java под атакой

крис касперски, no-email

долгое время java считалась _абсолютно_ безопасной, однако, лавинообразный рост открывшихся уязвимостей доказал обратное, впрочем, ничуть не помешав jav'e продолжить завоевание рынка, в результате чего сейчас наблюдается яростный шквал хакерских атак, часть которых отражается путем установки последних обновлений, часть же не может быть отражена в принципе, поскольку носит концептуальный характер, требующий радикального пересмотра архитектуры Java-машины, что быстро не делается. давайте попробуем познакомится со всеми этими атаками поближе.

введение

Производители Java-машин довольно оперативно реагируют на сообщения об уязвимостях, выпуская многочисленные заплатки, неспешно устанавливаемые пользователями на компьютеры, в результате чего шансы на успешную атаку с течением времени неуклонно уменьшаются и в какой-то момент дыра становится неактуальной.

Тем не менее, даже корпоративные пользователи обновляют свои компьютеры отнюдь не мгновенно и для реализации атаки у хакеров имеется по меньшей мере десяток часов, а то и дней (!) — срок более чем достаточный, особенно если учесть, что злоумышленники, работающие за деньги, ведут круглосуточный мониторинг всех информационных ресурсов, публикующих сообщения о безопасности, и всегда держат под рукой готовый атакующие программы, на заточку которых под конкретную дыру уходит не более часа. Помимо этого, злоумышленники активно занимаются самостоятельным поиском дыр, который обычно оказывается достаточно плодотворным.

Однако, не все дыры могут быть закрыты заплатки. В отличии от Си, подавляющее большинство ошибок которого носит характер мелкой косметической правки ("^#%! Мы опять забыли проверить длину строки перед ее копированием!"), дефекты виртуальных Java-машин намного более трансцендентальны — дыра не сосредоточенна в какой-то конкретной строчке кода, а представляет собой некоторую комбинацию свойств виртуальной машины, которая при стечении определенных обстоятельств приводит к возможности реализации атаки.

Легко понять, что возможных комбинаций у виртуальной машины _очень_ много, а условий, с которыми она вступают во взаимодействие, еще больше, поэтому ликвидация уязвимости требует огромной аналитической работы и существенной перестройки архитектуры виртуальной машины, что в свою очередь приводит к возможности появления новых дефектов. Тем более, что производители виртуальных машин тяготеют к закрытию дыр простыми шаблонными фильтрами.

Допусти, для реализации атаки необходимо задействовать свойства k, q и e. Допустим так же, что подобная комбинация свойств практически не используется в честных программах и потому производитель добавляет фильтр, блокирующий их выполнение, однако, если представить свойство k в виде прямых (или побочных) эффектов свойств m и n, то фильтр, очевидно, пропустит такую комбинацию мимо ушей. Отсюда следует важный вывод: ***даже если производитель рапортует об успешной ликвидации дыры, и даже если пользователи уже установили заплатки, это еще не значит, что дыры действительно нет. Исправляется ведь причина, а не следствие!*** Хакеры просто находят другой путь для достижения того же самого следствия, "воскрешая" дыры, о которых все забыли.

Уязвимости, завязанные на спецификацию Java-машины и набор исполняемых ей команд, вообще невозможно залатать без потери совместимости с уже существующими Java-приложениями. В некоторых случаях помогает установка нового Java-компилятора с последующей пересборкой исходных текстов проекта, которых, кстати говоря, у большинства пользователей просто нет, но даже если бы они и были — эту работу должен выполнять программист, поскольку следует заранее быть готовым к тому, что в тексты придется вносить изменения, иначе программа откажется собирается или поведет себя непредсказуемым образом.

На момент написания данной статьи в Java-машинах было обнаружено свыше трехсот дефектов, связанных с безопасностью, часть их которых давно закрыта, а часть остается актуальной и до сих пор. Рассказать о всех ошибках мы не можем, да, это, наверное, и ни нужно (достаточно зайти на www.securityfocus.com, набрать в строке запроса "java" и читать, читать и

читать). Мы же ставим перед собой совсем иную задачу: классифицировать ошибки по типам, перечислив основные объекты хакерских атак и источники угрозы.

нецензурный кастинг или обход системы типов

Жесткая типизация языка Java предотвращает множество непредумышленных ошибок программирования, связанных с присвоением одного типа другому. Защита реализована на уровне виртуальной машины, а _не_ на самом языке программирования (как, например, это сделано в Си++). Попытка совершить выход из функции, подсунув инструкции return массив или символьную строку вместо адреса возврата (типичный сценарий хакерских атаки на переполняющиеся буфера) приведет к аварийному завершению Java-приложения. На этом же держится и контроль границ массивов, и доступ к приватным полям классов, и многое другое.

