Эссе

«Методы защиты байт-кода Java и .Net».

Студента 912 гр. Бабокина Д.Ю.

Москва, 2003г.

1. Введение.

Традиционно защите кода программ от декомпиляции уделяется большое внимаение, т.к. зачастую исходные тексты программ составляют коммерческую тайну. Кроме того, знание внутреннего устройства некоторых систем может очень сильно помочь злоумышленникам организовать атаку на пользователей этой системы, используя при этом ошибки, допущенные при создании продукта. Так исходные тексты операционных систем Microsoft являются, пожалуй, самым хорошо охраняемым секретом корпорации.

Существует множество способов затруднить получение исходные текстов из бинарного файла, но строгой теории обсфукации до сих пор не существует. Методы защиты носят в основном эмпирический характер. Тем не менее, существуют попытки разработки методов, которые оперались бы на строгие теоретические основы и задача восстановления исходного кода после применения которых была бы вычислительно сложной задачей. [2]

Дело ещё более усложняется при необходимости защитить программы, написанные на языке Java или для платформы .Net. Проблема в том, что в этом случае код распространяется не в виде бинарных файлов, содержащих машинные комманды целефой платформы, а в виде так называемого байт-кода, который несравненно проще анализировать. Важной особенностью этих платформ, которая непосредственно влияет на сложность анализа, является так называемая управляемость кода. На деле это означает отсутствие арифметики указателей. Кроме того, в коде производимом стандартными компиляторами сохраняется очень много символьной информации вплоть до названия классов, полей, методов, имён локальных переменных. В итоге, существует достаточно большое количество декомпиляторов, которые позволяют получить практически оригинал исходных текстов программы. В первую очередь это касается языка Java (по причине большего времени присутствия на рынке), но существует ряд продуктов и для анализа программ для платформы .Net.

Программы, осуществляющие модификацию байт-кода с целью усложнения его анализа и декомпиляции называются *обфускаторами* (англ. obfuscator). Далее пойдёт речь как раз о таких программах и методах их работы.

Но прежде чем приступить к рассмотрению обфускаторов заметим, что не существует абсолютно надёжного способа защитить программы от декомпиляции. И главная цель работы обфускаторов состоит в том, чтобы максимально затруднить понимание логики работы программы и восстановление исходных текстов и, возможно, отбить желание этим заниматься у потенциального взломщика.

2. Обзор методов работы обфускаторов.

Прежде чем начать рассмотрение методов работы обфускаторов, имеет смысл кратко ознакомиться с инструментарными средствами, находящимися в руках у потенциально взломщика, пытающегося анализоровать программу. Здесь стоит провести границу между Java и .Net. Не смотря на то, что общие принципы исполнения кода виртуальной машиной у обоих платформ одинаковы и есть много общего в подходах к анализу кода для этих платформ, в реальной жизни ситуация складывается таким образом, что инструментарий и возможности для анализа Java программ несколько богаче. В первую очередь, это связано с тем, что платформа .Net достаточно молода и не так долго присутствует на рынке, как Java. Во-вторых, байт-код .Net несколько более сложен и изначально был спроектирован как промежуточный язык для компиляции из нескольких языков высокого уровня, в то время как в Java байт-код в подавляющем большинстве случаев компилируются программы только с языка Java. Чтобы стало понятно, каким образом это может влиять на сложность создания и качество работы декомпиляторов, стоит заметить, что подобные программы зачастую знают о шаблонах по которым транслируется некоторые конструкции языка высокого уровня в байт-код. Так, например, декомпиляторы узнают «в лицо» циклы for, while и подобные конструкции. В итоге,

даже после работы достаточно простого оптимизатора, изменяющего порядок следования линейных участков и выполняющих некоторые преобразования графа потока управления, декомпиляторы могут показывать гораздо более плохие результаты. Декомпилированная программа начинает содержать большое количество комманд байт-кода для которых не было найдено соответствующего выражения в конструкциях языка высокого уровня. Абсолютным лидером тут выступает команда *goto*, которая присутствует в наборе комманд байт-кода для обоих платформ, но отсутствует в Java, C# и многих других языках высокого уровня для платформы .Net.

