небезопасная безопасная Java

крис касперски, no-email

безопасность Java-технологий оказалась решающим аргументом при продвижению в сферу корпоративных enterprise-приложений с единственным конкурентом лице C#. однако, пограничный слой, отделяющий рекламный маркетинг от реальной жизни, оказался довольно тернистым и местами конкретно заболоченным. Java многое обещает, но каждый раз откладывает исполнение своих обещаний на неопределенный срок. давайте рассмотрим модель безопасности Java на макро- и микро-уровнях и проанализируем сильные и слабые стороны этой технологии.

введение

Java (в девичестве Oak, что в переводе с английского означает Дуб, символизируй, в отличии от русской культуры вовсе не тупость, а мощь) возникла в результате внутреннего проекта Stealth Project, начатого в 1990 году компанией Sun и позднее последовательно переименованного в Green Project, целью которого было создания языка программирования для свой же собственной операционной системы Green Operating System, используемой для управления встраиваемыми устройствами и бытовой электроникой.

Идея создания языка принадлежит Patrick'y Naughton'y, уставшему программировать микроконтроллеры на Си/Си++, преодолевая несовместимость различных компиляторов вкупе с их привязанностью к конкретному железу, для "отвязки" от которого Patrick Naughton решил создать эффективную системно-независимую виртуальную машину. Позднее к нему присоединились James Gosling (придумавший имя Oak, которое, впрочем, оказалось уже зарезервированной торговой маркой) и Mike Sheridan, завершившие свою работу в 1992 году и продемонстрировавшие успешную работу Green OS на PDA-компьютере типа Star7.

В 1994 году порядком разросшийся коллектив программистов предпринял попытку проникновения на рынок Web-приложений, сфокусировавшись на вопросах безопасности и подготовив специальную редакцию языка HotJava (в девичестве WebRunner), предназначенную для встраивания в бараузеры, и ставшую доступной для публичного скачивания в 1995 году.

Попытка оказалась успешной и с момента поддержки HotJava браузером Netscape web-серфинг перестал быть безопасным, а сам браузер превратился в один из основных объектов хакерских атак. Несмотря на это, Java просочилась практически во все сферы рынка и сегодня она встречается повсеместно: от сотовых телефонов до enterprise-серверов (см. врезку) и суперкомпьютеров.

Внедрение Java-технологий обычно происходит под эгидой лозунгов безопасности, а все дыры списываются на ошибки реализации конкретных виртуальных машин. Однако, истинная причина в том, что Java уязвима на _концептуальном_ уровне и никакими костылями ситуацию не исправить. Впрочем, у конкурентов дела обстоят не лучше и основной соперник Java — C# содержит намного больше лазеек, _умышленно_ привнесенных компаний Microsoft для достижения большей производительности в ущерб безопасности, что и не удивительно. Безопасность — весьма абстрактное понятие, не поддающиеся измерению и не имеющее числового представления, в то время как тесты производительности — мощное маркетинговое средство.

Разумеется, это еще не означает, что Java и C# должны с позором удалиться на свалку истории, однако, это не означает и того, что ожидаемый уровень безопасности достается даром. Вот о путях достижения безопасности и о ловушках, что подстерегают нас на пути, мы сейчас и поговорим.

Рисунок 1 фирменный логотип Java

>>> врезка что такое enterprise-приложения

По сложившейся традиции enterprise-приложениями (от английского "enterprise" – предприятие) называются приложения, ориентированные на "промышленное" применение в больших организациях. Соответственно, enterprise-сервера это сервера, обслуживающие предприятия и включающие в себя: web-сервера, сервера печати, базы данных и прочие жизненно важные для функционирования корпоративной сети службы. К ключевым характеристикам enterprise-приложений относят их отказоустойчивость, быстрое восстановление после "падений" и, конечно же, безопасность (см. http://en.wikipedia.org/wiki/Enterprise_server, http://wiki.debian.org/EnterpriseServer).

Рисунок 2 enterprise-cat

многоликая Java

Прежде, чем приступать к обсуждению системы безопасности Java-приложений, необходимо провести водораздел между Java-технологиями и одноименным языком программирования, с которым, собственно говоря, и ассоциируется торговая марка Java.

Общеизвестно, что Java является интерпретируемым языком, однако, она существенно отличается от большинства других интерпретируемых языков таких, например, как Perl, PHP или Python. Если в Perl'е интерпретации подвергается непосредственно сам исходный код, то программа, написанная на Java, транслируется в байт-код, исполняемой на виртуальной Java-машине (далее по тексту JVM).