Перехитрив типизацию, мы сломаем всю систему безопасности Java и сможем делать практически все, что угодно: переполнять буфера, вызывать приватные методы защищенных классов, и т. д. Естественно, это уже будут умышленные "ошибки", построенные на слабости механизмов контроля типов. Ну а мы про что говорим?! Хакерская атака есть ни что иное, как умышленный отход от спецификации, то есть, грубо говоря, сознательное преступление.

Далеко не все знают, что на уровне виртуальной машины Java-платформы защищены намного слабее, чем это следует из маркетинговой компании и популярных руководств для чайников, носорогов и прочих парнокопытных. В спецификации на JVM присутствует большое количество документированных (!) инструкций для низкоуровневого преобразования типов: **i2b**, **i2c**, **i2d**, **i2f**, **i2l**, **i2s**, **l2i**, **l2f**, **l2d**, **f2i**, **f2l**, **f2d**, **d2i**, **d2l**, **d2f** и даже пара команд для работы с классами: **checkcst**, **instanceof,** описание которых выложено на http://mrl.nyu.edu/~meyer/jvmref/ref-Java.html.

Более того, в пакете инструментария для разработчика Java-программ содержится _официальный_ код, показывающий как "правильно" преобразовать один тип к _любому_ другому (см. рис. 1). Так что в некотором смысле, это и не взлом вовсе, а документированная особенность Java.

```
.method public castMyType(Ljava/lang/Object;)LMyType;
      .limit stack 2
      .limit locals 2
          aload_1
          checkcast LMyType
          areturn
      .end method
```

Рисунок 1 байт-код функции castMyType, преобразующий один тип к любому другому

Как можно использовать эту особенность на практике? Допустим мы имеем два класса: trusted, исполняющийся в привилегированном интервале (а, значит, владеющий всеми ресурсами виртуальной машины) и untrusted, помещенный в "песочницу". Допустим так же, что класс trusted имеет ряд приватных методов, предназначенных строго для внутреннего использования и скрытых от "внешнего мира" системой типизации. На бумаге эта схема выглядит безупречной, но в действительности ее легко обойти путем создания подложного класса (назовем его spoofed) полностью идентичного данному, но только с public-атрибутами вместо private. Явное преобразование экземпляров класса trusted и spoofed позволит обращаться к защищенным методам, вызывая их не только из других классов, но даже из untrusted-кода! (см. рис. 2).

```
class trusted {
   private int value;
}
```

```
class spoofed {
   public int value;
}
```

```
spoofed svar=cast2spoofed(var);
svar.value=1;
```

Рисунок 2 использование преобразование типов для снятия "блокировки" с приватных методов trusted-класса, вызываемого из untrusted-кода

Такой беспредел происходит потому, что в JVM отсутствует runtime-проверки атрибутов полей класса при обращении к ним методами getfield/putfield, но если бы они даже и присутствовали (сжирая дополнительные процессорные такты), злоумышленнику ничего бы не стоило хакнуть атрибуты путем прямой модификации структуры класса двумя замечательными инструкциями JVM — getLong и putLong, специально предназначенными для низкоуровневого взаимодействия с памятью виртуальной Java-машины (однако, в этом случае атакующему придется учитывать версию JVM, поскольку "физическая" структура классов не остается постоянной, а подвержена существенным изменениям).

Теперь перейдем к переполняющимся буферам. Как уже было сказано выше, Java контролирует выход за границы массивов на уровне JVM и потому случайно переполнить буфер невозможно, однако, это легко сделать умышлено — достаточно всего лишь воспользоваться преобразованием типов, искусственно раздвинув границы массива. При записи в буфер JVM отслеживает лишь выход индекса за его границы, но (по соображениям производительности) не проверяет: принадлежит ли записываемая память кому-то еще. Учитывая, что экземпляры классов (они же объекты) располагаются в памяти более или менее последовательно (см. рис. 3), мы можем свободно перезаписывать атрибуты, указатели и прочие данные остальных классов (в том числе и доверенных!).

