АВТОМАТИЗИРОВАННОЕ СРЕДСТВО АЛГОРИТМИЗАЦИИ МАШИННОГО КОДА ТЕЛЕКОММУНИКАЦИОННЫХ УСТРОЙСТВ

Буйневич М.В.
Израилов К.Е.

Введение

Безопасность информации напрямую связана с уязвимостями (далее по тексту – Уязвимости) в программном обеспечении (ПО), используемом в телекоммуникационных устройствах (ТКУ). И несмотря на высокие требования к обеспечению безопасности, современные методы поиска Уязвимостей являются крайне неэффективными. Один тип применяемых методов практически полностью лишает аналитика безопасности ПО ТКУ (далее по тексту – Аналитик) возможности использования своего опыта, поскольку осуществляет поиск автоматически, что приводит к поверхностному выявлению проблем. Другой тип методов требует от Аналитика выполнения рутинной работы по изучению низкоуровневого кода ПО и воссоздания алгоритмов его работы, что резко увеличивает трудоемкость и количество ошибок в процессе поиска. Рациональным решением является метод (далее по тексту – Метод), представляющий собой некий гибрид этих двух типов и использующий специализированное средство, так называемую утилиту (далее по тексту – Утилита), для преобразования кода из низкоуровневого вида в высокоуровневый, осуществляя при этом его алгоритмизацию [*Буйневич М.В., Израилов К.Е.* Метод алгоритмизации машинного кода телекоммуникационных устройств.// Телекоммуникации.– 2012.– № 12.– С. 2-6.]. Типичная схема Метода состоит из следующих этапов: дизассемблирование кода, применение Утилиты и обработка результатов ее работы Аналитиком.

Предпосылки к разработке Утилиты

Существует некоторое количество программных средств по

алгоритмизации кода, но целесообразность их использования находится под большим вопросом. Одна часть средств практически не развивается (и значит уже сейчас не поддерживает современные процессоры), а другая является скудной попыткой решения теоретической задачи по декомпиляции кода. Таким образом, разработка вышеупомянутой Утилиты является более чем оправданной, а ее непосредственное назначение, как средства автоматизированного решения практической задачи по поиску Уязвимостей в коде (с возможностью дальнейшего поиска вручную), прогнозирует крайнюю восстребованности в области информационной безопасности. При этом выходными данными Утилиты является представление кода, подходящее для анализа Аналитиком без высокого уровня квалификации в области инструкций процессора, что также является важным фактором при выборе применяемых в составе Метода средств. Изложим архитектуру и идеи, положенные в основу проектирования Утилиты.

Алгоритм работы

Для упрощения и обобщения параметров Утилиты будем считать, что на вход она получает ассемблерный листинг процессора, на котором выполняется исследуемое ПО. Приведение бинарного кода к такому виду, как и предварительный его анализ, может быть выполнено на первом этапе Метода с применением программы IDA путем дизассемблирования, предварительного анализа и генерации текста ассемблера. Скрипты, поддерживаемые программой, позволят сгенерировать метаданные для использования Утилитой. Метаданные будут содержать деление кода на функции, выделенные структуры данных, именованные адреса по сооветствующим им функциям и глобальным переменным и т.д. Причем уже на этом подготовительном этапе Аналитик (используя логику и свой опыт) может произвести корректировку метаданных и начальную структуризацию кода, тем самым улучшив корректность результатов работы Утилиты.

Алгоритм работы Утилиты схож с классическим компилятором и

содержит фазы FrontEnd, MiddleEnd и BackEnd, последовательно выполняемые и взаимодействующие друг с другом посредством передачи данных в виде внутренних представлений.

FrontEnd отвечает за разбор входных данных (в нашем случае – ассемблера) и построение внутренних структур, необходимых для работы Утилиты. По сути, данная фаза преобразует текстовое представление ассемблера во внутреннее, которое является деревом абстрактного синтаксиса входной программы. По нему строится граф передачи управления между инструкциями, описывающий операторы перехода, метки, вызовы функций и т.п. После данной фазы внутреннее представление является набором функций ассемблера в виде однонаправленного графа, узлами которого являются инструкции ассемблера, а ребрами – потоки управления между ними (ссылки на следующую инструкцию, условный и безусловный переходы). Следует отметить, что полученное представление соответствует входному ассемблеру, а значит, является плохо структурированным.

