небезопасная безопасная ява

http://www.wasm.ru/print.php?article=unsjav1

The JavaTM Virtual Machine Specification
http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html

Unsafe Java I - Небезопасная жаба

1. Класс sun.misc.Unsafe
2. Структуры виртуальной машины
3. Особенности версии 1.5
4. Применение на практике
5. Бесконечный final

Как известно при разработке языка Java с самого начала делался упор на "безопасность" кода (так называемый "safe code"). Помимо всего прочего это означало отказ от указателей, работы с памятью и тому подобных низкоуровневых средств. Совсем отказаться правда не удалось, пришлось оставить лазейку, в первую очередь естественно для собственных классов. Но все, что использует Java Runtime, можем использовать и мы. В этой статье мы научимся писать небезопасный код на Яве и используем новоприобретенные знания для решения некоторых интересных проблем, которые штатными средствами Явы не решаются.

Рассматривать мы будем только Sun'овскую виртуальную машину, по двум причинам. Во-первых она применяется наиболее широко и является своеобразным эталоном. Во-вторых до IBM'овской у меня руки еще не дошли, а больше никаких (реально использующихся) я не знаю. Все приведенные ниже примеры протестированы с Java 1.4.2_11 и 1.5.0_06. Предполагается что читатель достаточно хорошо разбирается как в Яве, так и в общих принципах программирования.

1. Класс sun.misc.Unsafe

Малоизвестный класс sun.misc.Unsafe входит в комплект Sun Java Runtime начиная с первых версий. Как и все остальные классы в package sun.*, Unsafe не документирован, но имена (в большинстве своем нативных) функций, видимые при декомпиляции, говорят сами за себя. Явно присутствуют функции работы с памятью (allocateMemory, freeMemory,...), чтения и записи значений по заданному адресу(putLong, getLong,...) и некоторые более специализированные(throwException, monitorEnter,...). То есть в принципе все, что нам нужно.

Правда так просто инстанциировать Unsafe не удастся. Единственный constructor - приватный, а в getUnsafe() проверяется загрузчик вызвавшего класса и объект возвращается только если класс загружен Bootloader'ом. В противном случае получаем SecurityException.

```
public static Unsafe getUnsafe()
{
    Class class1 = Reflection.getCallerClass(2);
```

```
        if(class1.getClassLoader() != null)
            throw new SecurityException("Unsafe");
        else
            return theUnsafe;
    }
```

К счастью существует еще внутренняя переменная theUnsafe, до которой мы можем добраться с помощью Reflection. Всю черновую работу соберем в один класс (назовем его UnsafeUtil), который будем расширять по мере надобности.

```
public class UnsafeUtil {

        public static Unsafe unsafe;
        private static long fieldOffset;
        private static UnsafeUtil instance = new UnsafeUtil();

        private Object obj;

        static {
                try {
                        Field f =
Unsafe.class.getDeclaredField("theUnsafe");
                        f.setAccessible(true);

                        unsafe = (Unsafe)f.get(null);
                        fieldOffset =
unsafe.objectFieldOffset(UnsafeUtil.class.getDeclaredField("obj"));
                } catch (Exception ex) {
                        throw new RuntimeException(e);
                }
        };
}
```

Конечно можно просто внести UnsafeUtil в список загружаемых Bootloader'ом классов (указав путь в ключе -Xbootclasspath/a) и вызывать getUnsafe() в соответствии с замыслом Sun. Беда в том, что тогда все использующие UnsafeUtil классы также должны быть прописаны в bootclasspath'e (см. главу "5.3 Creation and Loading" в VM spec). Правда например package java.nio как-то ухитряется обходить это ограничение, но как именно пока не очень понятно. К тому же этот способ выходит за рамки "чистого" кода, так как требует дополнительных стартовых опций для виртуальной машины. Так что не будем мудрствовать и ограничимся чтением theUnsafe.

В первую очередь нам понадобятся естественно операции референцирования и дереференцирования, ObjectToAddress и AddressToObject соответственно.

```
public static long ObjectToAddress (Object o){
        instance.obj = o;
        return unsafe.getLong(instance, fieldOffset);
}

public static Object AddressToObject (long address){
        unsafe.putLong(instance, fieldOffset, address);
        return instance.obj;
}
```

С ними мы уже достаточно хорошо вооружены в техническом плане, не хватает только информации по внутреннему устройству Явы. Ее мы найдем в следующем разделе.