Естественно, для реализации данной атаки необходимо написать зловредное Java-приложение и забросить его на целевой компьютер. Атаковать уже установленные приложения не получится, поскольку мы не в состоянии осуществлять удаленное преобразование типов. Однако… кое-какие зацепки все-таки есть. Некоторые программисты при переносе Си-программ на Java испытывают потребность в работе с указателями на блок "неразборчивых" данных заранее неизвестной длины. Java таких фокусов не позволяет, вот и приходится выкручиваться путем явных преобразований. Фактически это означает, что программист сознательно отказывается от жесткой типизации и говорит Java-машине: "не надо проверять типы, границы массивов, etc – я и сам знаю как это делать". Таким образом, нельзя априори утверждать, что Java-приложения свободны от ошибок переполнения буферов. Пускай они встречаются намного реже, чем в Си-программах, но все-таки встречаются…

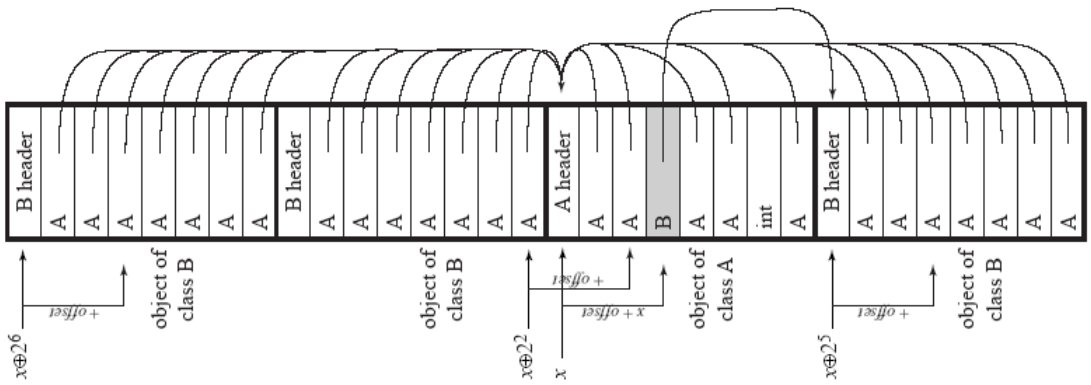

Рисунок 3 приблизительная схема размещения экземпляров объектов в памяти Java-машины

обход верификатора

Верификатор относится к одному из самых разрекламированных компонентов JVM. Весь Java-код (в том числе и добавляемый динамически) в обязательном порядке проходит через сложную систему многочисленных фильтров, озабоченных суровыми проверками на предмет корректности исполняемого кода.

В частности, если тупо попытаться забросить на вершину стека массив командой aload_1, а потом уйти в возврат из функции по ireturn (см. рис. 4), верификатор немедленно встрепенется и этот номер не пройдет.

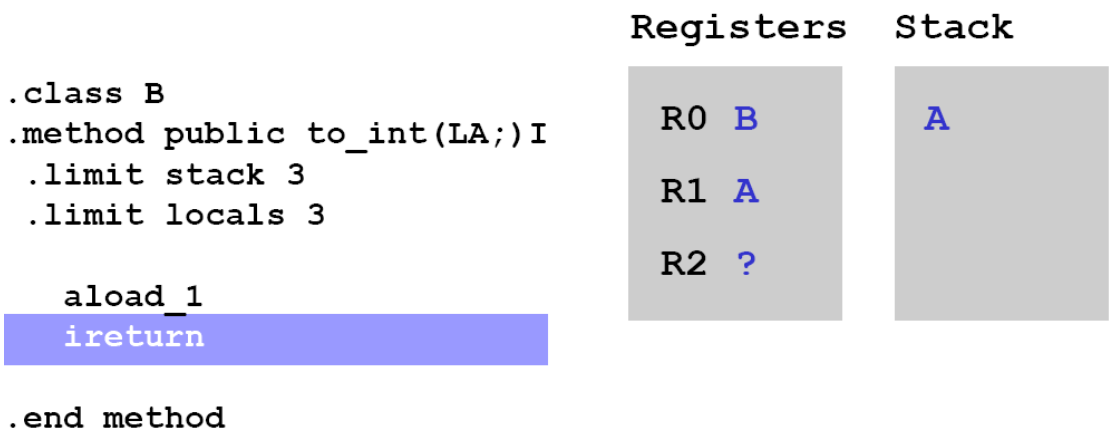

Рисунок 4 пример кода, не отвечающего требованиям верификатора (на вершине стека находится массив, в то время как ireturn ожидает увидеть целочисленный тип)

Как и любая другая достаточно сложная программа, верификатор несовершенен и подвержен ошибкам, первую из которых обнаружил сотрудник Маргбурского Университета (Германия) Карстер Зор (Karsten Sohr) в далеком 1999 году, обративший внимание на то, что верификатор начинает конкретно "буксовать" в случае, если последняя проверяемая инструкция находится внутри обработчика исключений, что позволяет, в частности, осуществлять "нелегальное" преобразование одного класса к любому другому (см. рис. 5).