Не представляет сложности найти в Интерене декомпиляторы и другие сдерства анализа Java программ. Соответствующие катологи подобных инструментов присутствуют в популярных катологах на www.google.com и www.yahoo.com и могут быть легко найдены с помощью соответствующих поисковых систем. Вот неполный список некоторых декомпиляторов: Mocha, WingDis, DejaVu, Jreverse Pro, Decafe Pro. Достаточно подробно рассмотрены плюсы и минусы нескольких декомпиляторов в работе [3].

При необходимости разобраться в работе большой и при том хорошо защищённой программы могут также помочь средства анализа графа потока управления, т.к. вручную анализоровать логику больших участков кода пытаясь разобраться в байт-коде представляется достаточно трудоёмкой задачей. Тут нет унивесальных средств, позоляющих мгновенно получать готовый результат, но существует несколько библиотек, которые предоставляют возможность работать с некоторым промежуточным представление логики программы. Примером может служить проект BCEL, на основе которого постороено достаточно много продуктов, работающих с Java байт-кодом. Для того, чтобы протовостоять подобным попыткам анализа в традиционных языках программирования, не использующих промежуточное представление в виде байт-кода, существует, например, метод основанный на преобразовании графа потока управления в "плоский" граф, где для передачи управления от одного базового блока к другому используется "диспетчер", который вычисляет номер следующего базового блока по номеру текущего, логическим условиям и, возможно, некоторой дополнительной информации[2]. Но есть сомнение в возможности реализации подобных методов в Java и .Net из-за отсутствия адресной арифметики.

.Net пока уступает Java в количестве аналогичных продуктов, но нет сомнения в том, что достаточно скоро ситуация изменится. Сейчас главным инструментом являются утилиты из стандартной поставки .Net Framework – *ilasm.exe* и *ildasm.exe*. Есть ещё не зависимый от Microsoft проект Anakrino. На этом инструментраий для анализа практически исчечпывается.

3. Методы работы обфускаторов.

Сразу оговорюсь, что дальше будут рассматриваться именно обфускаторы, т.е. программы, которые защищают байт-код, другие средства защиты, такие как генерация кода целевой платформы из байт-кода рассматриваться не будут, хотя тоже имеют своё место под солнцем.

Главный приём, которые используют обфускаторы – это удаление излишней символьной информации из Java класс-файлов и сборок .Net. Байт-код устроен так, что в него попадают все имена локальных переменных, методов, полей и классов. Иногда эта информация необходима. Например, при распространнении библиотеки написаной на Java необходимо оставить нетронутыми имена всех методов и классов, которые фигурируют в интерфейсах, предоставляемых библиотекой. Также нельзя изменять имена конструкторов классов и имя метода Main. Но всё-таки в большинстве случаев можно изменить символьную информацию. И здесь уже способы «запутать» (от англ. obfuscate) могут быть самыми изощрёнными. Самый простой способ, пожалуй, это простое переименование в набор бессмысленных символов. И, надо признать, обилие переменные с именем вроде Lfu7sG4mY63nGgxc6Chz действительно сбивает с толку. Можно не гнаться на изощрённостью названий переменных и просто

переименовывать в наиболее краткие имена. В этом случае будет также достигнут эффект уменьшения размера класс-файла или сборки. Рассмотрим маленький пример на C#:

```
private void CalcPayroll(SpecialList employeeGroup)
{
        while(employeeGroup.HasMore())
        {
                employee = employeeGroup.GetNext(true);
                employee.UpdateSalary();
                DistributeCheck(employee);
        }
}
```

После обфускации этот кусок кода может выглядеть, например, так:

```
private void _1(_1 _2)
{
        while(_2._1())
        {
                _1 = _2._1(true);
                _1._1();
                _1(_1);
        }
}
```

Кроме того, что для человека, который будет анализоровать этот код будет потеряна информация, которая была заключена в имени, будет также несколько сбивать с толку обилие одинаковых названий переменных и методов, которые в сущности различаются. Можно пойти дальше и применить некоторые более коварные методы – например, использовать для имён переменных нечитаемые символы или давать названия, совпадающие с некоторыми ключевыми или зарезервированными словами языка программирования высокого уровня, например, if, while, null и тому подобные. Кроме того, что это может вызвать крах недостаточно тщательно спроектированного декомпилятора, это сильно собъёт с толку неподготовленного человека.