Механизм разбора входного ассемблера зависит от типа процессора исполнения, и поэтому поддержка нового типа требует отдельной реализации этой фазы. Однако внутреннее представление будет приведено к платформенно-независимому виду с помощью использования обобщенных операций и объектов, не зависящих от типа процессора, что позволит поддерживать большое количество входных ассемблеров при неизменных последующих фазах.

MiddleEnd отвечает за анализ и преобразование построенных в FrontEnd внутренних структур. Ее входная информация, как и все преобразования в дальнейшем, является платформенно-независимой. Фаза содержит основные алгоритмы для преобразования внутреннего представления с целью его приведения к структурированному виду, одновременно осуществляя поиск потенциальных уязвимостей. Основными операциями фазы по преобразованию являются выделения таких конструкций высокоуровневых языков, как циклов, условных ветвлений, переменных и специфичных для неко-

торых языков конструкций (например, прерываний цикла «break» и операторов выбора «switch», используемых в языке C). Также на основании вызова функции и использования в ней переменных, фаза может предсказать сигнатуру, то есть входные и выходные параметры функции. Дополнительные оптимизационные действия позволят модифицировать представление так, чтобы конечный его вид был более понятен человеку. Например, оптимизация лишних пересылок данных, вычисление постоянных значений переменных и удаление пустых блоков сделает выходной код более компактным, гармоничным, а значит и читаемым.

Отдельного внимания заслуживает операция поиска Уязвимостей, поскольку она является эвристической и использует абстрагированные шаблоны, созданные на основании опыта Аналитика (например, шаблоны архитектуры внедряемого кода). Сфера охвата операции плохо освещена в современной литературе и для ее разработки может понадобится развитие отдельного направления в области безопасности кода. Как правило, вся теория и практика в этой сфере (идущая в основном из антивирусных компаний и сертификационных служб), посвящена поиску низкоуровневых Уязвимостей (таких как переполнения массива, некорректная работа с памятью, использование недокументированных или ошибочно работающих функций среды) или же кода по шаблонам уже обнаруженных вирусов. Высокоуровневые же Уязвимости (например, неверное с точки зрения безопасности или архитектуры взаимодействие модулей ПО) лишены такого внимания по причине невозможности решения задачи поиска на низкоуровневом представлении кода. Необходимо сразу отметить, что алгоритм операции не дает абсолютной гарантии даже о существовании Уязвимости; он лишь выделяет подозрительные участки кода, необходимые для анализа человеком в первую очередь.

Применение операции поиска возможно только на данной фазе по следующим причинам. Во-первых, исполняемый код хорошо подходит для автоматизированного анализа, так как преобразован во внутреннее представление. Во-вторых, к этому моменту собрана вся необходимая информация о

функциях, их вызовах и алгоритмах. И, в-третьих, представление является структурированным, а значит обнаружение Уязвимостей более эффективно. Например, злоумышленное внедрение в бинарный код может снизить уровень структурированности в отличие от аналогичного внедрения в исходный, что может быть определено автоматически по формальным признакам.

После данной фазы внутреннее представление содержит набор функций, каждая из которых состоит из дерева с узлами в качестве условных переходов и ребрами в качестве непрерывных последовательностей операций. Оно является структурированным и соответствует представлению каждой функции алгоритма в блочном виде. Такой вид хранения данных аналогичен графическому представлению Насси-Шнейдермана.

Исключения в виде безусловных переходов между ребрами никак не отражаются в дереве и реализуются путем введения меток и ссылок на них, что не сильно ухудшает представление алгоритма для понимания. Также все используемые в ассемблеры регистры заменяются на аргументы функций, локальные и глобальные переменные. В случае неизменности значения регистра он будет заменен на соответствующую константу.

BackEnd отвечает за генерацию текстового читабельного представления функций и их алгоритмов. Фаза не осуществляет преобразований внутренних данных; она лишь генерирует выделенные и оптимизированные алгоритмы, используя выходное представление Утилиты.

Полученный таким образом код должен хорошо пониматься человеком. Для этого выбран синтаксис, подобный языку C, но специализированный и дополненный конструкциями для лучшего описания алгоритмов и используемых в них данных.

Причинами выбора за основу языка C является его ориентация на структурный стиль программирования, сравнительно небольшое количество ключевых слов, лаконичность записи кода и распространенность. Сгенерированный на указанном языке код и является основным результатом работы Утилиты.