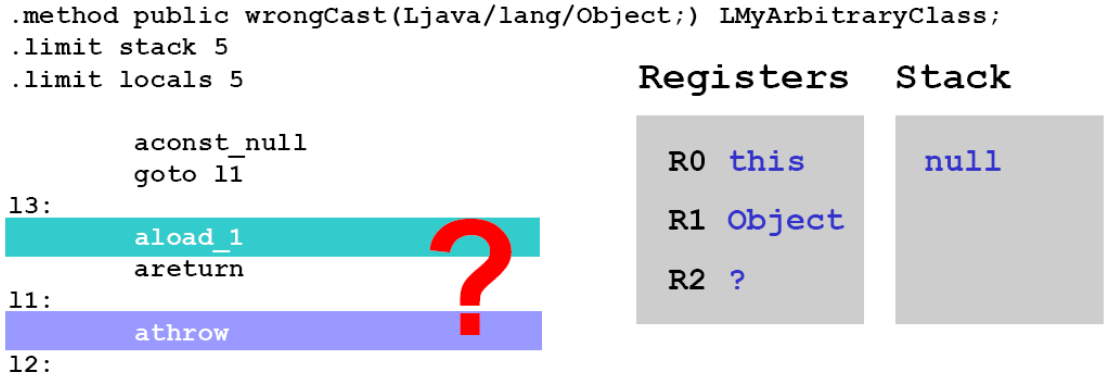

Рисунок 5 пример кода, "проскальзывающего" сквозь верификатор

Несмотря на то, что данная ошибка уже давно устранена (и сейчас представляет не более чем исторический интерес), сам факт ее наличия указывает на множество неоткрытых (а, значит, еще не исправленных) дефектов верификатора. Причем, учитывая резкое усложнение верификатора в последних версиях JVM, есть все основания полагать, что безопасность от этого не только не возросла, но даже, напротив, существенно пострадала. Ошибки в верификаторах виртуальных Java-машин обнаруживаются одна за другой — производители уже запыхались их латать, а пользователи — запарились ставить заплатки.

Другой интересный момент — атака на отказ в обслуживании. Обычно, для проверки метода, состоящего из N инструкций виртуальной машины, верификатору требуется совершить N итераций, в результате чего сложность линейно растет с размером класса и составляет O(n). Если же каждая инструкция метода взаимодействует со всеми остальными (например, через стек или еще как), то верификатору уже требуется N^2 итераций и сложность, соответственно, составляет O(N^2), что и продемонстрировано на рис. 6.

Подсунув верификатору метод, состоящий из десятков (или даже сотен!) тысяч инструкций, взаимодействующих друг с другом, мы введем ее в глубокую задумчивость, выход из которой конструктивно не предусмотрен и пользователю придется аварийно завершать работу Java-программы вместе с Java-машиной в придачу. Лекарства от данной "болезни" нет и хотя компания Sun делает некоторые шаги в этом направлении, пересматривая набор команд

JVM и выкидывая оттуда все "ненужное", сложность анализа байт кода по прежнему остается O(n^2). Особенно эта проблема актуальная для серверов, встраиваемых устройств, сотовых телефонов — там, где снятие зависшей Java-машины невозможно или сопряжено с потерей времени/данных.