В то время как при программировании стараются внести как можно большую ясность в код давая говорящие названия переменным, стараясь как можно лучше структрировать код, обфускаторы, можно сказать, играют на гране фола, используя самые экзотические случаи допускаемые спецификацией.

Кроме обфускации символьной информации, которая является классическим приёмом, иногда используют преобразование данных (data obfuscations):

- Может изменяться местоположение данных (data storage obfuscation). Суть такого преобразования заключается в том, чтобы перейти от естественного представления или места хранения данных к нехарактерному для данного типа переменной. Например, локальная преременная может быть сделана глобальной. Если такой переменной оказалсь индуктивная переменная, то это может вызвать определённые сложности при анализе, особенно если в теле цикла происходит вызов какого-либо метода (придётся проводить анализ метода для того, чтобы выяснить, что не просисходит изменения индуктивной переменной в теле этого метода).
- Над данными могут быть произведены некоторые кодирующие преобразования (data encoding obfuscation). Так, например, индуктивная переменная, используемая в цикле, может быть заменена: `i` можно заменить на `5*i+2` и поменять соответствующим образом условие выхода из цикла. Кроме того, под кодирующим преобразованием может пониматься разбиение одной переменной на несколько. Это, очевидно, понижает производительность, но и вместе с тем очень сильно запутывает

код. Рассмотрим пример, где происходит переход от булевских переменных к двум коротким целочисленным переменным:

До:		После:	
(1)	`bool A,B,C;`	(1')	`short a1,a2,b1,b2,c1,c2;`
(2)	`A = true;`	(2')	`a1=0; a2=1;`
(3)	`B = False;`	(3')	`b1=0; b2=0;`
(4)	`C = False;`	(4')	`c1=1; c2=1;`
(5)	`C = A & B;`	(5')	`x=AND[2*a1+a2,2*b1+b2]; c1=x/2;c2=x%2;`
(6)	`C = A & B;`	(6')	`c1=(a^a2) & (b1^b2); c2=0;`
(7)	`C = A | B;`	(7')	`x=OR[2*a1+a2,2*b1+b2]; c1=x/2;c2=x%2;`
(8)	`if (A) …;`	(8')	`x=2*a1+a2; if ((x==1)||(x==2)) …;`
(9)	`if (B) …;`	(9')	`if (b1^b2)…;`
(10)	`if (C) …;`	(10')	`if (VAL[c1,c2])…;`

Массивы AND, OR, VAL создаются обфускатором и располагаются в классе как статические члены, либо генерируются динамически.

- Статические члены могут быть заменены на вызовы методов. Такое преобразование особенно важно для строковых переменных, знание значения которых может сильно облегчить задачу восстановления исходного кода. В качестве примера рассмотрим функцию G, которая возвращает следующие значения: G(1)=”AAA”, G(2)=”BAAAA”, G(3)=G(5)=”CCB”, G(4) возвращает неиспользуемое в программе значение “XCB”, для остальных значений параметра функции, G может возвращать непредсказуемый результат или зацикливаться.

```
String G(int n) {
        int i=0,k;
        String S;
        while (1) {
                L1: if (n==1) {S[i++]=”A”; k=0; goto L6;};
                L2: if (n==2) {S[i++]=”B”; k=-2; goto L6;};
                L3: if (n==3) {S[i++]=”C”; goto L9;};
                L4: if (n==4) {S[i++]=”X”; goto L9;};
                L5: if (n==5) {S[i++]=”C”; goto L11;);
                        if (n>12) goto L1;
                L6: if (k++<=2) {S[i++]=”A”; goto L6;} else goto L8;
                L8: return S;
                L9; S[i++]=”C”; goto L10;
                L10: S[i++]=”B”; goto L8;
                L11: S[i++]=”C”; goto L12;
                L12: goto L10;
        }
}
```

Конечно же, использование одной функции для всех строк крайне не жаелательно, так как это сильно облегчает анализ. В реальной программе предпочтительнее разбить функцию G на несколько функций и встроить эти функции в граф потока управления.