Очень похожую реализацию кстати сделал Don Schwarz (http://don.schwarz.name/index.php?p=30). Это одно из очень немногих мест, где можно найти хоть какие-то примеры работы с классом Unsafe. К сожалению Don в свое время не оценил потенциал низкоуровнего программирования в Яве и остановился, сделав всего пару робких шагов. Мы же пойдем дальше.

2. Структуры виртуальной машины

Теперь посмотрим, в каком виде виртуальная машина (версии 1.4) хранит данные в памяти. Поскольку Ява работает с классами и их инстанциями, то ими и займемся.

Инстанция:

```
instance_struct {
        0       magic   // всегда равен 1
        4       class   // указатель на структуру класса,
class_struct*
        8       ...     // Дальше идут подряд переменные инстанции(то
есть все которые не static),
        12      ...     // порядок пока не очень понятен, судя по
всему в порядке объявления и
        16      ...     // обьекты перед примитивными типами
        ...
}
```

Переменные типа double и long занимают 64 бита, остальные по 32. В памяти инстанции выравниваются по 64-битной границе, дополняются при необходимости нулями. То есть по сути мы имеем обыкновенную сишную структуру плюс указатель на ее описание.

Класс:

```
class_struct {
        0       magic           // всегда равен 1
        4       class           // class_struct*, указатель на
структуру класса более высокого уровня, зачем нужен - непонятно
        8       ...             // значение неизвестно
        12      super_count     // количество уровней наследования:
0x18 - один(наследует от Object), 0x1c - два и т.д. до восьми(0x34),
потом 0x10. У интерфейсов тоже 0x10.
        16      interface       // class_struct*, указатель на какой-
либо из интерфейсов класса (на какой именно непонятно), часто просто
0
        20      interface_list // указатель на массив с элементами
типа class_struct*, все интерфейсы класса
        24      ...
        28      ...
        32      ...
        36      ...             // указатели на структуры восьми
высших суперклассов начиная от Object и кончая this (если поместится)
        40      ...
        44      ...
        48      ...
        52      ...
        56      size            // размер инстанции класса в DWORD'ах
        60      this_class      // instance_struct*, указатель на
инстанцию java.lang.Class соответствующую данному классу
        64      access_flags    // доступ к классу как описано в VM
spec ( 0x1 - public, 0x10 - final и т.д.)
```

```
        68      ...
        ...
}
```

Здесь я привел только те куски, которые мы будем использовать в дальнейшем и в которых я более или менее уверен. На самом деле class_struct значительно длиннее и содержит кроме того указатели на функции класса, статические переменные и все остальное, что может понадобится виртуальной машине. Все эти структуры по понятным причинам нигде не документированы и разбираться надо вручную - хоть и несложно, но достаточно трудоемко. Если у кого-то есть желание помочь, буду только рад.

3. Особенности версии 1.5

С переходом на последнюю (на момент написания) версию 1.5.0_06 внутренние структуры виртуальной машины претерпели некоторые изменения. К счастью небольшие: изменился в основном порядок полей, значения остались в большинстве прежними. Структура класса выглядит теперь следующим образом:

```
class_struct_1_5 {
        0       magic           // всегда равен 1
        4       class           // class_struct*, указатель на
структуру класса более высокого уровня, зачем нужен - непонятно
        8       ...             // значение неизвестно
        12      size            // размер инстанции класса в DWORD'ах
        16      super_count     // количество уровней наследования:
0x20 - один(наследует от Object), 0x24 - два и т.д. до восьми(0x38),
потом 0x14. У интерфейсов тоже 0x14.
        20      interface       // class_struct*, указатель на какой-
либо из интерфейсов класса (на какой именно непонятно), часто просто
0
        24      interface_list // указатель на массив с элементами
типа class_struct*, все интерфейсы класса
        28      ...
        32      ...
        36      ...
        40      ...             // указатели на структуры восьми
высших суперклассов начиная от Object и кончая this (если поместится)
        44      ...
        48      ...
        52      ...
        56      ...
        60      this_class      // instance_struct*, указатель на
инстанцию java.lang.Class соответствующую данному классу
        64      ...
        68      ...             // точные значения неизвестны
        72      ...
        76      ...
        80      access_flags    // доступ к классу как описано в VM
spec ( 0x1 - public, 0x10 - final и т.д.)
        84      ...
        ...
}
```

Обратите внимание на переехавшее вперед поле size и сдвинутые по сравнению с версией 1.4 значения поля super_count. При написании кода придется учитывать подобные мелкие отличия. Поэтому будем в самом начале опрашивать версию виртуальной машины и сохранять результат в переменной

vm1_5. Интересно кстати, что версии 1.5.0_0x, x<6 используют все еще старые структуры. То есть достаточно глобальные изменения в виртуальной машине не обязательно приурочены к значительному скачку версии - сам по себе примечательный факт.