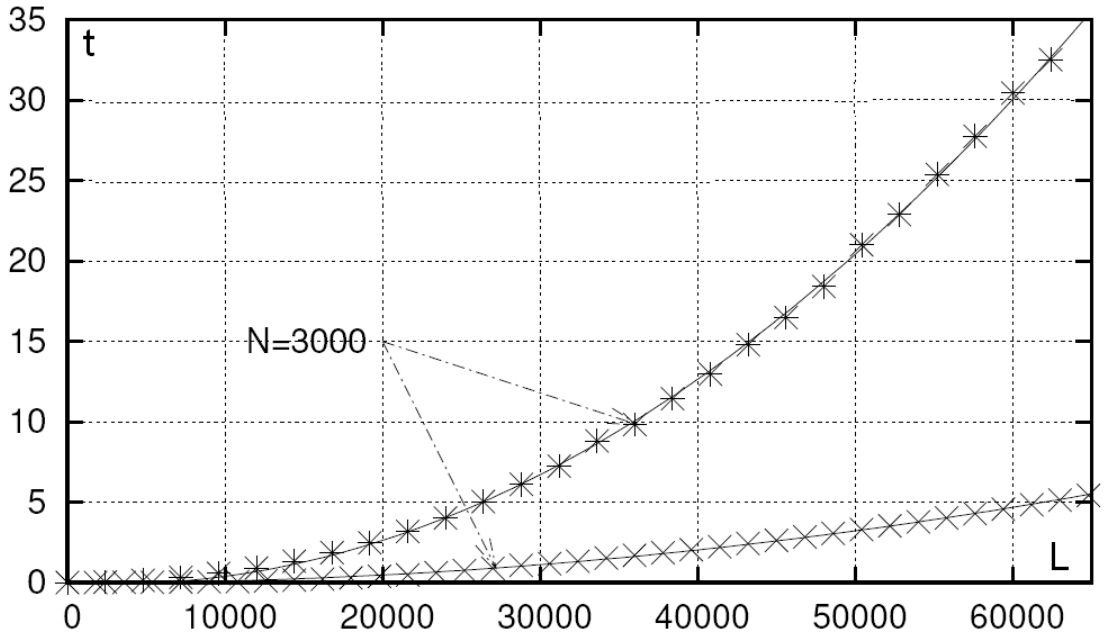

Рисунок 6 приблизительное время верификации функции с невзаимодействующими и взаимодействующими инструкциями (нижняя и верхняя кривая соотвественно), по горизонтальной оси откладываются инструкции, по вертикальной — время в секундах, измерения проводились на 2.53 GHz P4, запущенном под управлением ОС Linux и Sun HotSpot VM 1.41.

ошибки в JIT компиляторах

Современные процессоры очень быстры (прямо мустанги какие-то!), но Java-машины настолько неповоротливы, что способы выполнять только простейшие приложения, не критичные ко времени исполнения. Например, проверять корректность заполнения web-форм перед их отправкой на сервер, однако, попытки создать на Java что-то действительно серьезное, наталкиваются на неоправданно низкую производительность JVM для преодоления которой придумали JIT-компиляторы (Just-In-Time), транслирующие байт-код непосредственно в "наивный" (native) код целевого процессора, в результате чего Java-программы по скорости выполнения не сильно уступают своим аналогам на Си, но в некоторых случаях даже превосходят их (впрочем, тут все зависит от того, чей оптимизатор круче).

Откомпилированный машинный код выполняется с минимумом проверок и верификатор из динамического вырождается в статический. В частности, если произойдет переполнение буфера, то хакер без труда сможет внедрить туда shell-код и передать ему управление, захватив все привилегии виртуальной машины, достаточно часто выполняемой с правами администратора.

Отсутствие динамического анализа и скрупулезных проверок времени исполнения (их наличие сильно замедлило бы производительность) позволяет злоумышленнику сравнительно честными путями вырываться за пределы виртуальной машины, вызывая произвольные API-функции операционной системы или даже модифицируя саму (!) виртуальную машину по своему усмотрению!

К тому же JIT-компиляторы при некоторых обстоятельствах сурово ошибаются, генерируя неправильный код. Рассмотрим следующий пример (см. рис. 7), срывающий крышу Symantec JIT-компилятору, используемому, в частности, в Netscape-браузере версий 4.0-4.79 под Windows/x86.

Байт-код забрасывает на вершину стека нулевую константу (команда aconst_null), после чего вызывает локальную подпрограмму командой jsr 11, где тут же выталкивает двойное слово

с вершины стека в виртуальный регистр R1 и возвращается из нее обратно, переходя по адресу, содержащимся в виртуальном регистре R1 (а в нем как раз и лежит адрес возврата из локальной подпрограммы). Так что с точки зрения верификатора все выглядит предельно корректно и у него никаких претензий нет.

