Механизмы обеспечения безопасности в виртуальной java-машине.

На уровне языка программирования безопасность проявляется в следующем:

- Строгая проверка операций приведения типов. Не даёт обращаться к объекту одного класса как к объекту другого, что может привести к нарушению механизма защиты полей и методов через идентификаторы public, protected, private.
- Использование gc предотвращает утечки памяти и ошибки переполнения кучи, что особенно важно для мобильных приложений.
- Правильность индексов массивов проверяется runtime. Переменные нельзя использовать, пока они не проинициализированы, что предотвращает чтение содержимого памяти.
- Операции со строками выполнены на уровне языка, что предотвращает ошибки переполнения буфера
- Во время выполнения байт-кода, строго проверяется доступ к фреймам стека, что не позволяет изменять адреса возврата и т.п.
- Любая ошибка безопасности выбрасывается как exception, а не приводит к разрушению программы. Типичный пример – Null Pointer Exception

Общая структура классов java.

С момента появления языка java идея использования виртуальной машины и представления программы в виде промежуточного байт-кода стала широко известной (хотя и раньше существовали подобные по идее языки). Рассмотрим причины, повлиявшие на популярность этой техники.

Самым очевидным отличием байт-кода от обычных программ является их “внешний вид” и способ запуска. Код java-платформы представляется в виде class-фалов, в каждом из которых, как следует из названия, храниться java-класс. Точнее, там хранится информация, необходимая для создания специального объекта виртуальной машины, представляющей возможности для работы с экземплярами данного класса. Эта информация состоит из описания данных, принадлежащих каждому объекту класса, нестатические поля объекта, например, и методов их обработки – методов *класса*, принадлежащих сразу всем объектам этого класса. Рассмотрим структуру class-файла более подробно.

Class-файл	
`u4 magic`	4 байта, содержащих значения CAFEBABE
`u2 minor_version`	2 + 2 байта, определяющие версию файла. Java-машина не будет выполнять код более новый, чем она сама
`u2 major_version`	
`u2 constant_pool_count`	Здесь хранятся константы, необходимые для инициализации статических полей, а также строки, содержащие имя класса, пакета. Это некий аналог кучи в работающей системе.
`cp_info constant_pool[constant_pool_count-1]`	
`u2 access_flags`	Определяет, является ли этот класс public, final. Абстрактный ли этот класс или интерфейс.
`u2 this_class`	Номер элемента в массиве constant_pool[], содержащего блок информации о классе(структуру CONSTANT_Class_info)

`u2 super_class`	Аналогичное описание класса-родителя
`u2 interfaces_count` `u2 interfaces[interfaces_count]`	Массив, содержащий номера элементов хранилища констант(constant pool), представляющих собой структуры типа CONSTANT_Class_info с информации о всех интерфейсах данного класса
`u2 fields_count` `field_info fields[fields_count]`	Описание всех полей класса,
`u2 methods_count` `method_info methods[methods_count]`	всех методов
`u2 attributes_count` `attribute_info attributes[attributes_count]`	Набор структур типа attribute_info, содержащих информацию о классе. Сейчас используются атрибуты SourceFile и Deprecated. Такие же структуры содержатся внутри field_info и method_info

Конечно же, наиболее интересным местом с точки зрения безопасности являются сами коды методов, которые хранятся в атрибутах внутри структур method_info. Однако сначала я опишу работу загрузчика классов, через который проходят все классы, рано или поздно исполняемые java-машиной.

Загрузчик классов

Несмотря на то, что загрузчик классов не работает непосредственно с байт-кодом (для него это просто набор байтов), он является важным элементом системы безопасности. Некорректная работа загрузчика классов может стать дырой, через которую начинается атака на java-систему. Множество атак так или иначе связано с ошибками реализации загрузчика классов аплетов в браузерах разных производителей. Именно поэтому он является тщательно охраняемым объектом виртуальной машины. Кратко рассмотрим работу и реализацию загрузчика классов.

Когда вы запускаете приложение Java с помощью стандартной команды:

java Имя_главного_класса

виртуальная машина Java первым делом создает системный загрузчик, загружает с его помощью .class-файла вашего главного класса и вызывает статический метод вашего класса, соответствующий объявлению:

public static void main(String[] argv)

(или же сообщает об ошибке, не обнаружив такого метода).

Java – язык с отложенной загрузкой кода. Первоначально загружается только один класс – тот, который передан в качестве параметра утилите «java». Как только код этого класса обращается к какому-то другому классу (любым способом: вызовом конструктора, обращением к статическому методу или полю), загружается другой класс. По мере выполнения кода, загружаются всё новые и новые классы. Ни один класс не загружается до тех пор, пока в нем не возникнет реальная потребность. (Такое по ведение заложено в стандартный системный загрузчик.)