Согласно терминологии, предложенной компанией Sun, реализатор JVM называется клиентом и всякий клиент вправе исполнять байт-код так, как ему заблагорассудится (естественно, в рамках спецификации JVM). Наряду с программными реализациями JVM существуют и аппаратные, демонстрирующие производительность ничуть не уступающую чистому машинному коду процессором семейства x86, Alpha и др. (а зачастую, даже превосходящую его в силу грамотной оптимизации байт-кода JVM). С другой стороны, большую популярность завоевали JIT (Just-In-Time) компиляторы, на лету транслирующие байт-код в "нативный" (native) машинный код соответствующего процессора и сочетающие высокое быстродействие с дешевизной реализации.

Таким образом, Java-приложения представляют собой двоичные файлы, не имеющие ничего общего с исходными текстами, составленными на языке Java. Байт-код виртуальной машины предоставляет довольно богатый набор инструкций, описанный в спецификациях на JVM, что позволяет сторонним разработчикам создавать свои собственные трансляторы, работающие с языками программирования отличными от Java. Так, например, уже появились и завоевали популярность **Жасмин** (Java-ассемблер), **Ephedra** (компилятор, транслирующий

Си/Си++ программы в байт-код JVM), **Component Pascal** (компилятор, транслирующий Pascal и Oberon программы в байт-код JVM), etc. Так же имеются трансляторы и для других языков: Ада, Бейсик, Форт, Кобол, etc.

Java представляет собой объективно-ориентированный язык программирования со строгим контролем типов, выполняемым на уровне JVM, что обеспечивает защиту как от "нечестных" трансляторов (не придерживающихся оригинальной спецификации), так и от прямой модификации байт-кода в hex-редакторе. В тоже самое время, такой подход существенно затрудняет трансляцию Си-программ, известных своим нецензурным кастингом (от английского to cast – явное преобразование типов) и вольным обращениям с указателями.

Говоря о безопасности Java, мы, главным образом, сосредоточимся на JVM, поскольку системы контроля, встроенные непосредственно в сам язык программирования Java, работают лишь на стадии трансляции, страхуя Java-программистов от непредумышленных ошибок, но не спасающих от целенаправленной атаки на байт-код.

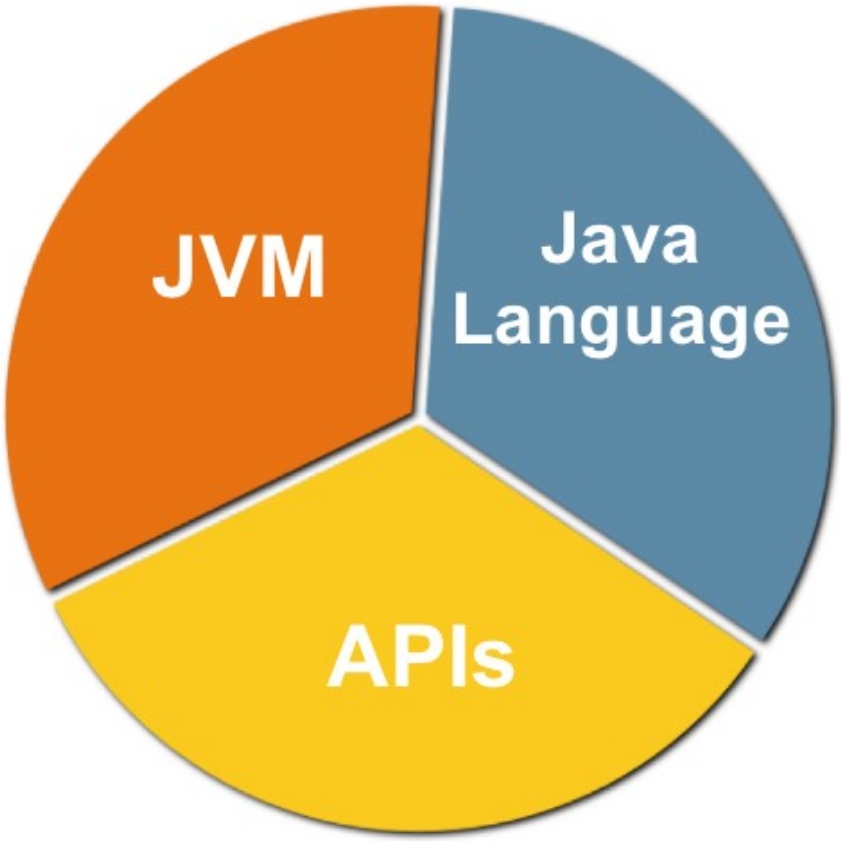

Рисунок 3 Java-технологии

концептуальная модель безопасности

Ничто не работает так, как планировалось запрограммировать (первый закон программирования). Ничто не программируется так, как должно работать (следствие первого закона программирования). Отсюда: машинная программа выполняет то, что Вы ей приказали делать, а не то, что бы Вы хотели, чтобы она делала.