Ограничения

Утилита имеет ряд ограничений в использовании, впрочем, не сильно влияющих на ее применимость. Основным ограничением является то, что восстановленный код не обязан быть синтаксически верным с точки зрения языка. Это не является отрицательным показателем, так как целью применения Утилиты является получение не исходного компилируемого кода программы, а восстановление алгоритмов в виде, подходящем для понимания человеком. Также, точная структура алгоритмов восстановленного кода может сильно отличаться от исходной, что является допустимым, так как не влияет на общее понимание работы программы. С точки зрения применения для различных платформ, доступные для поддержки процессоры должны содержать инструкции и объекты, имеющие аналоги сущностям языка C (например, x86, PowerPC, MIPS). В практике применения Утилиты платформенное ограничение не является существенным, поскольку большинство используемых в ТКУ процессоров соответствуют этому требованию.

Схема применения

Рассмотрим схему применения Утилиты в контексте Метода по поиску и анализу «зараженного» ПО ТКУ (см. рис. 1) – то есть, модифицированного злоумышленником. В случае, когда злоумышленник имеет физический доступ к бинарному коду, он может встроить программную «закладку» – скрыто внедренный в защищаемую систему код, выявление которого невозможно стандартными средствами. По запросу (например, во время профилактической проверки), ПО ТКУ подвергается анализу с помощью Метода по следующему сценарию. Бинарный код на первом этапе работы Метода дизассемблируется программой IDA и обрабатывается ее скриптами, что создает ассемблерное представление в нотации процессора Y. На втором этапе Утилита генерирует представление алгоритмов в виде, удобном для последующего ручного анализа Аналитиком. Целью подобного анализа на заключительном

третьем этапе является нахождение Уязвимостей в коде по отклонению алгоритмов его работы от заявленных в документации. Такое текстовое представления алгоритмов содержит пометки о подозрительных участках, которые могут являться, например, результатом внедрения кода злоумышленника или программными логическими ошибками; на них требуется обратить внимание в первую очередь. На рисунке 1 на подозрительный участок указывает пометка о потенциальной программной «закладке».

Пример

Рассмотрим применение Метода и разработанного прототипа Утилиты на следующем примере.

Предположим, что исходным кодом программы является функция нахождения максимального из 3-х чисел на языке C. В качестве исполняемого процессора возьмем PowerPC.

Исходное представление программы (Листинг 1) имеет следующий вид:

```
int max3(int x, int y, int z) {
   int m, n;
   if(x > y)
      m = x;
   else
      m = y;
   if(m > z)
      n = m;
   else
      n = z;
   return n;
}
```

После компиляции, получения исполняемого кода и последующего дизассемблирования посредством IDA, ассемблерное представление программы (Листинг 2) будет следующим:

```
#function 0x00000001, max3              // max3(x, y, z)
0x00000001:  max3:       // { x(%r3), y(%r4), z(%r5), m(%r6), n%(r7)
```

```
0x00000002:  cmpw %r3, %r4          //  if(x > y)
0x00000003:  ble label_1            //  {
0x00000004:  mr %r6, %r3            //    m = x;
0x00000005:  b label_2              //  }
0x00000006:  label_1:               //  else {
0x00000007:  mr %r6, %r4            //    m  = y;
0x00000008:  label_2:               //  }
0x00000009:  cmpw %r6, %r5          //  if(m > z)
0x0000000A:  ble label_3            //  {
0x0000000B:  mr %r7, %r6            //    n = m;
0x0000000C:  b label_4              //  }
0x0000000D:  label_3:               //  else {
0x0000000E:  mr %r7, %r5            //    n = z;
0x0000000F:  label_4:               //  }
0x00000010:  mr %r3, %r7            //  return n;
0x00000011:  blr                    // }
```

где #function – метаданных, созданные скриптом для программы IDA.

Примечание: комментарии в конце каждой строки написаны вручную для лучшего понимания соответствия команд процессора исходному коду и на процесс анализа не влияют.

Код на Листинге 1 является входными данными для Утилиты, и, следовательно, для фазы FrontEnd. После обработки фазой, будет создано его внутреннее представление, являющееся графом выполнения инструкций (см. рис. 2).

MiddleEnd произведет обработку графа, создание дополнительных данных и их анализ (см. рис. 3). Используя полученную информацию, фаза преобразует входной граф в дерево, описывающее алгоритм входного кода в структурированном виде – он аналогичен диаграмме Насси-Шнейдермана (см. рис. 4).