Главный класс приложения всегда загружается системным загрузчиком. А какие загрузчики будут использоваться для загрузки всех прочих классов?

В Java поддерживается понятие «текущего» загрузчика классов. Текущий загрузчик – это тот загрузчик классов (экземпляр некоторого наследника ClassLoader), который загрузил класс, код которого исполняется в данный момент. Каждый класс «помнит» загрузивший его загрузчик. Загрузчик, загрузивший некоторый класс, всегда можно узнать, вызвав метод getClassLoader:

public ClassLoader getClassLoader()

у объекта типа Class, соответствующего данному классу. Например, если мы находимся внутри некоторого метода класса MyClass, то вызов MyClass.class.getClassLoader() вернет ссылку на загрузчик, загрузивший этот класс, т.е. загрузивший тот самый байт-код, который выполняет вызов «MyClass.class.getClassLoader()». Когда возникает необходимость загрузить другой класс вследствие обращения к его конструктору, статическому методу или полю, виртуальная Java-машина автоматически обращается к текущему загрузчику классов, о котором «помнит» текущий исполняемый класс. При этом другой класс также «запоминает» этот загрузчик в качестве текущего. Иначе говоря, текущий загрузчик, загрузивший данный класс, по умолчанию наследуется всеми классами, прямо или косвенно вызываемыми из данного. Так как главный класс приложения обычно загружается системным загрузчиком, то он же используется и для загрузки всех остальных классов, необходимых приложению. В случае Java-апплета броузер загружает главный класс апплета своим собственным загрузчиком (умеющим читать классы с Web-сервера); в результате тот же самый загрузчик используется для загрузки всех вспомогательных классов апплета.

Такое на первый взгляд странное поведение загрузчика – разделение пространств имён – выражается в том, что даже один и тот же class-файл, загруженный разными загрузчиками(даже разными экземплярами одного и того же) видится системе как два совершенно разных класса. Зачем это сделано? - для предотвращения атак с подменой классов, ведущих в свою очередь к неправильному доступу к объектам (type confusion attack). Вообще, принято различать два основных вида “взлома” java-машины: нарушение доступа к типам и атака на classloader’ы.

Предположим, что в некотором важном системном классе объявлена private переменная. Если бы разделения пространства имён не было, то загрузив свой класс, имя которого совпадает с системным, злоумышленник смог бы получить доступ к этой переменной, объявив её в своей реализации класса как public. Если, например, эта переменная определяет права доступа к системным ресурсам, то взломщик полностью обходит ограничения, налагаемые “песочницей”.

Таким образом, некорректно написанный загрузчик классов является наиболее уязвимой частью системы. Существуют ещё два элемента, имеющих непосредственное отношение к безопасности. Это верификатор байт-кода и SecurityManager. Последний – высокоуровневый интерфейс к функциям управления безопасностью и редко содержит ошибки. Верификатор байт-кода напротив реализует сложный алгоритм анализа кода, способный предотвратить нарушения безопасности. На данный момент этот алгоритм хорошо отлажен в большинстве верификаторов, но если в нём всё же есть ошибки, это может привести к полному нарушению безопасности. На данный момент работа верификаторов стабильна. Посмотрим, как они работают.

Верификатор байт-кода

Чтобы понять предназначение верификатора, проведём такую аналогию. Когда мы передаём исходный код программы компилятору, он следит, чтобы текст соответствовал определённым синтаксическим правилам (например, должно совпадать количество открывающихся и закрывающихся скобок), некоторой семантике (нельзя, например, обращаться к примитивным типам как к объектам). Верификатор тоже работает со специальным языком программирования – байт-кодом. Байт-код – это ассемблер виртуальной машины. Далее он может быть переведён непосредственно в команды процессора JIT или HOTSPOT компилятором. В отличие от обычного ассемблера, байт-код отражает потребности объектно-ориентированной технологии, в частности создание объектов и обработку исключений. В связи с такой аналогией, работа верификатора очень

похожа на работу компилятора, плюс ещё один этап – моделирование потока выполнения команд. Это самый сложный и “таинственный” этап его работы. Однако, всё по порядку.