Изобилие переполняющихся буферов в Си-программах (приводящих к возможности удаленного захвата управления системой) носит фундаментальный характер, обусловленный природой самого языка программирования. Си поддерживает массивы лишь формально, и реально программистам приходится работать не с массивами, а с указателями на области памяти неизвестной длинны. Язык не выполняет никакого контроля границ буферов, всецело полагаясь на программиста, а программистам, как известно, свойственно ошибаться. Именно потому, принципиальная возможность создания безопасных программ на Си практически никогда не достигается в их конкретных реализациях, зачастую создаваемых в жестких временных рамках

и протестированных на уровне: если запускается и не падает, значит, работает. Еще ни один крупный проект, написанный на Си/Си++ не избежал ошибок проектирования. Достаточно взять SendMail или IE и подсчитать количество дыр, обнаруженных за время их существования.

Java в этом смысле выглядит весьма заманчиво. Встроенный контроль типов снимает с программиста бремя постоянных проверок границ массивов, делая их переполнение достаточно маловероятным событием (хотя, полностью исключать его все же нельзя). Автоматический сборщик мусора снижает актуальность проблемы утечек ресурсов и появления "висячих" указателей, хотя это достается дорогой ценой — снижением производительности и невозможностью создавать приложения реального времени. К тому же, подчистка мусора освобождает лишь ресурсы, уходящие из области видимости, но не способна предотвратить "локальные" утечки памяти, которые сплошь и рядом рискуют обернуться глобальными. Достаточно, например, выделять память в бесконечном цикле вплоть до полного ее исчерпания.

Возьмем, к примеру, FireFox (существенная часть которого написана с использованием Java) и сравним его с Оперой, реализованной на Си++. Лавинообразный рост дыр, обнаруживаемых в FireFox'e убедительно доказывает, что Java сама по себе от ошибок проектирования никак не спасает. Надежность программы в первую очередь зависит от профессионализма ее создателей, а уже потом от свойств выбранного языка программирования. Создавать надежную программу можно и на Си++. Опера — лучшее тому подтверждение. Это не только самый быстрый, но и самый надежный браузер, из всех имеющихся на сегодняшний день.

Складывается парадоксальная ситуация. При всей ненадежности языка Си/Си++, написанные на нем программы, как правило, намного надежнее своих Java-собратьев, хотя по логике все должно быть наоборот! А вся причина в том, что большинство старых (и опытных!) программистов, освоивших Си/Си++, не видят никаких причин для перехода на Java-платформу, преимущественно выбираемую молодыми (а, значит, неопытными) программистами. Таким образом, тот факт, что приложение написано на Java еще не является гарантом его надежности. Пускай, ошибок переполнения (типичных для Си/Си++ программ) там скорее всего не окажется, зато может присутствовать масса других дефектов проектирования и/или реализации.

Но оставим непредумышленные ошибки в стороне и перейдем к целенаправленным атакам на байт-код.

микро-уровень

JVM представляет собой виртуальную машину со встроенным контролем типов, прямым аналогом которой являются "железные" процессоры с теговой архитектурой (например, наш отечественный Эльбрус) — розовая (или голубая? нет, лучше скажем "заветная") мечта теоретиков от программирования, абстрагирующихся от реальных концепций. На макро-уровне, действительно, можно работать с объектами, не задумываясь об их внутреннем представлении, но на микро-уровне неизбежно приходится сталкиваться с физическими ограничениями объективно-ориентированного подхода.

Для достижения сколь ни будь приемлемой эффективности, в исполнительную машину приходится включать "нечестные" механизмы, работающие в обход обозначенной системы типов. Применительно к JVM это – прямые вызовы машинного кода и класс sun.misc.Unsafe, реализующий (как и следует из его названия) небезопасные методы работы с памятью — **getLong** (чтение двойного слова из памяти по заданному адресу) и **putLong** (запись двойного слова в память по заданному адресу).

