Защита программ от дизассемблирования

С. И. Алейников, А. О. Богатов

Аннотация. Задача дизассемблирования для архитектуры x86 весьма сложна. Для ещё большего её усложнения, что важно для защиты программ от взлома, неоднократно предпринимались попытки преобразовать бинарный код, чтобы увеличить трудоёмкость дизассемблирования. Задачей данной работы является, прежде всего, сравнение известных методов обфускации бинарного кода. Мы также оценим эффективность приёмов борьбы с ними и выявим наиболее перспективные техники. В заключение мы ответим на вопрос, насколько вообще действенна обфускация бинарного кода. Предлагаются два нестандартных метода. Перекрывающиеся инструкции, если использовать их повсеместно, фактически уничтожают граф потока управления. Двойное шифрование делает невозможными малейшие изменения кода при попытке взлома.

1. Введение

Большинство программ распространяются в виде исполняемого файла. Для взлома программы с целью пиратства необходимо прежде всего перевести машинный код в ассемблерное представление. Такой процесс носит название *дизассемблирования*. Если удастся сделать неразрешимой задачу взломщика уже на этом этапе исследования программы, шансы на сохранность интеллектуальной собственности станут стопроцентными.

Трудность задачи дизассемблирования для архитектуры x86 известна давно. Она обусловлена следующими причинами:

1. отсутствует чёткое разделение данных и кода, в результате чего код можно принять за данные и наоборот;

2. не фиксирована длина инструкций, вследствие чего иногда нельзя определить, где начинается та или иная инструкция;

3. имеется косвенная адресация, поэтому мы можем даже примерно не знать адрес, на который осуществится переход в результате исполнения некоторой команды передачи управления;

4. возможна модификация кода во время исполнения;

5. наконец (хоть это и не является отличительной особенностью рассматриваемой архитектуры), главную проблему создают многочисленные ветвления кода.

Возникает вопрос: как использовать данные особенности при создании кода, трудного для дизассемблирования, и каковы шансы успешно восстановить код, подвергнутый подобным преобразованиям?

Для ответа мы прежде всего рассмотрим два базовых метода дизассемблирования, затем обсудим преобразования графа потока управления и выявим его неустойчивость к многократной обработке и частотному анализу. В последнем разделе мы предложим нестандартные способы защиты кода, наиболее важными из которых кажутся двойное шифрование и перекрытие инструкций.

2. Разновидности и цели дизассемблирования

Прежде всего следует поставить границы рассматриваемой задачи. Анализировать программы можно двумя способами.

Во-первых, статически (дизассемблирование) — анализируя программный код, записанный в исполняемом файле, физически этот файл не исполняя.

Во-вторых, динамически (отладка) — анализируется программный код во время исполнения (в пошаговом режиме или при помощи статического анализа снятого «дампа» программы).

Приёмы защиты от статического анализа полностью уязвимы при анализе динамическом (как и наоборот). Поэтому для надёжной защиты программы требуется использовать приёмы против всех типов анализа. В настоящей работе рассматривается прежде всего защита от анализа первого рода.

Основными целями дизассемблирования (как и отладки) являются понимание и/или изменение кода программы — к примеру, для обхода защиты от копирования, извлечения/изменения функциональности. Защита на уровне бинарного кода придаёт программе, даже если её исходный код подвергся обфускации, большую устойчивость ко взлому. Поэтому исследование возможностей обфускации бинарного кода действительно актуально.

3. Простейшие методы дизассемблирования

Имеются простейшие приёмы дизассемблирования бинарного кода, не дающие хорошей гарантии результата, но являющиеся интуитивно очевидными (см., например, [3]). В случае анализа методом линейного прохода инструкции дизассемблируются последовательно.

Как только алгоритм видит следующий необработанный байт, он пытается рассмотреть этот байт как начало инструкции. Если не получилось восстановить инструкцию, алгоритм переходит к следующему байту. Основной недостаток этого способа — очень часто данные трактуются как инструкции.

При использовании рекурсивного обхода одновременно с дизассемблированием инструкций строится граф потока управления. После декодирования инструкции передачи управления (ИПУ) начинается декодирование по всем возможным адресам перехода. Главный недостаток такого подхода — в том, что возникают значительные затруднения в случае динамического вычисления адреса перехода (косвенной адресации).

Приведём простой пример (взятый из [2]) работы этих двух методов на одном и том же коде. Исходный (ассемблируемый) код таков:

```
push ebp                    ;55
mov esp, ebp                ;89 e5
call 1978808 <branch fnct>  ;e8 00 00 74 11
    [junk]                  ;0a 05
cmp 0, eax                  ;3c 00
jne 8048014 <L1>            ;75 06
mov 0, eax                  ;b0 00
jmp 8048019 <L2>            ;eb 07
    [junk]                  ;0a 05
L1: mov (1740000), eax      ;a1 00 00 74 01
L2: mov ebp, esp            ;89 ec
pop ebp                     ;5d
ret                         ;c3
nop                         ;90
```