4. Применение на практике

Перейдем к практической части и попробуем приспособить теорию к делу. Для начала решим одну проблему, которая существует почти столько же сколько и сама Ява. А именно напишем функцию sizeof() для объектов. Желающие могут использовать свой любимый поисковик и посмотреть(например по Java+sizeof), сколько и каких решений предлагалось за последние годы, от использования Reflection до вычисления размера занятой памяти и деления его на количество объектов. Точного ответа при этом не давало, что интересно, ни одно. Нам же достаточно просто прочитать поле class_struct.size

```
public static long sizeOf(Object object){
    return
unsafe.getAddress(unsafe.getAddress(ObjectToAddress(object)+4)+(vm1_5
?12:56));
}
```

Функция возвращает результат в DWORD'ax, если нужен в байтах не забудьте умножить на 4. С помощью sizeOf() можно теперь копировать содержимое инстанций - так называемая "shallow copy". "Shallow" - так как в случае внутренних переменных типа Object (и от него производных) копируются естественно только указатели, а не объекты целиком.

```
public static void copyObjectShallow(Object objectSource, Object
objectDest) {
    unsafe.copyMemory(ObjectToAddress(objectSource),ObjectToAddress(o
bjectDest),sizeOf(objectSource)*4);
}
```

copyObjectShallow бывает особенно полезна если иметь дело с объектами, содержащими большое количество переменных примитивных типов. То есть когда класс используется в основном для хранения данных, как структура в С. Копировать переменные по одной (единственный штатный способ Явы) - удовольствия мало.

Как известно, приведение типов в Яве осуществляется динамически, с учетом иерархии классов. Бинарного каста (reinterpret_cast в терминах С++) Ява к сожалению не поддерживает. Заполним этот пробел.

```
public static Object reinterpret_cast(Object o, Class cl){
    unsafe.putAddress(ObjectToAddress(o)+4,unsafe.getAddress(ObjectTo
Address(cl)+8));
    return o;
}
```

reinterpret_cast возвращает указатель на объект o, приведенный к заданному через параметр cl типу. Имеет ли такое преобразование смысл, должен как всегда решать сам пользователь. Не следует только забывать, что ошибка при подобных манипуляциях с памятью почти всегда вызовет не безобидную Java Exception, а что-нибудь вроде Access Violation в виртуальной машине. Классический случай применения reinterpret_cast - когда один и тот же класс загружается два раза двумя разными ClassLoader'ами.

Если собираетесь писать многопоточную программу, не забывайте о синхронизации. Например имеет смысл добавить в определения приведенных выше функций слово synchronized. Иначе может случиться так, что разные потоки попытаются одновременно изменять структуры классов и радости тогда будет много.

Исходный код класса UnsafeUtil вместе с примерами использования отдельных его функций находится в приложении к статье.

5. Бесконечный final

Чтобы лучше оценить те практически неограниченные возможности, которые открывает перед нами манипулирование структурами классов, разберем пример посложнее.

Как известно, "final" в объявлении класса запрещает наследование от него. Например классы типов, такие как String, Integer и т.д. умышленно сделаны разработчиками языка конечными. Новички с завидным постоянством спрашивают на форумах, можно ли написать собственный класс строк, наследующий от String, и с таким же постоянством получают ответ "нельзя". И тем не менее правильный ответ - можно.