Начнем с прямого вызова машинного кода, являющегося _документированной_ особенностью JVM, во всяком случае в ее реализациях от Sun вплоть до версии 1.5.6 (начиная с 1.5.6 возможность вызова машинного кода как будто бы исключена. "как будто бы" потому, что спецификации на JVM старших версий отсутствуют и всю информацию приходится добывать путем "обратного проектирования").

С каждым методом класса связана специальная структура, одним из полей которой является указатель на машинный код (точнее, псевдо-указатель, но это уже детали). Если он равен нулю, то выполняется "родной" байт код, расположенный в хвосте структуры, в противном случае, управление передается по псведоуказателю. Изначально этот механизм задумывался для вызова внутренних RTL-функций, критичных к производительности и для компиляции в память JIT-трансляторами.

Стоп! Что же это такое получается?! Неужели в Java изначально присутствовала зияющая дыра в безопасности?! Ведь любой злоумышленник запросто может внедрить в байт-код настоящий машинный код, делающий все, что угодно!!! На самом же деле, в Sun вовсе не

дураки сидят и перед запуском Java-приложения среда исполнения тщательно проверяет байт-код, отбрасывая пользовательские классы с ненулевым указателем. Динамическая проверка намного менее щепетильная и хотя непосредственная модификация указателя на машинный код посредством метода putLong в большинстве случаев отлавливается средой исполнения, байт-код, откомпилированный в память, может беспрепятственно "хачить" указатели по своему усмотрению и среда исполнения оказывается не в состоянии отличить "честную" модификацию указателя, выполненную IT-компилятором, от "нечестной".

Впрочем, даже не прибегая к машинному коду с одними лишь методами getLong/putLong можно существенно пошатнуть модель безопасности Java, произвольным образом модифицируя внутренние данные классов и меняя типы переменных вместе с атрибутами классов (public, final, etc), что позволяет реализовать тот самый "нецензурный кастинг", приводящий к ошибкам переполнения (к умышленным, разумеется) и возможности удаленного захвата управления машинной с передачей управления на shell-код (только для функций, откомпилированных в память).

Важно понять, что методы getLong/putLong являются не функциями, поставляемыми вместе с библиотекой времени исполнения, а командами JVM! То есть, заблокировать их вызов напрямую не получится, а если бы и получилось — многие штатные библиотеки тут же бы отказали в работе. Вообще же говоря, вообразить набор инструкций исполнительной машины без возможности низкоуровневой работы с отдельными ячейками памяти — невозможно! А раз так, у нас есть все основания полагать, что инструкции getLong/putLong — это надолго (если не навсегда) и потому следует присмотреться к ним повнимательнее.