- Данные могут быть перегруппированы (data aggregation obfuscation). Например, многомерный массив может быть заменён на одномерный с соответствующей корректировкой всех обращений к этому массиву. В некоторых случаях может оказаться целесообразным наоборот увеличение размерности массива. Несколько переменных могут быть объединены в одну – например два целых 32-битных числа могут быть заключены в одном 64-битном.
- Может быть изменён порядок данных в массиве (data ordering obfuscation). Например, если для сохранения i-го элемента некоторой последовательности используется i-й элемент массива, этот массив может быть перемешан путём замены обращения к i-му элементу на обращение к f(i)-му элементу.

Следующая тип обфусцирующих преобразований затрагивает граф потока управления. Надо заметить, что методы описаные ниже имеют много общего с теми действиями, которые выполняют оптимизаторы. Подтвержением этому может служить тот факт, что класс-файлы подвергшиеся оптимизации зачастую с трудом обрабатываются разного рода декомпиляторами. Итак, рассмотрим несколько преобразований графа потока управления, которые препятствуют декомпиляции и пониманию кода:

- Первая группа преобразований направлена на преобразование вычислений (computation transformations). Её можно разбить на три разновидности преобразований – маскирвка настоящего графа потока управления за вычислениями, которые не влияют на результат работы программы, трансформация графа потока управления таким образом, чтобы было невозможно напрямую сопоставить ему конструкции языка высокого уровня и увеличение связности графа потока управления для осложнения его анализа.
- Вторая группа преобразований направлена на удаление абстакций кода, присущих языкам высокого уровня. Для повышения читаемости программ часто выделяют многие куски кода в отдельные функции, стараются структурировать программы. Для нарушения такой структуризации программ используют замену вызова метода на код метода (inlining), оформление куска кода в виде отдельного метода (outlining), дублирование методов с целью сделать неочевидными цели вызовов некоторых методов (cloning) и «расслоение» кода (interleaving). «Расслоение» кода подразумевает использование одного и того же участка кода для разных целей. Заметим ещё, что дублирование методов в сочетании с механизмом виртуальных функций может сделать задачу анализа достаточно сложной. К этой же группе преобразований может быть также отнесено развёртывание и расщипление цикла.

Последний тип защиты, который используют обфускаторы можно назвать несколько нечестным, но он может доставить большое количество хлопот анализирующей стороне. Этот тип защиты полностью основан на извесных недостатках декомпиляторов – есть особенности работы декомпиляторов, которые вызывают крах декомпилятора при попытке обработать определённые файлы. Так, например, извесный декомпилятор Mocha языка Java не может обрабатывать файлы в которых встречаются в методах инструкции после комманды return.

Заключение.

В работе дано краткое описание методов обфускации, используемых во многих коммерческих и бесплатных обфускаторах языка Java и для платформы .Net, присутствующих на рынке. Задача обфускации не нова и поэтому уже успели сложиться некоторые классические методы. В основном эти методы и были рассмотрены. Несомненно, это обзор не может претендовать на полноту, так как многие коммерческие продукты используют свои собственные методы, которые не опубликованы и составляют коммерческую тайну. Кроме того, как уже упомяналось, ведутся работы по построению методов обфускации эффективность которых математически доказана и задача восстановления исходного текста программы составляет NP-полную задачу. Эти методы не были рассмотрены, так как он находятся только в стадии разработки.

Литература.

1. A Taxonomy of Obfuscating Transformations, http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborsonLow97a/index.html
2. Wang C. Hill, J. Knights, J. Davidson, Software Tamper Resistance: Obstructing Static Analysis of Programs, Technical Report #12, Dep. Of Comp. Science, Univ. of Virginia, 2000.
3. Java Decompilers, http://www.andromeda.com/people/ddyer/java/decompiler-table.html.
4. http://www.ict.nsc.ru/ws/Lyap2001/2350/
5. Byte Code Engineering Library. http://bcel.sourceforge.net/
6. Hans Peter van Vliet. Mocha - The Java decompiler. http://wkweb4.cableinet.co.uk/jinja/mocha.html
7. H. Johnson, S. Chow, V. Zakharov, An approach to the obfuscation of control-flow of sequential computer programs, Accepted to the 2001 Information Security Conference (ISC'01), to appear in LNCS.