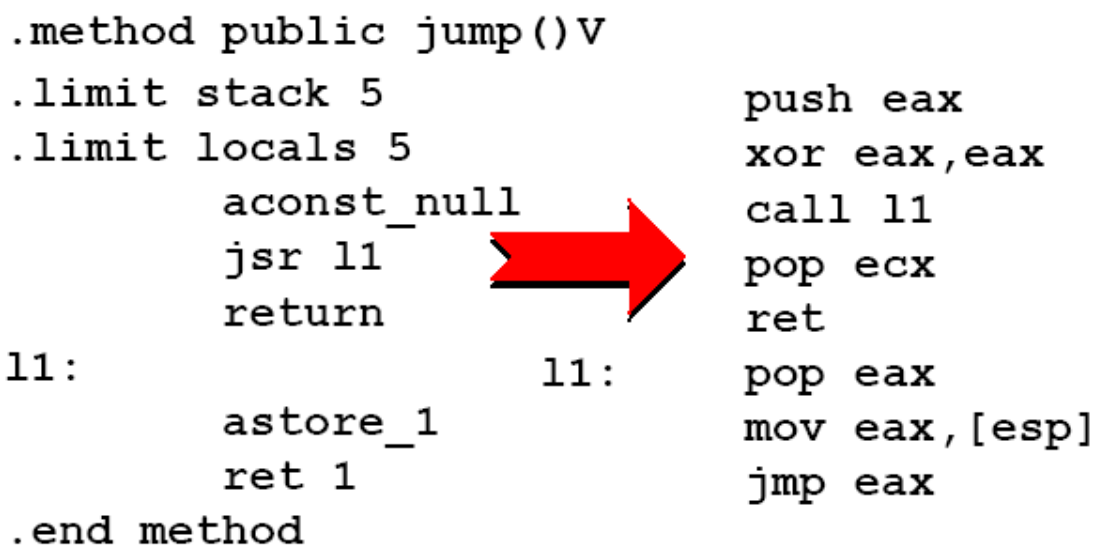

Рисунок 7 байт-код (слева) и результат его компиляции в x86-код Symantec JIT-компилятором

Что же касается JIT-компилятора, то перед входом в функции он сохраняет регистр EAX в стеке (условно соответствующий виртуальному регистру R1), далее обнуляет его (команда XOR EAX,EAX), но не кладет в стек, а прямо так в регистре и оставляет. Потом вызывает локальную подпрограмму (инструкция CALL l1), забрасывая на стек адрес возврата (т.е. адрес первой следующей за ней команды — инструкции POP ECX). В самой же подпрограмме компилятор стягивает с вершины стека двойное слово, помещая его в регистр EAX (команда POP EAX), что совершенно правильно, а вот затем, отрабатывая RET 1, он вместо того, чтобы сразу прыгнуть на JMP EAX, по совершенно непонятным причинам еще раз лезет в стек и копирует в EAX двойное слово, находящееся на его вершине (инструкция "MOV EAX, [ESP]", в результате чего реальный переход осуществляется по физическому указателю, находящемуся в регистре EAX.

Обычно там находится мусор и программа (вместе с Java-машиной) просто рушится, однако, при желании можно воздействовать на EAX, засунув в него указатель на shell-код или что-то типа того. Для этого перед вызовом функции jump() достаточно выполнить следующую последовательность команд виртуальной машины: iload_1/ireturn.

Сейчас эта дыра уже заткнута (и приведена лишь в качестве примера), но свежими уязвимости всегда можно разжиться на www.securityfocus.com

повышение собственных привилегий

Несанкционированное повышение привилегий актуально главным образом для Java-приложений, поступающий из ненадежных источников (например, из сети) и выполняемых в "песочнице" (sandbox), прорыв за пределы которой приводит к полной анархии. Злоумышленник получает возможность исполнять любой код, открывать порты, обращаться к локальным файлам и т. д.

В последних версиях JVM песочницу растащили на стройматериалы, ушедшие на создание новой системы безопасности, обеспечивающей разграничение доступа не на уровне Java-приложений (как это было раньше), а на уровне отдельных классов. Доверенные (trusted) классы могут делать все что угодно (если только не оговорено обратное). Остальные же — довольствуются обращением к публичным методам доверенных классов. Если атакующий сможет добраться до приватных (или защищенных) методов доверенного класса, его цель будет достигнута.

В верификаторе Java-машины, встроенной в MS IE версий 4.0, 5.0 и 6.0, присутствовал коварный дефект, позволяющий создавать полностью инициализированные экземпляры классов, даже при возникновении исключения в методе super().

Метод super() похож на указатель this, поддерживаемый Java/Си++, однако, в отличии от this, указывающего на экземпляр данного класса, super() вызывает конструктор суперкласса (или базового класса в терминах Си++) к которому принадлежит данный экземпляр производного класса (узнать подробнее о методах this() и super() можно на http://www.faqs.org/docs/javap/c5/s5.html).

Хорошая идея — взять доверенный класс и создать экземпляр производного класса (или sub-класса в терминах Java) и проинициализировать его вызовом super(). Тогда злоумышленник сможет вырваться за пределы песочницы (которой, как мы помним, уже нет). Пример реализации прорывного кода приведен ниже (см. рис. 8).

```
public class VerifierBug extends
com.ms.security.SecurityClassLoader {

        public VerifierBug(int i) {
         super();
        }

        public VerifierBug() {
         try {
          this(0);
         } catch (SecurityException) {}

        }
```

Рисунок 8 исходный код Java-exploit'а, пытающийся повысить собственные привилегии, но пресекаемый еще на стадии трансляции Java-компилятором