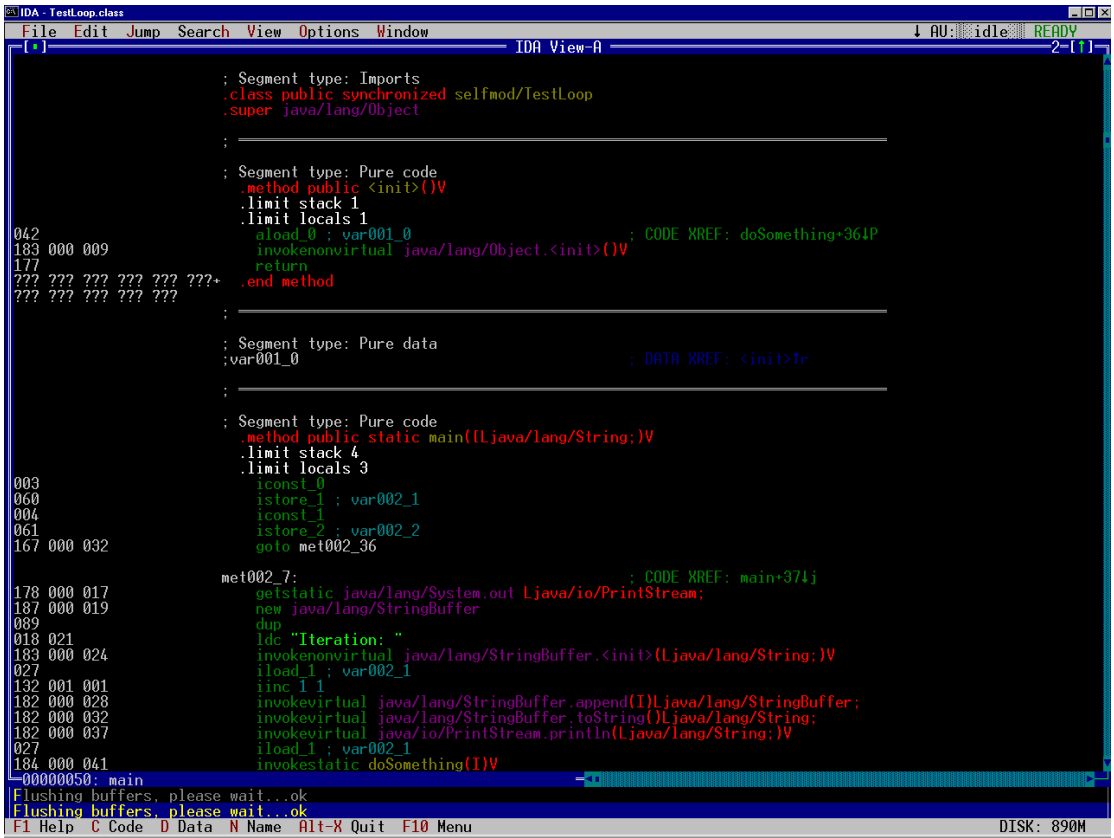

Рисунок 4 байт-код в дизассемблере IDA-Pro

макро-уровень

Начиная с JVM 1.0 в ней появилась метафора "песочницы" (sandbox) — изолированной среды, в которую помешаются потенциально опасные программы (например, Web-приложения, полученные из ненадежных узлов). Песочница как бы отрезана от файловой системы и может общаться только с тем узлом, с которого было загружено данное Java-приложение. "Как бы" потому что не существует ни одной реализации JVM, отвечающей этому требованию не только

на бумаге. Ряд атак на IE и FireFox как раз и основан на возможности прорыва за пределы "песочницы" и перезаписи локальных файлов.

Рисунок 5 JVM и "песочница"

Решение проблемы заключается в запуске IE/FireFox от имени пользователя, которому недоступны никакие файлы, кроме тех, что требуется для работы бразуреза. Тем не менее, атакующему по-прежнему доступны cookies, кэш страниц и другие данные, утечка которых крайне нежелательна, а в некоторых случаях недопустима и зачастую влечет к потере контроля над своими аккаунатами, поэтому, достаточно многие компании отказываются от Java, запрещая выполнение Java-приложений в браузере.

Но это еще что! Java-приложения, находящиеся на локальном диске, по умолчанию считаются безопасными и им доступны все ресурсы JMV, в том числе — файлы, сетевые соединения и т.д. Создание компьютерного вируса на Java не только возможно, но и не сильно отличается от вирусов, написанных на остальных языках программирования (Си, Паскале, Ассемблере). Сказанное относится и к web-страничкам, сохраненных на диск. При последующем открытии они уже считаются "безопасными" со всеми вытекающими отсюда последствиями. То есть, для успешной атаки злоумышленнику достаточно заманить жертву на страницу с вредоносным Java-приложением, которую жертва сохранит на диск. Вообще-то, при желании настройки браузера легко изменить, но тогда перестанут работать и все действительно безопасные приложения, нуждающиеся в доступе к файлам/сетевым соединениям. Вот такая, значит, напряженная ситуация получается.

Осознавая ущербность предложенной модели безопасности компания Sun начиная с JVM 1.1 ввела поддержку электронной подписи, благодаря которой вредоносный код потерял все шансы. Естественно только на бумаге, а в реальной жизни… Чтобы не терять совместимость с уже написанным программным обеспечением, проверка цифровой подписи по умолчанию была либо выключена, либо при загрузке неподписанного Java-приложения выдавала запрос на подтверждение, на который большинство пользователей отвечало утвердительно. Java-скриптов цифровая подпись вообще никак не коснулась и при открытии сохраненной WEB-странички с диска они по-прежнему получали все права.