Спецификацией виртуальной java-машины определён следующий порядок проверок, выполняемых при проверке байт-кода:

1. Проверка структуры class-файла на соответствие стандартам, описанным в спецификации java-машины, версии байт-кода. На этом этапе делается вывод, безопасно ли подвергать данный файл синтаксическому разбору (parsing). Если да, то файл анализируется, и в памяти создаются структуры данных, используемые при дальнейшей проверке.
2. Этот этап связан с проверкой правильности объявления полей и методов класса. Он выполняется после процесса сборки (linking) для данного класса. В процессе сборки осуществляется разрешение символьных ссылок на суперклассы и реализуемые интерфейсы, а также инициализация статических полей. Далее проверяется объявление членов класса на соответствие специальной грамматике. Также проверяется, являются ли ссылки на хранилище констант правильными, т.е. указывают ли они на необходимые структуры данных.
3. Наиболее сложный и в то же время являющийся “изюминкой” работы этап, на котором моделируется работа кода и выясняется, безопасен ли он. Этот этап более подробно будет описан ниже.
4. В отличие от первых трёх этапов, которые происходят в тесном взаимодействии с загрузчиком классов, этот выполняется во время выполнения программы. Однако действия выполняемые на нём аналогичны шагу 2 применительно к нестатическим ссылкам. Наличие этого этапа связано с отложенной стратегией загрузки классов в виртуальной машине. Класс загружается только тогда, когда встречается ссылка на него. В это время проверяется правильность этой ссылки, а также правильность использования загруженного класса в точке кода, вызвавшей его загрузку.

Наконец, посмотрим, как работает анализ кода на третьем этапе. Основная цель анализа заключается в проверке на возможность переполнения стека виртуальной машины и на совпадение типов операндов в стеке при любом ходе выполнения программы. То есть, какие бы циклы, условные операторы, ветвления или обработчики исключений ни привели в данную точку программы, в стеке в данный момент должно оказаться одно и то же количество параметров с одними и теми же типами (Конечно, здесь более правильно говорить о сегменте (frame) стека). Для того, чтобы осуществить такой анализ, для каждой команды сохраняется состояние стека и локальных переменных в ходе каждого пути выполнения кода. Далее эта информация сравнивается и в случае несоответствия загрузка класса прерывается с возбуждением исключения VerifyError.

Итак, определившись с основными понятиями процесса проверки, выясним как же непосредственно происходит проверка. Вот реализуемый алгоритм:

Подготовка	• Локальные переменные инициализируются так, что их состояние отражает тип аргументов проверяемого метода. Если это не статический метод, то первым аргументом является ссылка this, а остальные отражают аргументы метода. • Опустошается стек. Это его начальное состояние. Это модель части стека, принадлежащей данному методу. • На первую инструкцию метода ставится флаг, обозначающий, что именно с неё надо начать проверку.
Основной цикл	• Выбираем инструкцию с установленным флагом. Если таких нет, то проверка считается успешно завершённой. Снимаем флаг и переходим к следующему шагу. • Моделируем действие выбранной на первом шаге команды на стек и локальные переменные. Если инструкция использует стек, то проверяется, хватит ли места на нём и достаточно ли на нём аргументов для выполнения инструкции. Если нет, то выбрасывается VerifyError. Если инструкция оперирует с локальными переменными, то проверяется соответствие используемых типов и, в случае если инструкция модифицирует переменные, производится соответствующее изменение модельных данных. После успешного завершения этого шага считается, что аргументы для данной инструкции полностью правильны. • Теперь определяем команды, которые могут выполниться вслед за этой. В случае обычной команды выбирается следующая за ней. Если это условная команда, то выбираются все варианты. В случае обработки исключений выбирается первая команда обработчика. В случае, если переход осуществляется за пределы метода, генерируется исключение проверки байт-кода. • Наконец, для каждой из инструкций, выбранных на предыдущем шаге выполняем стандартную последовательность действий. Если эта инструкция ещё ни разу не выполнялась, то помечаем, что её входными аргументами являются значения, вычисленные на шагах 2 и 3, ставим на неё флаг, показывающий, что её надо будет проверить. Если же инструкция уже исполнялась, то сравниваем аргументы, передаваемы в неё с шагов 2 и 3 с её текущим состоянием и в случае несовпадения генерируем исключение. Ставим на неё флаг и продолжаем обрабатывать команды с шага 3. Флаг ставится на инструкцию также в том случае, если в процессе сравнения и слияния, выполненном на этом шаге, были внесены изменения в параметры инструкции. • Повторяем цикл сначала.

Список источников

1. “The Java™ Virtual Machine Specification Second Edition”, Tim Lindholm, Frank Yellin. http://www.aw.com/cp/lindholm-yellin.html
2. "The Java Virtual Machine Specification", technical report, Sun Microsystems, Mountain View, CA, 1995; http://java.sun.com/
3. The Java Language Specification", technical report, Sun Microsystems, Mountain View, CA, 1995; http://java.sun.com/
4. “Java and Java Virtual Machine Security. Vulnerabilities and their Exploitation Techniques”, Last Stage of Delirium Research Group; http://lsd-pl.net/