МЕТОД АЛГОРИТМИЗАЦИИ МАШИННОГО КОДА ТЕЛЕКОММУНИКАЦИОННЫХ УСТРОЙСТВ

Буйневич М.В.
Израилов К.Е.

Введение

Телекоммуникацией принято считать передачу информации по различным сетям. Для ее осуществления предназначены специальные технические средства, представляющие собой, как правило, законченные устройства на базе различных процессоров (их выбор зависит от специфичности задачи). Технические средства могут содержать программное обеспечение (ПО), реализующее алгоритмы обработки передаваемой информации. Будем называть такие средства – телекоммуникационными устройствами (ТКУ). Сферы деятельности с применением ТКУ зачастую обладают высокими требованиями к безопасности передаваемой информации. Причиной этого является высокая ценность информации (как коммерческой, так и государственной) в современном мире. В связи с этим, используемое ПО должно быть хорошо протестировано на наличие вредоносного кода. Такой код, внедренный злоумышленником сознательно (далее этот процесс будем называть – “заражением”, а само ПО – “зараженным”), может сильно снизить общую безопасность инфокоммуникационной, что приводит к актуальности задачи обнаружения факта и вида заражения. По причине отсутствия исходного кода применяемого ПО и сложности поиска уязвимостей в исполняемом коде одним из решением указанной задачи является алгоритмизация машинного кода для дальнейшего анализа специалистами “вручную”. Под алгоритмизацией в данном контексте понимается восстановление алгоритмов и данных из представления, в котором они отсутствует (то есть из машинного кода).

В статье предлагается вариант такого метода алгоритмизации (далее – Метод).

Метод алгоритмизации машинного кода

Из существующих и активно используемых на данным момент методов поиска вредоносного кода можно выделить следующие.

Самым базовым является метод ручного исследования ассемблерного представления с целью выявления и анализа подозрительных частей кода. Хотя такой метод наиболее приближен к реально-выполняемым процессам ПО, основными и критичными недостатком его применения являются: недопустимо большой объем ассемблерного листингов, сложности в восстановлении алгоритмов кода вручную и сильная зависимость от квалификации оператора с точки зрения знания конкретного процессора выполнения. Широко развитые средства дизассемблирования кода, применяемые в методе, не позволяют устранить эти недостатки.

В качестве развития метода ручного анализа часто применяется метод автоматического поиска вредоносного кода в исполняемом. Типичным примером реализации метода является антивирусное ПО, на текущий момент активно поддерживаемое и развиваемое. Однако, оно способно хорошо выявлять наличия такого код лишь по шаблонам (как кода, так и алгоритмов), не позволяя отслеживать встроенные подпрограммы, изменяющие логику выполнения, не имеющие аналогов а базах антивирусов. При этом, во-первых метод в основном применяется в сфере настольных приложений, а во-вторых он не имеет удовлетворительных выходных данных, позволяющих продолжить поиск и анализ вручную.

Одним из способов улучшения ручного анализа является метод, строящий блок-схемы алгоритмов на основании меток и переходов в ассемблерном коде. Наиболее известным примером является реализация метода в продукте IDA.

Именно данный продукт наиболее часто применяется для решения задачи поиска зараженного ПО. Метод позволяет автоматизировать часть рутинной работы ручного метода, выдавая результат в виде, интуитивно понятном человеку. Впрочем, недостатками метода в рамках текущей задачи поиска являются как неприспособленность к обнаружению именно враждебного кода, так и большой объем информации, необходимой для анализа вручную. Для примера, грубая оценки гласит, что время анализа алгоритмов работы всех функций бинарного кода среднего ПО ТКУ размером 10Мб с помощью данного метода будет занимать около 5ти человека/лет.

Отдельно можно выделить метод декомпиляции, основной задачей которого является воссоздание исходного кода ПО из его исполняемого, что может позволить произвести его анализ на высокоуровневом языке (как правило, соответствующему используемому при разработке данного ПО). Метод тесно связан с процессом обратной разработки (или "реверс-инжинирингом"), позволяющим исследовать машинный код устройства или программы с целью воссоздания его алгоритмов. К сожалению, создание полноценного декомпилятора является практически неразрешимой задачей по причине отсутствия в машинном коде информации о исходном. Поэтому, данный метод можно рассматривать только как теоретический, имеющий практическую реализацию с сильными упрощениями и ограничениями. В настоящий момент существует некоторое количество утилит, являющимися практическими реализациями метода. К ним относятся такие программы, как Boomerang, DDC, REC и IDA (с плагином восстановления исходных кодов Hey-Rays), которые однако плохо подходят для решения поставленной задачи получения алгоритмов ПО по следующим причинам. Одни из них (Boomerang, DDC) практически не развиваются, а текущая реализация является не полной. Вторые (REC) предназначены для получения не столько понятного, сколько синтаксически верного исходного кода. Третьи (IDA с плагином Hex-Ray) позволяют проводить алгоритмизацию лишь для малого коли-

чества процессоров (x86, ARM), да и они наполняют результирующий код избыточной информацией. При этом ни одна из программ изначально не “заточена” для поиска уязвимостей в ПО ТКУ.