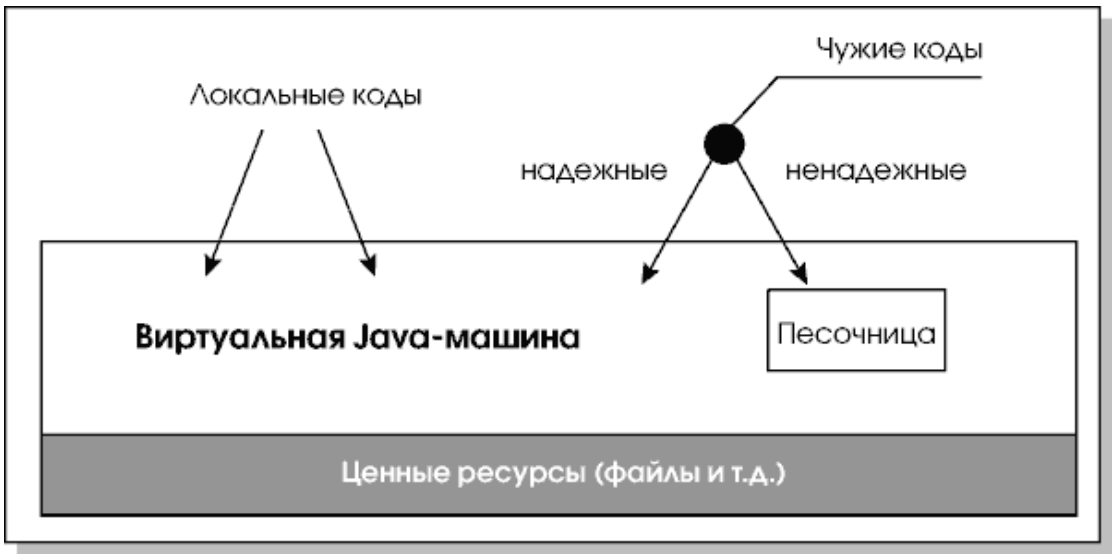

Рисунок 6 модель безопасности JVM 1.1

В следующий версии виртуальной машины политика безопасности была пересмотрена и существенно расширена. Деление на "ненадежные" и "надежные" приложения исчезло, уступив место правам доступа. Теперь приложения могли обращаться только к определенному перечню ресурсов, определяемых администратором системы, что само по себе огромный прогресс, поскольку концепция "все или ничего" (т. е. "песочница" или "живая" среда) оказалась крайне не гибкой. Трудно представить себе полновесное приложение, довольствующееся "песочницей". С другой стороны, если делегировать каким попало Java-приложения права доступа ко всем ресурсам — то какая тут к черту "безопасность"?!

Введение селективного контроля за ресурсами потребовало реализации "контекста выполнения" — проверяя права доступа объекта к ресурсу, JVM вынуждена анализировать не только данный объект, но и предыдущие элементы стека вызовов, предоставляя доступ тогда и только тогда, когда нужным правом владеют все объекты в стеке (в терминологии Sun это называется принципом минимизации привилегий).

Принцип минимизации привилегий, как легко видеть, вступает в противоречие с принципом инкапсуляции. Объект foo, опирающийся на объект bar, в "правильных" ООП языках не знает о внутреннем устройстве bar и потому bar может (при необходимости) пользоваться ресурсами, недоступными для foo. Классическим примером тому является системный вызов операционной системы, вызываемый прикладной программой. Объект "файл", имеющий прямой доступ к диску, предоставляет остальным объектам набор методов, для создания/удаления/чтения и записи файлов, гарантируя, что никакой другой объект не разрушит данные на диске. Если же следовать принципу минимизации привилегий, то прямой доступ к диску необходимо предоставить всем объектам, а это уже чистый абсурд.

Другими словами, если объект foo имеет право вызывать данный метод объекта bar с заданными аргументами, то bar обязан обслужить вызов, в противном случае пришлось бы учитывать возможный граф вызовов объектов, что требует огромных затрат памяти и процессорного времени.

Механизм, реализованный в JVM, обходит эту проблему путем создания привилегированного интервала, при выполнении которого контекст (т. е. предыдущие вызовы объектов) игнорируется, в результате чего становится возможным создание программ, нарушающих делегированные им права доступа (неважно — сознательно или нет). На это еще можно было бы хоть как-то закрыть глаза если бы не тяжеловесность реализации и высокие накладные расходы. Как женщина не может быть "слегка" беременной, так и система не бывает "практически" безопасной. Расплачиваться за проверку прав доступа, не будучи при этом уверенным, что она выполняется правильно — это же какое промывание мозгов необходимо устроить, чтобы удержать пользователей на Java! В действительности, в Java намного меньше технологии, чем маркетинга.

заключение

Несмотря на все недостатки, присущие Java, следует признать, что подобного уровня защищенности на сегодняшний день не обеспечивает ни один из существующих языков программирования. Единственный серьезный конкурент — C# проигрывает языку Java по многим позициям. Во-первых, он намного менее распространен, а, во-вторых, его разработчики пошли на сделку с производительностью в ущерб безопасности, реализовав небезопасные методы обращения к памяти и данным.