Единственная проблема с которой столкнется атакующий — Java-компилятор откажется транслировать такой код (и это, в общем-то, правильно), однако, если вручную запрограммировать зловредную программу на Java-ассемблере (в качестве которого можно взять бесплатный транслятор Жасмин — jasmin.sf.net), верификатор байт-кода примет ее как родную, поскольку Java-машина, реализованная в IE, выполняет линейный анализ кода, а с этой точки зрения код вполне нормален (см. рис. 9)

```
.class public VerifierBug
.super com/ms/security/SecurityClassLoader

.method public <init>()V
.limit stack 5
.limit locals 5
aload_0
bipush 0
l1:
invokenonvirtual VerifierBug/<init>(I)V
l2:
aconst_null
l3:
return

.catch java/lang/SecurityException from l1 to l2 using l3

.end method

.method public <init>(I)V
.limit stack 5
.limit locals 5
            aload_0
            invokenonvirtual com/ms/security/SecurityClassLoader/<init>()V
            return
.end method
```

Рисунок 9 байт-код exploit'a, повышающего свои привилегии и свободно проходящего через верификатор MS IE версий 4.0, 5.0 и 6.0

Похожие ошибки содержатся и в других виртуальных машинах. В частности, в Netscape версий 4.0 - 4.79 вообще можно обойтись без вызовов this() и super(), заменив их ветвлениями (jsr/astore/ret). Так что для исследований открыт полный простор.

дыры в runtime-библиотеках и системных классах

В конце апреля 2007 года в Apple QuickTime Player'e всех версий, вплоть до 7.1.5 обнаружилась огромная дыра, позволяющая Java-приложениям исполнять произвольный код на удаленной системе. Достаточно зайти на WEB-страничку злоумышленника и… все. Учитывая огромную распространенность Apple QuickTime Player'a с одной стороны и Microsoft Internet Explorer'a с другой произошло своеобразное перекрестное опыление, в результате чего пострадали сразу обе системы: как вся линейка Windows NT (включая Висту) и Mac OS.

Впрочем, сама Java-машина тут не при чем. Ошибка сидит во внешнем (по отношению к ней компоненте), и потому уязвимость распространяется не только на IE, но и FireFox.

```
// инициализирует Quick-Time
QTSession.open();

// получаем обработчик, указывающий на что угодно
byte b[] = new byte[1 /* здесь может быть любое число */];
QTHandle h = new QTHandle(b);

// превращаем обработчик в указатель на объект
// огромное отрицательное значение обходит проверку диапазона
QTPointerRef p = h.toQTPointer(-2000000000 /* смещение */, 10 /* размер */);

// перезаписываем объект
p.copyFromArray(0 /* смещение */, b /* источник */, 0, 1 /* длина */);
```

Листинг 1 exploit, пробивающий практически любую Java-машину, при наличии установленного Apple QuickTime Player'a с версией 7.1.5 или более ранней

Этот пример наглядно доказывает, что говорить о безопасности Java в отрыве от надежности всех остальных компонентов операционной системы и ее окружения — наивно. Java должна либо быть "вещью в себе" и не допускать никаких внешних вызовов (так вели себя некоторые диалекты Бейсика на 8-ми разрядных компьютеров, из лексикона которых были исключены операторы CALL, PEEK и POKE), либо открыто признать, что на шатком фундаменте крепости не построишь и доверять Java-приложениям (даже с учетом всей многоуровневой системы безопасности все равно нельзя).