Рассмотрим процесс создания выполняемого кода с целью перехода к новому Методу, призванному избавится от недостатков приведенных ранее.

Начальное представление программы, как правило, представляет собой исходный код на языке высокого уровня. Он является хорошо читаемым и понимаемым для разработчика ПО. Конечным представлением является бинарный код, содержащий машинные инструкции для используемого процессора. Исключением являются программы, предназначенные для выполнения на виртуальных машинах, которые в статье не рассматриваются.

Процесс получения исполняемого кода из исходного классически делится на фазы компиляции, ассемблирования и сборки. Данный процесс является однонаправленным, то есть обратное преобразование исполняемого бинарного кода в исходный в общем случае невозможно. У такого ограничения есть следующие причины:

- потеря метаданных в процессе сборки исполняемого кода [1], таких как:
  - типы и имена объектов;
  - сигнатуры функций;
  - структурированный вид алгоритмов;
  - специальные конструкции языка (например, циклы, ветвления и т.п.);
  - другая информация, используемая для удобства программирования (конструкторы/деструкторы, шаблоны и т.п.);
- оптимизация кода;
- обфускация кода.

Процесс получения исполняемого кода из исходного преобразует структурный вида программы из логических элементов языка в “хаотический” набор выполняемых операций, меток и переходов между ними.

Предлагаемый Метод осуществляет обратный процесс, преобразующий код из конечного (бинарного) представления в начальное (алгоритмическое). Необходимо отметить, что целью такого преобразования является не получение точного представления программы на языке высокого уровня (что практически невозможно), а восстановление алгоритмов работы функций программы. Данное уточнение является важным отличием Метода от аналогичных, поскольку оно позволяет решать практическую задачу – алгоритмизацию, в отличии от не решаемой теоретической – воссоздание исходного кода.

Метод, минимизирующий или устраняющий недостатки альтернативных методов, приведенных выше, целесообразно разработать с учетом следующих соображений. Во-первых, использование в нем готовых (и по возможности поддерживаемые) подходов и программных решений упростит разработку в целом. Во-вторых, максимальное использование платформенной независимости расширит количество поддерживаемых процессоров при минимальных изменениях в Методе. В-третьих, применение этапа поиска внедренного кода по заданному шаблону позволит частично автоматизировать обнаружение уязвимостей. Шаблон может описывать наиболее распространенный тип уязвимости, создаваемой злоумышленником – внедрение вредоносного кода в ПО ТКУ. И, в-четвертых, специальная разработка выходного представления для описания алгоритмов работы упростит их ручной анализ. При этом, алгоритмы удобно представлять в структурированном виде, как достаточно удобном для понимания и диагностирования логических ошибок.

Опишем предлагаемый Метод, состоящий из последовательно выполняемых этапов.

Этап 1. Преобразование бинарного кода в его текстовое представление (ассемблер).

Этап 2. Разбор ассемблера и создание его внутреннего представления.

Этап 3. Преобразование внутреннего представления к структурированному виду.

Этап 4. Поиск потенциально-внедренного кода.

Этап 5. Генерация алгоритмического представления исходного кода.

Хотя входными данными первого этапа (а следовательно и Метода) является исполняемый код, его представление может быть как бинарным (набор байтов), так и текстовым (ассемблер). Для ограничения форматов входных данных, целесообразно выбрать лишь один вид – текстовое (ассемблерное) представление, поскольку преобразование из бинарного в текстовый для конкретного процессора является однозначным и выполняется большим количеством специализированных утилит – дизассемблеров. В частности, одним из лидеров данного типа утилит является продукт IDA, имеющей помимо полноценного решения задачи этапа следующие возможности:

- поддержка большого числа процессоров;
- выделение сегментов данных;
- выделение функций;
- определение библиотечных функций;
- выделение меток и переходов внутри функций;
- интерактивная работа с кодом (позволяющая корректировать процесс дизассемблирования вручную).