Можно сколько угодно критиковать Java, но новых инструментов от этого не добавится. Тем не менее, пользователям всегда следует помнить, что выбор программного обеспечения должен осуществляться на основании реальных данных об их надежности, а не только потому, что они написаны на Java. Программистам же не следует забывать о том, что Java лишь уменьшает вероятность появления _некоторых_ ошибок проектирования, взимая за эту довольно большую мзду, так что выбор наиболее предпочтительного инструмента по-прежнему остается предметом острых дискуссий.

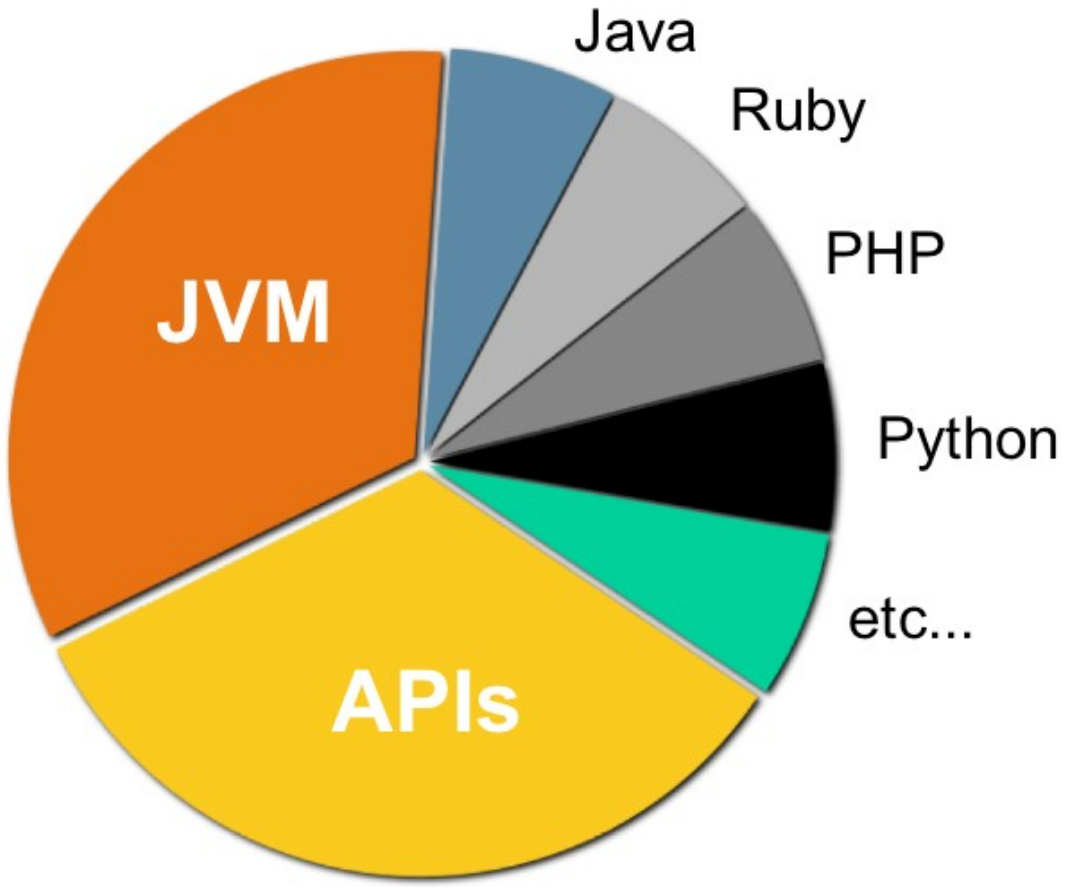

Рисунок 7 прогноз популярности интерпретируемых языков на блажащее время (по данным http://www.tbray.org/ongoing/When/200x/2006/02/02/LAMP-Java-Sun)

>>> врезка что читать

- **Java (Sun):**
  - описание технологий Java на википелии (на английском языке): http://en.wikipedia.org/wiki/Java_%28Sun%29;
- **Java (programming language)**:
  - описание языка программирования Java на википелии (на английском языке): http://en.wikipedia.org/wiki/Java_%28programming_language%29; тоже самое на русском языке: http://ru.wikipedia.org/wiki/Java;
- **The Java Virtual Machine Specification**:

- официальая специкация на JVM от Sun (на английском языке): http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html;