Для простоты и наглядности возьмем функцию hashCode(). Как известно хэш строк вычисляется по алгоритму

```
s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```

где n - длина строки и s[i] - i-й символ. Предположим теперь, что нас не устраивает стандартный алгоритм и мы хотим вычислять сумму не с начала строки, а с конца. Вот так:

```
s[n-1]*31^(n-1) + s[n-2]*31^(n-2) + ... + s[0]
```

То есть нужен класс, который наследует от String и имплементирует новую hashCode(). И именно эти свойства имеет класс MagicString, который можно найти в директории /string в исходниках к статье. MagicStringWthStub реализует ту же самую идею, только чуть элегантнее, например без использования Reflection. Недостатком в этом случае является необходимость написания stub'a, что впрочем можно легко автоматизировать. MagicString обходится без дополнительных классов.

Код самих классов я здесь приводить не хочу, чтобы не загромождать статью. Основная идея состоит в том, что мы вносим String в список суперклассов(смещения 24-52 + поле super_count) и подправляем поле access_flags нужным образом (убираем final). Детали реализации можно посмотреть в приложенных исходниках. Проверим теперь MagicString в действии:

```
        String s = (String)(Object)(new MagicStringWithStub("AB"));
        System.out.println("Magic String hashcode: "+s.hashCode());
        String s1 = new String("AB");
        System.out.println("Standard String hashcode:
"+s1.hashCode());
```

На выходе получим

```
Magic String hashcode: 2111
Standard String hashcode: 2081
```

Как видим, единственное небольшое неудобство состоит в том, что кастить в String приходится через Object. Понятно почему: компилятор-то о наших играх ничего не знает. В остальном мы полностью достигли цели: получили String с нестандартным хэш-кодом.

В следующей статье (при условии, что у меня дойдут руки ее написать :)) мы научимся создавать на Яве самомодифицирующийся код.

Благодарности

Quantum - за вдумчивое и терпеливое рецензирование черновых вариантов статьи. Если я не последовал каким-либо его советам, то исключительно по причине собственной лени.

Приложение

unsafe_java_1_code.zip (7 KB) - Примеры к статье

[C] Stiver

Unsafe Java II - Мутагенез земноводных

http://www.wasm.ru/article.php?article=unsafe_ii
http://www.wasm.ru/print.php?article=unsafe_ii
Первая статья была посвящена классам, объектам и общим принципам работы с памятью виртуальной машины. Сейчас пришло время сделать еще один шаг вглубь и посмотреть на важнейшие составные части классов, а именно функции. В плане практического применения мы научимся:

1. изменять байткод после загрузки
2. вызывать функции, не импортируя их

Но сначала небольшое лирическое отступление. После появления "Unsafe Java I" я получил большое количество отзывов, общий смысл которых сводился к "а нафиг все это нужно?". Поэтому пользуюсь случаем подчеркнуть, что никоим образом не призываю применять ниже описанное в повседневной работе. Случаи, требующие настолько глубокого копания в виртуальной машине Явы действительно достаточно редки. Однако они существуют. И в конце концов бывает просто приятно "перелезть через забор" :)

Примеры к этой статье, за исключением простейших, оформлены для разнообразия в виде небольшого crackme. Так что имеет наверное смысл сначала попробовать разобраться с ним собственными силами, а потом уже читать "решение". Хотя особо нетерпеливые или ленивые читатели могут конечно сделать и наоборот. Для тех новичков, которые читают по-немецки: по адресу http://www.buha.info/board/showthread.php?t=52196 можно найти подробную пошаговую (не поленился же **DarkTom**, за что ему большое спасибо) инструкцию, как исследовать этот, а значит и подобные, crackme.

Все примеры протестированы с Sun Java под Windows версии от 1.4.2_08 до 1.5.0_08 включительно. К своему стыду должен признаться, что несмотря на обещание, так пока до Linux'а и не добрался. Так что у кого есть время - может заняться. Байткод примеров скомпилирован версией 1.4.2_08, другие компиляторы могут немного отличаться, что приведет к другим смещениям в

коде. Соответственно если у кого-то примеры в собственной компиляции не работают, первым делом сравните получившийся байткод с моим. Что поделать, хотите работать на низком уровне - учитесь не доверять компилятору.

1. Где живут функции

Как всегда, первым делом нужно определить, где копать. Поэтому посмотрим снова на уже знакомые нам по первой части структуры классов и поищем в них поля, отвечающие за хранение функций и кода. Так как внутренняя жизнь виртуальной машины довольно, я бы даже сказал неоправданно, часто претерпевает более или менее мелкие именения, то придется нам рассмотреть в общей сложности три случая. Опять же, здесь я приведу только те поля, которые в дальнейшем действительно понадобятся.

Версии 1.4.x, где x>7