Линейный проход даёт такой результат:

```
push ebp                    ;55
mov esp, ebp                ;89 e5
call 1978808 <branch fnct>  ;e8 00 00 74 11
or 675003c, al              ;0a 05 3c 00 75 06
mov 0, eax                  ;b0 00
jmp 8048019                 ;eb 07
or 740000a1, al             ;0a 05 a1 00 00 74
adc ecx, 90c35dec(ecx)      ;01 89 ec 5d c3 90
```


Следующий результат получен методом рекурсивного обхода:

```
push ebp                    ;55
mov esp, ebp                ;89 e5
call 1978808 <branch fnct>  ;e8 00 00 74 11
or 675003c, al              ;0a 05 3c 00 75 06
mov 0, eax                  ;b0 00
jmp 8048019                 ;eb 07
    [junk]                  ;0a 05 a1 00 00 74 01
mov ebp, esp                ;89 ec
pop ebp                     ;5d
ret                         ;c3
nop                         ;90
```

Как видно, метод линейного прохода допустил больше ошибок, второй нормализовался к концу, при этом оба метода приняли первую последовательность «мусорных» байт за код.

4. Основные методы затруднения дизассемблирования

Основные методы защиты от статических атак:

1. *Использование неразличимости данных и инструкций.* Цель — заставить дизассемблер принимать данные за инструкции или наоборот. В первом случае нарушается логика программы, появляются лишние блоки неопределённого кода, становится сложнее найти строковые константы (которые находились в секции данных) и отследить ссылки на них (к примеру, при поиске по ключевым словам ссылок на вызов диалога). Во втором случае часть кода остаётся вообще не обработанной дизассемблером. Методы, используемые для достижения этого эффекта:
   - *вставка «мусора»* — вставка «лишних» байт в недостижимые участки кода так, чтобы они казались достижимыми и были расшифрованы инструкции, следующие за ними;
   - вставка переходов на данные / (текстовых) ссылок на код.
2. *Вставка «непроницаемых предикатов»* для превращения безусловных переходов в условные. Этот приём подходит для создания никогда не исполняемых веток кода, изменения порядка дизассемблирования последующего кода и запутывания логики программы (требуется понять, что в этом условии управление передаётся всегда по одной ветке).

3. *Подмена адресов возврата функций* — используется для усложнения или полного исключения возможности вычислить статически все переходы. Обычно используется несколько техник:

   - *динамическое вычисление адреса перехода* — переход выполняется командой вида `jmp [edi]`;
   - *подмена адреса возврата в стеке* в самой функции: при переходе по команде `call <ptr>` в стек помещается адрес следующей команды, и, при возврате через `ret`, переход происходит именно на этот адрес. Командой вида `mov [esp-4], <new_ret_addr>` этот адрес можно изменить.

4. Наиболее эффективный метод против статического анализа — это *шифрование всего программного кода*. Совмещенный с антиотладочными приёмами для дешифратора, он даёт самые эффективные результаты. Основой для этого метода является возможность изменения программного кода во время исполнения. При использовании этого приёма для анализа остаётся доступным только небольшой участок незашифрованного кода — дешифратор, а весь остальной код можно получить либо сняв «дамп» с работающей программы, либо проанализировав дешифратор и написав эмулятор его выполнения.

У большей части способов защиты есть один недостаток — их невозможно реализовать при помощи компилятора с языка высокого уровня, требуется работа на уровне ассемблерных команд.

В работе [1] рассмотрен механизм запутывания бинарного кода с использованием следующих методов: вставка «мусора», «непроницаемых предикатов», подмена адреса возврата. Получены следующие результаты: неверно дизассемблировано около 75% инструкций методом линейного прохода и примерно 40% инструкций — методом рекурсивного обхода. В обоих случаях процент ошибок достаточно велик, чтобы утверждать, что дизассемблер со своей задачей не справился.

5. Обход основных методов защиты

В статье [2] предложена следующая техника дизассемблирования запутанного (без помощи шифрования) машинного кода.

Первоначально, чтобы локализовать свои действия, разбиваем весь код на функции: ищем в бинарном файле шаблоны, напоминающие заголовки функций. Затем декодируем инструкции, считая каждый байт возможным началом инструкции (т.е. получаем множество разнообразных вариантов).

Выделяя среди полученных инструкций ИПУ, строим первый вариант графа потока управления. Выделяем внутри каждой функции базовые блоки.

После этого из пары перекрывающихся блоков удаляем один. Конфликты разрешаются по следующим правилам: прежде всего, стартовая инструкция функции включается в окончательный вариант графа потока управления. Далее, если из некоторого блока достижимы два перекрывающихся блока, то блок-родитель считается неверным и удаляется. Затем из каждой пары перекрывающихся блоков оставляем тот, который имеет больше связей. Наконец, если конфликты ещё остались, удаляем один из конфликтующих блоков случайным образом.

Заполняем пустоты (там может содержаться код, достижимый при помощи ИПУ с динамически вычисляемым адресом перехода), при этом руководствуясь соображениями правдоподобия (используя эвристики).

При необходимости мы можем эмулировать исполнение некоторого блока кода незначительной длины.

Возможны и иные способы преодоления защиты.