- **FAQ Applet Security**:
  - официальный faq по безопасности Java-апплетов от Sun (на английском языке): http://java.sun.com/sfaq;
- **Low Level Security in Java**:
  - официально описание безопасности JVM на микро-уровне (на английском языке) http://java.sun.com/sfaq/verifier.html;
- **New Java SE 6 Feature: Type Checking Verifier**:
  - официальное описание возможностей верификатора (на английском языке) https://jdk.dev.java.net/verifier.html;
- **Unsafe Java I - Небезопасная жаба**:
  - статья, посвященная структуре байт-кода и работой с памятью JVM на низком уровне посредством команд getLong/putLong (на русском языке): http://www.wasm.ru/print.php?article=unsjav1 (первая часть) http://www.wasm.ru/print.php?article=unsafe_ii (вторая часть);
- **Java crackme / proof of concept**:
  - обсуждение статьи "Unsafe Java I - Небезопасная жаба" на форуме cracklab http://www.cracklab.ru/f/index.php?action=vthread&topic=5363&forum=2&page=-1
- **Механизмы безопасности Java**:
  - глава из книги "Java в три года", довольно беспристрастно описывающая систему безопасности Java на макро-уровне (на русском языке): http://www.jetinfo.ru/1998/11-12/1/article1.11-12.19981237.html;
- **C to Java byte-code compiler**:
  - трансляторы Си в байт-код JVM (на английском языке): http://en.wikipedia.org/wiki/C_to_Java_Virtual_Machine_compilers;

>>> врезка изменения JVM

Структура байт-кода и набор инструкций JVM не остается постоянным, а меняется от версии к версии, что существенно затрудняет как создание независимых трансляторов от сторонних производителей, так и реализацию атак на байт-код, поскольку, хакеру приходится либо фокусироваться на строго определенных версиях (которых может вообще не оказаться у жертвы), либо учитывать особенности каждой, отдельно взятой, реализации JVM, что весьма непросто.

К тому же, команды виртуальной машины медленно, не неуклонно движутся к изъятию потенциально опасных инструкций. В частности, из лексикона Java SE 6 исчезли команды JSR и JSR_W, представляющие собой отдаленный аналог Бейсик-команды GOSUB, передающий управления на процедуру. Вот что по этому пишет Sun: "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" ("*Новый верификатор запрещает выполнение инструкций JSR и RET. Эти инструкции используются для вызова подпрограмм при генерации try/finally-блоков. Вместо этого, компилятор будет встраивать код программ непосредственно по месту вызова" – перевод мой, КК*).

>>> врезка верификатор

Верификатор байт-кода является неотъемлемой частью JVM и проверяет каждую выполняемую инструкцию виртуальной машины (в том числе и добавленную динамически, путем, создания самомодифицирующего кода, например), предотвращения поступление заведомо некорректной информации.

Считается, что верификатор предотвращает следующие операции:

- подделка указателей, например, получение указателя как результат выполнения арифметической операции (однако, инструкции getLong/putLong позволяют обращаться к любой ячейке памяти и верификатор им не помеха);
- нарушение прав доступа к компонентам классов, например, присваивание переменной, снабженной описателем final (инструкции getLong/putLong без труда обходят это ограничение);

- использование объектов в каком-либо другом качестве, например, применение к объекту метода другого класса (инструкции getLong/putLong позволяют обойти систему контроля типов);

Фактически, верификатор решает намного более скромные задачи, препятствуя:

- вызову методов объектов с недопустимым набором параметров;
- вызову инструкций JVM с недопустимым набором параметров;
- некорректную операцию с JVM-регистрами;
- переполнение или исчерпание стека.

Для достижения максимальной производительности, верификатор совершает ряд допущений, смягчая проверку "мертвого" кода (т. е. кода, который по мнению верификатора никогда не получит управление), а реализация верификатора в JIT-компиляторов из динамической (т.е. выполняемую на _каждом_ шаге) и вовсе вырождается в статическую (т.е. выполняемую _до_ компиляции). В частности, проверка границ массивов (отнимающая много времени) опускается всякий раз, когда JIT компилятор считает, что нарушения доступа на данном участке кода заведомо не происходит. Сравнивая реализации JVM от Sub и Microsoft, нельзя не заметить, что: а) реализация от Microsoft работает существенно быстрее; б) реализация от Sun выполняет намного больше проверок. Реализация JVM от IBM обеспечивает достаточно высокую производительность без ущерба существенного ущерба для безопасности.