```
class_struct{
        ...     ...
        100     functions       // указатель на массив с элементами
типа func_struct*
        ...     ...
        128     constantpool    // указатель типа constantpool_struct*
        ...     ...
}
func_struct{
        0       magic
        4       class                   // class_struct*
        8       constantpool            // constantpool_struct*
        12      ...
        16      strsize                 // размер структуры в DWORD'ax
        18      ...
        ...     ...
        32      name_index              // индекс имени функции в
массиве констант
                                // (т.е. в структуре
constantpool_struct)
        34      sig_index               // индекс сигнатуры (описание
параметров и
                                // возвращаемого значения) в массиве
констант
        36      ...
        40      bc_length               // длина байткода в байтах
        ...     ...
        48      invocation_counter      // количество вызовов функции
        ...     ...
        56      code                    // псевдоуказатель на машинный
код
        ...     ...
        72      bytecode                // здесь начинается сам байткод
        ...     ...
}
constantpool_struct{
        ...     ...
        8       pool_size       // количество констант + 1
        12      tags            // constant_tags*
        16      pool_cache      // указатель на кэш методов, смотри
пояснения в тексте
        ...     ...
        28      constants       // начало массива констант
```

```
        ...     ...
}
constant_tags {
        ...     ...
        8       length  // длина массива
        12      array   // байтовый массив
        ....
}
```

Возможно кстати, что приведенные выше смещения справедливы также и для более старых версий, т.к. значения x<8 я просто не проверял.

Версии 1.5.x, где x<6

```
func_struct
{
        0       magic
        4       class_struct
        8       funcconst               // funcconst_struct*
        12      constantpool            // constantpool_struct*
        ...     ...
        24      strsize                 // размер структуры в DWORD'ax
        26      ...
        ...     ...
        36      invocation_counter      // количество вызовов функции
        ...     ...
        44      code                    // псевдоуказатель на машинный
код
        ...     ...
}
funcconst_struct{
        0       magic
        4       class_struct
        ...     ...
        24      strsize         // размер структуры в DWORD'ax
        ...     ...
        30      bc_length       // длина байткода в байтах
        32      name_index      // индекс имени функции в массиве
констант
                        // (т.е. в структуре constantpool_struct)
        34      sig_index       // индекс сигнатуры (описание
параметров и
                        // возвращаемого значения) в массиве констант
        36      ...
        ...     ...
        48      bytecode        // здесь начинается сам байткод
        ...     ...
}
```

Остальные структуры совпадают с версиями 1.4.x

Версии 1.5.x, где x>5

```
funcconst_struct{
        0       magic
        4       class_struct
        ...     ...
        32      strsize         // размер структуры в DWORD'ax
        ...     ...
        38      bc_length       // длина байткода в байтах
        40      name_index      // индекс имени функции в массиве
```

```
                        // констант (т.е. в структуре
constantpool_struct)
        42      sig_index       // индекс сигнатуры (описание
параметров и
                        // возвращаемого значения) в массиве констант
        44      ...
        ...     ...
        48      bytecode        // здесь начинается сам байткод
        ...     ...
}
constantpool_struct{
        ...     ...
        8       pool_size       // количество констант + 1
        12      tags            // constant_tags*
        16      pool_cache      // указатель на кэш методов, смотри
главу 3
        ...     ...
        32      constants       // начало массива констант
        ...     ...
}
```

Остальные структуры соответствуют предыдущему случаю 1.5.x с x<6

Внимательный читатель наверняка уже заметил, что я скачу "галопом по Европам". Неразобранными остались например constantpool_struct и constant_tags - интересные структуры с достаточно нетривиальным устройством, про кэш методов вообще по сути ничего не сказано, кроме того, что он есть. За бортом остались и отличия серверной VM, хотя все примеры написаны так, что будут работать и с ключом -server тоже. На самом деле я начал было писать полноценные объяснения, но статья быстро разрослась до совершенно неприличного размера и пришлось их снова убрать, переместив в следующую часть. Так что желающие непременно получить полную картину смотрят исходники и/или терпеливо ждут третьей части цикла.

2. Модификация байткода

На настоящий момент метод ClassLoader.defineClass0(...), задачей которого является физическая загрузка класса и инициализация его структур, представляет собой своеобразный переломный пункт в жизненном цикле байткода. Официально считается, что изменять загружаемый класс можно только до передачи его (в виде байтового массива) в defineClass0. Действительно, создание и редактирование классов "на лету" до загрузки - вручную или с помощью многочисленных библиотек типа BCEL - пользуется устойчивой популярностью. И при необходимости успешно отлавливаются простой заменой стандартного класса ClassLoader на свой собственный. Нас же интересует сейчас именно общий случай, то есть возможность изменения байткода в любой момент работы программы.

Как ни странно, никаких особых ухищрений с нашей стороны не потребуется. Достаточно знать где лежит код и четко представлять, в каком виде он там лежит. Сложностей с местонахождением возникнуть не должно, смотрим предыдущую главу. Придется только опять вычислять смещения в зависимости от версии Явы - Sun'овцы видимо все еще находятся в процессе творческого поиска. В отношении формата есть однако свои нюансы.

Правила ассемблирования и формат скомпилированного кода полностью описаны в VM spec, особенно обратите внимание, что все числовые значения хранятся по схеме big endian. В целом код в памяти будет таким же, как и в class файле, за одним большим исключением: ссылки на константы классов, полей и функций в constant pool'e заменяются ссылками на кэш, то есть по сути просто на порядковый номер обьекта. Сейчас разберем на наглядном примере.

В методе main класса TestSelfmod есть строчка

