инструкций
- Встраивание механизмов, препятствующих отладке и анализу

С другой стороны, перечисленные свойства не позволяют осуществлять преобразования хоть сколько-нибудь более высокого уровня (к примеру, изменение графа потока исполнения). Из-за отсутствия метаданных результат подобных преобразований может оказаться неработоспособным.

Тем не менее, доступных преобразований достаточно много. Несмотря на невозможность запутывания на «высоком» уровне, в настоящее время разработано множество инструментов, которые позволяют более или менее качественно скрывать общую логику приложения за наслоением низкоуровневых преобразований.

Серьезных защит на рынке единицы. Все они похожи по своей сути и обеспечивают все базовые низкоуровневые преобразования, перечисленные выше. Основные отличия лежат в архитектурах виртуальных машин и некоторых дополнительных опциях.

Преимуществом использования Guardant Online является технология бесшовного соединения объектного файла Guardant API с кодом приложения. Это возможно только благодаря наличию «метаданных» об объектном файле Guardant API в ядре защиты.

В результате применения технологии становится крайне сложно определить точки вызовов Guardant API и определить, какие параметры передаются загружаемому коду.

Защита кода .NET приложений

.NET приложения распространяются в байт-коде на языке относительно высокого уровня.

Код таких приложений изначально ориентирован на переносимость и возможность реализации интерпретатора под любой аппаратной платформой. Язык строго типизирован, на код наложены ограничения. Все это исключает возможность воспользоваться «вольностями» аппаратно-ориентированного кода при реализации защиты.

Остаются доступными лишь некоторые методы защиты

- Символьная обфускация имен типов, событий, свойств и методов
- Высокоуровневое преобразование графа потока исполнения
- Шифрование байт-кода и отложенная компиляция

При этом исполняемый файл содержит значительное количество метаданных, которые сильно упрощают анализ. В некоторых случаях вплоть до полного восстановления участков исходного кода программы.

Среди доступных на рынке средств защиты считанные единицы используют все возможные базовые методы. Среди них Автозащита Guardant 6.0.

Заключение

В данном уроке речь шла о способах защиты загружаемого кода от анализа по принципу чёрного ящика, а также об общих принципах защиты приложений с использованием технологии загружаемого кода.

Основное преимущество новой технологии заключается в возможности «персонализации» логики электронного ключа под конкретное приложение. Это обеспечивает глубокую и взаимную интеграцию приложения и средства защиты – электронного ключа.

Технология позволяет воспользоваться всеми преимуществами «навесной» защиты и при этом сохранять свойство уникальности. Она также предоставляет редкую возможность – защищать некоторые алгоритмы от копирования и статического анализа.

К загружаемому коду предъявляются определенные требования и только при их соблюдении могут быть использованы все перечисленные преимущества. Однако в этом случае стойкость защиты приложения в целом в этом случае может быть очень высока.

Дополнительные источники информации

При возникновении вопросов, на которые вам не удалось найти ответа в этом пособии, рекомендуем обратиться к следующим дополнительным источникам информации:

WWW: http://www.guardant.ru

Web-сайт разработчика содержит большой объем справочной информации об электронных ключах Guardant.

Служба технической поддержки:

e-mail: hotline@guardant.ru
тел. +7 (495) 925-77-90