Таким образом, для преобразования бинарного кода на первом этапе Метода рациональным решением является не написание своего дизассемблера, а

использования достаточно мощной программы IDA, расширяющей эффективность выполнения этапа дополнительными возможностями.

Назначением второго этапа является преобразование кода из "нечеткого" ассемблерного вида в более формализованный и платформенно-независимый с последующим анализом и созданием вспомогательных структур, хранящих синтезированную информацию о коде. Такое новое представление удобно при ручном анализа и необходимо в случае применения автоматизированных средств, поскольку оно абстрактно относительно несущественных особенностей языков программирования, имеет строгую структуру, всю дополнительно собранную информацию и не зависит от особенностей процессора входного кода.

Дальнейшее преобразование кода производится на третьем этапе Метода. Если предыдущий этап оперирует представлением кода в линейном виде, то в данном представление он будет иметь структурированный вид . Код будет состоять из набора функций и вложенных блоков, не имеющих безусловных переходов между собой – то есть являться математическим графом. Представление подобно графической диаграмме Насси-Шнейдермана, получившей широкое распространение в ряде стран (и имеющий официальный стандарт DIN 66261, разработанный Немецким институтом по стандартизации [2]), что позволяет утверждать об обоснованности выбора внутреннего вида. Такой вид позволит применять к нему "математическую теорию графов" и "теоретическую информатику" на последующих этапах Метода. Одним из основных предназначений этапа, помимо задачи структурирования, является аппроксимация метаданных исходного кода, потерянных в процессе компиляции и ассемблирования. Они должны содержать информацию о таких высокоуровневых сущностях языка, как циклы, условные переходы, типы данных и их объединения в структуры. Данный этап имеет наибольшую погрешность в представлении кода относительно его реального аналога, поскольку он пытается "предугадать" начальный вид, исходя из особенностей конечного, общей теории и практики реализации

компиляторов(ассемблеров) и психологии разработчиков ПО в конкретной области применения ТКУ.

Четвертый этап содержит набор шаблонов и алгоритмов для поиска зараженного кода. Поскольку характеристики внедренного кода крайне субъективны и его формализация является отдельной трудновыполнимой задачей, то цель этапа заключается в выделении потенциально-внедренного кода для последующей перепроверки человеком. Такой подход хотя и не дает 100% гарантии в успешности поиска, но сильно сужает область для ручного анализа. Это позволяет сократить общее время выполнение поставленной задачи и упростить способ ее достижения (поскольку представление найденного кода является понятным человеку, а для его изучение не требуется высокая квалификация оператора). Тем не менее, из-за субъективности характеристик внедренного кода (как указывалось ранее), не выделенный код также необходимо подвергнуть ручному анализу. Это позволит как обнаружить не найденные на этом этапе потенциально-опасные места в коде, так и эффективнее проанализировать найденный (в следствии наличия контекста его использования). Для этих целей предназначен следующий этап.

Генерация полного алгоритмического представления исходного кода производится на пятом этап Метода. Он, используя полученную информацию о коде (включая аппроксимированные метаданные), его структурированное представление и обнаруженные потенциально-внедренные участки, создает текстовое представление кода в виде, хорошо понятному человеку. Помимо самих алгоритмов в формализованном виде (например, на специально разработанном псевдоязыке), этап выводит дополнительную информацию, которая может быть использована оператором в процессе ручного анализа (например ссылки на адреса во встроенном коде, нарушения в его структуре, потенциально опасные или разрушенные участки, работа с портами ввода-вывода, частота использования переменных/функций/ветвлений и т.п.).

Выводы

Исходя из актуальности задачи поиска вредоносного кода в ПО ТКУ, недостатках в применяемых методах и отсутствии этих недостатков в предложенном, можно сделать вывод об его востребованности для решении данной задача. А особенности его этапов (включающие работу с формализованными данными и графами, шаблоны и алгоритмы поиска внедренного кода, генерация выходного кода) приводят к целесообразности создания специализированной Утилиты – программного средства, автоматизирующего большинство этапов Метода.

Литература

1. Рассмотрение представления кода программы с позиции метаданных, ФУНДАМЕНТАЛЬНЫЕ ИССЛЕДОВАНИЯ И ИННОВАЦИИ В НАЦИОНАЛЬНЫХ ИССЛЕДОВАТЕЛЬСКИХ УНИВЕРСИТЕТАХ

2. http://wwwlrh.fh-bielefeld.de/IN_Prak/inprak6.htm