```
Unsafe unsafe = UnsafeUtilEx.unsafe;
```

Компилируется она в

```
0002    B20011  getstatic #0011 <sun.misc.Unsafe
crackme.UnsafeUtilEx.unsafe>
0005    4D      astore_2
```

Для удобства код отформатирован так же, как показывает его JavaBite (http://www.wasm.ru/baixado.php?mode=tool&id=284). 0x0011 - номер константы (тип Fieldref) в массиве констант. Но в памяти окажется следущее:

```
B20100
4D
```

Здесь 0x01 - индекс в кэше методов. По индексу 0x00 будет лежать метод TestSelfmod.<init>, а по индексу 0x02 - поле UnsafeUtilEx.vm1_5. То есть порядок констант в constant pool'e сохраняется и в кэше.

Итак с форматом мы тоже разобрались, можно править. Просто пишем свой код поверх старого, простенький пример смотрите в классе TestSelfmod. Полезной работы здесь всего три строчки:

```
int i = 0;
...
i += 23;
...
System.out.println(i);
```

По идее программа должна вывести 23 на консоль. Но с помощью

```
unsafe.putAddress(address+0x85, 0xB22A0184L);
```

мы подставляем в команду 0x840117 (т.е. iinc 1 23) число 42 (результат - 0x84012A, не забывайте про big endian) и именно его и получаем на выходе. Заметьте, что функция изменяет собственный код во время выполнения - классическая самомодификация. Еще одним примером является TestLoop, там вызванная функция изменяет вызывающую, чтобы выйти из бесконечного цикла (ifne заменяется на ifeq).

Также посмотрите на строки