С помощью полученного дерева в фазе BackEnd будет сгенерировано следующее описание алгоритма в текстовом виде (Листинг 3):

```
max3(arg_1, arg_2, arg_3) {
   cmpw(arg_1, arg_2);
   if(<=){
      var_2 = arg_2;
```

```
    } else {
       var_2 = arg_1;
    }
    cmpw(var_2, arg_3);
    if(<=){
       ret_1 = arg_3;
    }else{
       ret_1 = var_2;
    }
    return (ret_1);
}
```

Также листинг является результатом работы Утилиты.

Проведя сравнение Листинга 1 (соответствующего исходному коду) с Листингом 3 (соответствующего восстановленному алгоритму кода) и исходя из того, что они подобны с точки зрения описания алгоритма нахождения максимального из 3-х чисел, можно сделать вывод об успешном применении Утилиты.

Для обоснования эффективности операции поиска Уязвимостей рассмотрим пример применения Утилиты к «зараженной» функции.

Предположим, что исследуемая функция «selectCryptoMethod(int type)» осуществляет выбор способа криптографии данных – программного или аппаратного, а «заражение» ее бинарного кода выполняет принудительный переход на программный способ (Листинг 4):

```
void selectCryptoMethod(int type) {
   /* после ”заражения” бинарный код следующей операции будет
   заменен безусловным переходом на блок с вызовом
   функции selectSoftwareCryptoMethod() с помощью “goto label;” */
   if (type == 1) {
      selectHardwareCryptoMethod();
   } else if(type == 2) {
      /* после ”заражения” начало блока будет иметь метку “label:;”,
      на которую осуществляется переход */
      selectSoftwareCryptoMethod();
   } else {
      error("Unknown type of crypto method");
      return;
   }
```

```
    initializeCryptoModuleMethod();
}
```

Примечание: комментарии в исходном коде добавлены вручную для описания изменений, которые будут в бинарном коде после его «заражения».

В результате применения Утилиты будет получено следующее описание алгоритма (Листинг 5):

```
void funct(int arg) {
    goto label;
    /*
        Warning!
        Suspicious structure of program.
        The code may have been modified.
    */
    {
        funct1();
    } else if (arg == 2) {
label:;
        funct2();
    } else {
        funct3("Unknown type of crypto method");
        return;
    }
    funct4();
}
```

Как видно из Листинга 5, исследуемый алгоритм осуществляет вызов функций «funct1()» и «funct2()» в соответствии с аргументом «arg». Исходя из содержания текстовой строки, передаваемой в функцию «funct3()», можно утверждать, что алгоритм отвечает за выбор способа криптографии. А операция поиска внедренного кода при помощи комментария с заголовком «Warning!» указывает о потенциальном местонахождении вредоносных инструкций.

Выводы

Тема алгоритмизации машинного кода является крайне актуальной, поскольку позволяет развивать такие области прикладного программирования, как «реверс-инжиниринг», поиск Уязвимостей в ПО и его аудит. А затрагиваемые в ней направления по изучению языка алгоритмов (описывающего логику их работы в удобном для человеческого восприятия виде) и единого процессорно-инвариантного представления (используемого для создания и обработки самих алгоритмов) позволят внести вклад и в соответствующие теоретические области.

Для решения практических задач в приведенных прикладных областях рациональным является применение утилит автоматизации. Основным отличием предложенной Утилиты от существующих является инвариантность ее алгоритмов работы и выходного представления от большого количества потенциально поддерживаемых процессоров исполнения. Также основной упор в ее работе делается не столько на корректность получаемого на выходе кода, сколько на простоту создаваемого описания алгоритмов и их понимание человеком. В Методе, использующем Утилиту, первоначальный анализ и подготовка данных могут быть полностью реализованы с помощью внешней программы IDA, что даст возможность заменить разработку одного из его этапов (а именно, дизассемблирование) на применение профессионального продукта. Отличительной особенностью Утилиты является наличие модуля поиска Уязвимостей (например, внедренного кода по специальным шаблонам и формальным признакам) в ПО ТКУ. Также предоставляемая Утилитой информация хорошо подходит для дальнейшего ручного анализа.

Исходя из успешности применения Утилиты на приведенных примерах, можно сделать вывод о ее работоспособности, а с учетом функциональных преимуществ – и о востребованности в практике решения задачи поиска Уязвимостей в ПО ТКУ.