Для разделения кода и данных дизассемблер может использовать несколько проходов по коду, использовать косвенные признаки. Правда, стопроцентной гарантии это не даёт.

В автоматическом режиме отделение «непроницаемых предикатов» невозможно, но возможно вычислить в коде ссылке на те данные, которые дизассемблируются как код, если дизассемблер делает несколько проходов по коду.

Переходы же по динамически вычисляемым адресам невозможно отследить иначе, как в отладчике во время исполнения.

Получение расшифрованного текста программы возможно двумя путями: во-первых, снятием дампа при помощи отладчика — в этом случае приходится бороться с антиотладочными приёмами; во-вторых, в известном коммерческом дизассемблере IDA Pro можно написать макрос, эмулирующий работу дешифратора, и получить код, пригодный для дизассемблирования, без использования отладчика.

6. Нестандартные приёмы защиты

Предложим не использовавшиеся в известных нам работах приёмы защиты.

Первый способ запутывания, вытекающий из рассмотренного выше, — это *использование недокументированных/нестандартных команд*. Есть шансы, что дизассемблер, руководствуясь соображениями правдоподобия, примет такие команды за что-либо другое и ошибётся.

Главный его недостаток — это одноразовый способ. После однократного обнаружения недокументированной команды она становится известной, заносится в базу команд дизассемблера и становится документированной.

Разновидность данного подхода, пригодная для борьбы с отладкой, — использование известных ошибок известных отладчиков и антивирусных анализаторов. Например, в отладчике TurboDebugger можно вызвать исключение путём следующих операций со стеком: `neg esp; neg esp`.

Следующий весьма эффективный метод — это *использование перекрывающихся инструкций*. В статье [1] приведены данные о том, что авторы предпринимали попытку включить этот метод в свой арсенал запутывающих преобразований. Результаты оказались такими: эффективность крайне высока, но велика трудоёмкость (сложно построить по произвольному коду эквивалентный ему и содержащий перекрывающиеся инструкции). Тем не менее, во многих случаях можно искусственно получить перекрывание инструкций. Вот типичный пример:

```
<..some code..>
add eax, <code_of_jump_instruction>
jmp -5
```

Последовательность байт, представляющая эти инструкции, выглядит так (вставлен переход по адресу +17):

```
b8 00 00 eb 17 eb f6
```

Выполняется первый переход — передача управления на 3-й переход, а из него мы попадаем в середину 2-й инструкции. Более того, `required_instruction` обязана быть инструкцией перехода — чтобы не произошло ошибки при попытке повторного разбора перехода номер 3 (иначе он будет выполняться тоже с середины инструкции).

Основной плюс метода — в том, что известные дизассемблеры не способны интерпретировать один и тот же участок кода другим способом, уже однажды поставив ему в соответствие последовательность команд. Задача полного рассмотрения всех вариантов в случае использования перекрывания инструкций практически невыполнима.

Таким образом, можно сформулировать следующую исследовательскую задачу: построить систему эквивалентных преобразований бинарного кода, направленных на получение как можно большего числа перекрывающихся инструкций.

Приём против отладки дешифратора — использовать в качестве кода для расшифровки *динамически вычисляемый CRC кода* самого *дешифратора*. Смысл таков:

1. Считаем CRC (Cyclic Redundancy Code) первого дешифратора (можно брать различные функции от кода дешифратора, в простейшем случае — `xor` всех байт). Цель приёма — в том, чтобы невозможно было установить точку останова на первый дешифратор (на какую-либо его инструкцию) и перехватить управление. Точка останова меняет байт в том месте, где она расположена, и CRC будет посчитан неверно. (Для невозможности установки аппаратной точки останова необходимо при этом использовать аппаратные регистры контроля точек останова.)
2. При помощи получившегося CRC первым дешифратором расшифровываем второй дешифратор.
3. Второй дешифратор расшифровывает основной код (желательно, чтобы до него труднее было добраться, расшифровывать код в динамически выделяемую область памяти).

7. Заключение

Итак, для взлома бинарного кода необходимо начать с его дизассемблирования. Существует множество способов затруднить дизассемблирование программы. Однако эффект от большинства из них, приложив некоторые усилия (использовав повторные проходы, статистический анализ, эмуляцию), можно свести к минимуму. По нашему мнению, способы, базирующиеся на усилении стандартных трудностей (неразличимость кода и данных, различная длина инструкций), дают лишь временный эффект, пока взломщики не научились их обходить. Наиболее надёжной защитой нам представляется шифрование кода, в особенности двойное (и вообще многоуровневое). Перспективным и заманчивым представляется также создание перекрывающихся инструкций. Хотя автоматическим методам получения ассемблерного кода обфускация противодействует весьма эффективно, но вообще дизассемблирование — во многом творческий процесс, и эксперт, варьируя подходы и методы, способен преодолеть большинство трудностей.

Литература

[1] Cullen Linn, Saumya Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly.

[2] Christopher Kruegel, William Robertson, Fredrik Valeur and Giovanni Vigna. Static Disassembly of Obfuscated Binaries.

[3] Shengying Li. A Survey on Tools for Binary Code Analysis. August 2004.

[4] `http://www.wasm.ru/`