```
unsafe.putAddress(key_func_data,
unsafe.getAddress(key_func_data) ^ 0xEA6716E2L);
unsafe.putAddress(key_func_data+4,
unsafe.getAddress(key_func_data+4) ^ 0xB21E0C9AL);
unsafe.putAddress(key_func_data+8,
unsafe.getAddress(key_func_data+8) ^ 0x7F131577L);
unsafe.putAddress(key_func_data+12,
unsafe.getAddress(key_func_data+12) ^ 0x003D86EFL);
```

в ValidatorMain. А когда поймете, что и как они делают, начинайте экспериментировать сами.

На самом деле конечно не все так безоблачно и при экспериментах придется учитывать еще как минимум два момента. Момент первый: весь код проходит через верификатор. Абсолютно весь, в том числе и тот, который добавляется динамически. Так что вписать на место кода всякий мусор не удастся, это должны быть более или менее осмысленные команды. Правила верификатора (почти все) описаны все в той же VM spec. Есть правда возможность отложить по времени некоторые проверки, сыграв на чрезмерном "уме" верификатора - его умении распознавать "мертвый" код. Для такого кода выполняются по-видимому только самые общие проверки и в то же время нам ничто не мешает с помощью свежеприобретенных умений в нужный момент послать его на выполнение, занопив например безусловный переход.

Кроме того не следует забывать, что Ява - язык полукомпилируемый. На некотором моменте (когда именно, зависит от настроек) байткод будет пропущен через настоящий компилятор и выполнятся будет уже машинный код. В связи с этим при изменении байткода обычно имеет смысл выставлять значения invocation_counter и code в ноль. Если код уже прошел через HotSpot compiler, то таким образом будет произведена принудительная перекомпиляция. Если функция правится еще до ее первого вызова, сбросом результатов компиляции можно естественно пренебречь.

3. Подмена функций

Как известно, имена всех использующихся классов и функций лежат в скомпилированном class файле открытым текстом. Поэтому первым инструментом при исследовании написанных на Яве программ и является не какой-нибудь хитрый декомпилятор, а обыкновенный grep. Присутствие java.io.* выдает работу с файлами, BigInteger.modPow может быть признаком RSA и так далее. Для разработчиков этот факт естественно неприятен. Встает вопрос, можно ли вызвать функцию так, чтобы её не было видно в импорте?

Первое что приходит в голову - использовать reflection. Самой функции тогда действительно не будет видно. Беда в том, что вместо нее появится пакет java.lang.reflect, в частности например Method.invoke, который не менее безошибочно обратит на себя внимание. Поэтому уточним задачу: нужно вызвать функцию так, чтобы вызывающий класс остался "чистым". Импортирование Unsafe естественно тоже запрещено.

Разберемся сначала с теорией. Как уже упоминалось выше, вызов функции в байткоде осуществляется по ее порядковому номеру в кэше. По этому индексу в кэше лежит в свою очередь указатель на соответствующую func_struct (строго говоря не только и не всегда, но в самые тонкости лезть не будем). Чтобы переправить функцию A на B, достаточно подменить этот самый указатель, и тогда последующие обращения к A будут на самом деле вызывать B, несмотря на то, что код останется прежним. Тот же самый алгоритм будет работать кстати и в отношении открытых переменных (полей) классов.

Допустим, мы по каким-либо причинам хотим спрятать возведение числа в степень, то есть вызов Math.pow(...). Давайте подумаем и распишем необходимые для этого действия. Во-первых нужен конечно подходяший вызов какой-нибудь функции A, которую нам не жалко подменить. С этим сложностей не предвидится - можно взять из стандарных библиотечных, можно просто

определить собственную. Значительно сложнее и интересней будет вторая половина задачи, т. е. получение указателя на Math.pow. Чтобы найти указатель в массиве class_struct.functions нужно для начала загрузить класс Math (нигде не сказано, что он уже присутствует в системе) и получить указатель на его class_struct. Так как Math не должен присутствовать в импорте, то загружать придется через Class.forName("java.lang.Math").

Все бы хорошо, но тогда в импорте окажется Class.forName. Она правда встречается на порядок чаще, чем reflection и особых подозрений вряд ли вызовет, но как известно лучше уж перестараться, чем наоборот. Поэтому вызов Class.forName мы тоже спрячем по описанной выше схеме, с той только разницей, что Class уже загружен и получить его структуру - задача тривиальная. План действий выглядит в итоге следующим образом:

1. Определение "ненужных" функций A и B
2. Получение указателя на Class.forName
3. Перенаправление B на Class.forName
4. Вызов B с параметром "java.lang.Math"
5. Получение указателя на Math.pow
6. Перенаправление A на Math.pow
7. Вызов A

Теперь проследим эти теоретические построения в коде. ValidatorMain.wktm станет у нас Class.forName, а KeyValidator.edtb сыграет почетную роль Math.pow. Вспомогательная функция ValidatorMain.atbz возьмет на себя задачу, которую мы до сих пор в наших рассуждениях ловко обошли молчанием, а именно собственно поиск нужного указателя в массиве functions. Казалось бы тривиальное действие, но дело в том, что определенного порядка в расположении функций судя по всему не существует, он меняется от версии к версии практически случайным образом и задается похоже методом среднепотолочного тыка. Посмотрите например на безобразную конструкцию в TestLoop.doSomething, где мы имеем дело всего с двумя функциями и я жестко задал их номер в массиве. Поэтому остается только громоздкий поиск по имени и сигнатуре, что и проделывает ValidatorMain.atbz.

Отсюда по шагам, сверяйтесь с кодом примеров. Получаем указатель на Class.forName:

```
        long class_struct =
unsafe.getAddress(unsafe.getAddress(this_struct+15*4)+4);
        long func_data = atbz(class_struct, 0x4E726F66, 0x25);
```

Пишем его в кэш на место ValidatorMain.wktm и загружаем Math:

```
        unsafe.putAddress(this_const_cache+33*4, func_data);
        Object obj = wktm(s);
```

С целью немного усложнить crackme строка s тоже собирается динамически, но это уже мелочи. Получаем указатель на Math.pow:

```
        long math_struct =
unsafe.getAddress(UnsafeUtilEx.ObjectToAddress(v)+8);
        long math_func_data = atbz(math_struct, 0x00776F70, 0x5);
```

Заменяем KeyValidator.edtb на Math.pow:

```
unsafe.putAddress(k_const_cache+13*4, math_func_data);
```

Теперь вызов edtb(name, i) в KeyValidator.mshv возведет name в степень i. Причем импорт класса KeyValidator остался совершенно "чистым".

4. Заключение

Не исключено, что при применение описанных приемов в "промышленном масштабе" придется учитывать ряд дополнительных, не упомянутых здесь тонкостей. Некоторое представление о них можно получить отсюда: http://www.cracklab.ru/f/index.php?action=vthread&topic=5363&forum=2&page=-1
С другой стороны большинство замечаний **bsl_zcs** носят опять же чисто академический характер: опции -XX: точно так же недокументированы, как и Unsafe, и систем скажем с -XX:CompileThreshold=2 мне еще никогда не встречалось и очень вряд ли встретятся.

До сих пор я цитировал только отдельные поля из структур виртуальной машины. В следующей, третьей, части я приведу полные (по возможности :)) структуры с описанием.

Приложения

unsafe_java_2_code.zip (8 KB) - Примеры к статье

jsr + jsr_w

Как известно, в байткоде Явы есть инструкция jsr (+ ее вариант jsr_w), которая выполняет прыжок на subroutine, то есть что-то вроде бейсиковской GOSUB. В VM Spec описывается ее применение при компилировании связки try-finally и я благополучно всю жизнь этой самой спецификации и верил. Тем более, что вроде действительно пару раз наблюдал что-то похожее, хотя особо и не приглядывался. А теперь вот решил разобраться с этой конструкциией - а конструкции-то и нет.. При компилировании эталонного кода получаю следующее:

Код:
```
void tryFinally() {
    try {
        tryItOut();
    } finally {
        wrapItUp();
    }
}
```

Спецификация:

Код:
```
0       aload_0
1       invokevirtual tryItOut()
4       jsr 14
7       return
8       astore_1
9       jsr 14
```

```
12      aload_1
13      athrow
14      astore_2
15      aload_0
16      invokevirtual wrapItUp()
19      ret 2
```

На самом деле:

Код:
```
0       aload_0
1       invokevirtual tryItOut()
4       aload_0
5       invokevirtual wrapItUp()
8       goto 18
11      astore_1
12      aload_0
13      invokevirtual wrapItUp()
16      aload_1
17      athrow
18      return
```

Проверено на компиляторе Sun всех версий 1.5.x и паре последних 1.4.2.x. То есть тихой сапой выполняется просто инлайн блока finally. Его содержимое в результате копируется n раз, где n = <количество блоков catch> + 2. А если я пять видов Exception ловлю и в finally у меня три страницы кода?? Слабо было хотя бы опцию сделать, инлайнить или нет?

Причем под это безобразие они ухитрились еще и теоретическую базу подвести, смотри например

The Costs and Benefits of Java Bytecode Subroutines - Freund (1998)
Subroutine Inlining and Bytecode Abstraction to Simplify Static and Dynamic Analysis - Biere (2005)

Которая в основном сводится к причитаниям, какие эти прыжки сложные и как с ними неудобно жить. Согласен, верифицировать их неудобно и выглядят не здорово, но это же не оправдание, чтобы упрощать себе жизнь таким образом. Ну клоуны, слов нет..

Вопросов в итоге два:

1) Это все действительно так, или я что-то важное упустил из виду? Может есть все-таки какой-нибудь способ заставить компилятор 1.5.x генерировать нормальный код?

2) Получается, что инструкции jsr и jsr_w теперь вообще не будут присутствовать на выходе компилятора? Значит можно просто проверять код на их наличие, и если найдены - вывод, что кто-то поработал руками или обфускатором?

https://jdk.dev.java.net/verifier.html
The new verifier does not allow instructions jsr and ret. These instructions are used to make subroutines for generating try/finally blocks. Instead the compiler will inline subroutine code which means the byte code in subroutines will be inserted in places where the subroutines are called.