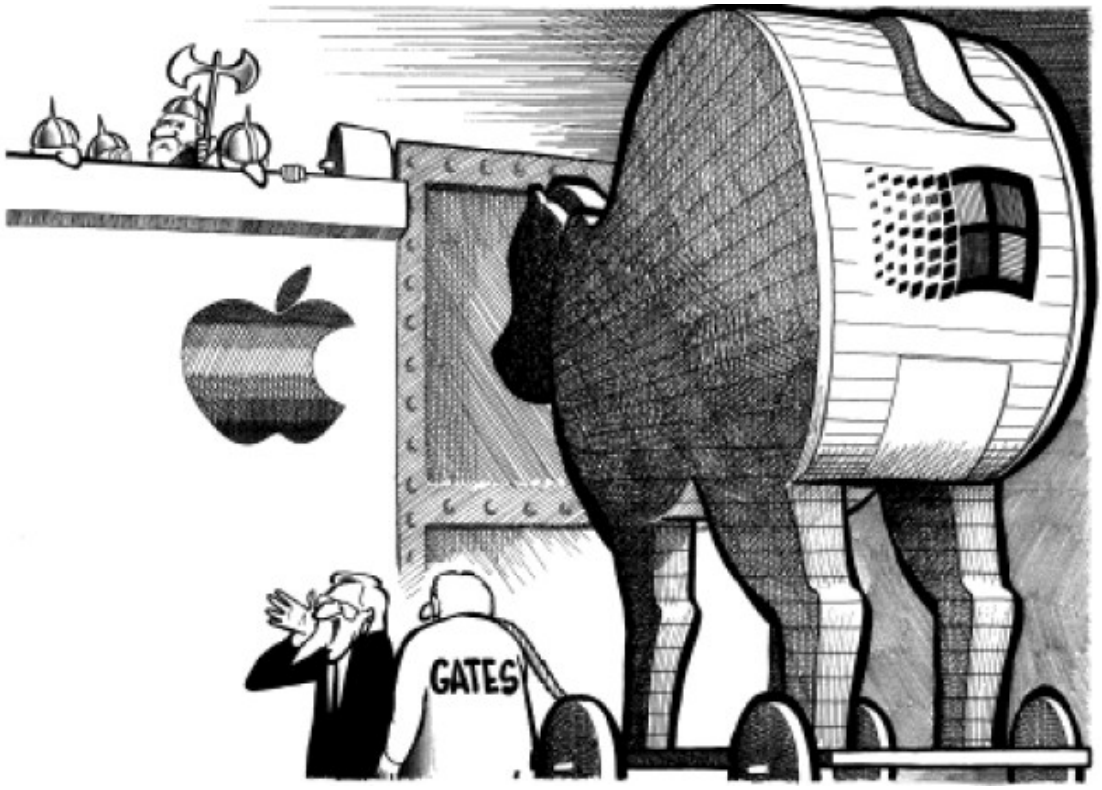

Рисунок 10 Microsoft приводит в лагерь Apple троянского коня

Впрочем, отказ от внешних вызовов ничего не решает, поскольку системные библиотеки, поставляемые вместе с Java-машиной, ничем не лучше прочих компонентов и могут содержать различные дефекты проектирования. Хотите пример? Пожалуйста! В начале 2007 года в Sun JRE 5.0 Update 9 (включая и более ранние версии) была обнаружена грандиозная ошибка, связанная с обработчиком заголовков GIF-файлов и содержащая уязвимость, приводящую к возможности передачи управления на shell-код, расположенный непосредственно в самом GIF-файле.

Технические подробности можно найти на www.securityfocus.com/archive/1/457159, а поживиться exploit'ом — на http://www.securityfocus.com/archive/1/457638. Для нас же важен сам _факт_ небезупречной реализации Java-машины. В то время как теоретики от программирования старательно выводят запутанные диаграммы, иллюстрирующие "продвинутую" модель безопасности Java с многоуровневой системой защиты, хакеры дизассемблируют библиотечные файлы на предмет поиска реальных уязвимостей.

Военная мудрость гласит — чем больше расставлено линий обороны, тем _выше_ вероятность прорыва противника, ибо надежная линия обороны справляется со своей задачей и одна. Можно сколько угодно укреплять замок крепостными валами и рвами, но это не защитит его от пикирующего бомбардировщика. С "воздуха" (то есть изнутри системных библиотек) Java вообще _никак_ не защищена. Ошибки там были и будут! Достаточно просто взглянуть на размер дистрибутива (полсони мегабайт в упакованном виде) и попытаться представить себе: сколько человеко-часов требуется для его тестирования и реально ли собрать такое количество высококвалифицированных специалистов под "одну крышу". А ведь помимо стандартных библиотек общего назначения, есть еще и нестандартные, например, JGL, используемая для отрисовки трехмерных объектов и широко применяемая в задачах на моделирование…

заключение

Интерес хакеров к Java-технологиям неуклонно растет. Отрабатываются исследовательские методики, создаются инструменты для анализа байт-кода и различные испытательные стенды для верификатора, etc. Словом, ведется интенсивная наступательная работы и в обозримом будущем по всей видимости следует ожидать взрывного роста атак на Java-машины, защищенность которых на практике оказывается значительно ниже, чем в теории.

Как этому противостоять? Увы, универсальных рецептов нет. С другой стороны, эксплуатация Java-приложений ничем не отличается ото всех остальных, написанных на

Си/Си++, DELPHI, PHP. Залог здоровья — в своевременной установке заплаток и правильном выборе партнеров — реализация JVM от Sun по многим параметрам лучше, чем у IBM, однако, в силу своей огромной распространенности у Sun-машины гораздо больше шансов быть атакованной.

Собственно говоря, никаких поводов для паники у нас нет. Java-технологии оказались уязвимыми? Ну и что?! Ах, реклама вам обещала обратное! Значит, в следующий раз не верьте рекламе. Как говориться, программ без ошибок не бывает. Бывает — плохо искали. Просто так исторически сложилось, что хакеры, убежденные в несокрушимости Java-платформы долгое время не уделяли ей никакого внимания. Теперь же, лед тронулся, господа!!!