howest
De Hogeschool West-Vlaanderen
Lid van de Associatie Universiteit Gent

CAN JAVASCRIPT FROM UNTRUSTED SOURCES BE SAFELY & SCALABLY EXECUTED ON THE JVM?

Promotor: Mh Hans Ameel

Onderzoeksvraag uitgevoerd door

THOMAS TOYE

Voor het behalen van de graad van Bachelor in de

NEW MEDIA AND COMMUNICATION TECHNOLOGY

Howest | 2015-2016

Can JavaScript from untrusted sources be safely executed on the JVM?

Thomas Toye, hello@thomastoye.be, 3NMCT, Howest University College West Flanders

Advisor & supervisor: Hans Ameel, hans.ameel@howest.be, Howest University College West Flanders

Preface

I'm Thomas Toye. I have extensive experience with all sorts of back-end programming languages (from PHP and Python to JavaScript, Java and Scala). I want to investigate techniques and tools to make web back-ends faster and more secure.

Years and years ago, when Flash was at its peak, I read an article on how to communicate from an embedded Flash application on a web page to a Java applet on the same page. “Surely the author of this article has gone mad”, I thought. And in the same way, some people might believe I have gone mad. After all Java and JavaScript are two very different languages. However, running JavaScript on the Java Virtual Machine is not as strange as it may seem, and can solve real problems.

JavaScript has gone mainstream in recent years. What started out a language created in a mere 10 days and used for small things such as custom mouse pointers, has become a global trend that allows the rapid creation of single page applications, 3D animations and even complete desktop and mobile applications. But there is also another side to JavaScript: in 2009, Node.js was released. It allowed to write JavaScript server-side. By the end of 2010, Node.js entered a viral state, rapidly gaining developer mind share. It became the rapid-prototyping back-end platform of choice for many companies, and is often mentioned with the Internet of Things in the same breath.

But rapid-prototyping tools are only half the problem. We are also concerned with the run-time performance of our programming languages. That's where Google comes in: after some slow years in JavaScript engines, Google innovated with the V8 JavaScript engine (which also allowed Node.js to come to life). Its speed improvements over existing JavaScript engines meant that developers started to pay more attention to JavaScript again.

Although it has been possible to run JavaScript on the Java Virtual Machine for a long time now (thanks to Rhino), another recent innovation is trying to make this as mainstream as Node.js: Nashorn. Nashorn is a JavaScript engine written in Java and developed by Oracle. It's included in JDK8, which was released in May 2014. Its main selling point is its speed: the Nashorn team claims Nashorn is orders of magnitude faster than Rhino (Oracle, 2014[1]).

[1] Oracle Corp.. (2014, December 12). Nashorn Architecture and Performance Improvements in the Upcoming JDK 8u40 Release (Nashorn) [Blog post]. Retrieved from

When we're talking about server-side JavaScript, what we really want to talk about is Node.js. As a server-side JavaScript platform, it has attracted so much developers that there is no real alternative that will attract developers. Thus, it is important a back-end JavaScript platform implements at least part of the Node.js APIs, which will attract developers because the platform feels familiar.

https://blogs.oracle.com/nashorn/entry/nashorn_performance_work_in_the

Abstract

The purpose of this research paper is to find if it is possible to securely run JavaScript, originating from an untrusted source, on the Java Virtual Machine. After an analysis of the risks associated with this, I review viable platforms for running both JavaScript and Node.js on the JVM. My findings indicate that it is currently not possible to securely execute JavaScript on the Java Virtual Machine.

Table of contents

Table of figures

Glossary

JVM	A Java Virtual Machine is a program that can interpret Java bytecode
JSR	“Java Specification Requests (JSRs) are the actual descriptions of proposed and final specifications for the Java platform. At any one time there are numerous JSRs moving through the review and approval process“ (Oracle Corporation, n.d.[2])
JavaScript	A high-level dynamic language, most commonly in the browser, but it also has applications server-side.
JavaScript engine	A JavaScript engine is a computer program that can interpret and execute JavaScript code.
Rhino	Rhino is a Java-based JavaScript engine.
Nashorn	Nashorn is a Java-based JavaScript engine.
Node.js	Node.js is a platform for server-side JavaScript.
Java	Java is one of the most popular programming languages. It is usually compiled to bytecode, which his then run on a Java Virtual Machine.
DoS	A Denial of Service attack is an attack that aims to make a system unavailable for other users.
Threat	A threat is any agent that can exploit a vulnerability.
Vulnerability	A vulnerability is a weakness in a system.
Risk	The combination of a vulnerability and a threat, the possibility of damage to a system.
Pointer	A memory space that hold the address of (another) memory space. It “points” to the other memory space.
Reference type	A reference type is a memory space that references an object on the heap.
Primitive type	A primitive type is a memory space that holds the value of the type in its own memory space.
0-day	A 0-day, or zero-day, is a vulnerability that is not disclosed and immediately exploited by malicious agents. The system vendor
JVM language	“JVM languages” refer to languages to get compiled to bytecode that can run on a JVM. Examples of JVM languages are Java, Scala, Kotlin and Clojure.
Java classloader	The Java classloader is responsible for locating Java classes and loading them into memory. Custom classloaders can be implemented to locate classes in different locations or transform the bytecode.

[2]Oracle Corporation. (n.d.). JSRs: Java Specification Requests. Retrieved February 22, 2016, from https://jcp.org/en/jsr/overview

Introduction

Security

When talking about security, we use a few terms that commonly get mixed up. Here are the terms used in this research paper and a short explanation.

Threat and threat assessment

A threat is any agent that can exploit a vulnerability. This includes automated malware, such as spyware, worms and viruses, human threats, examples of which include fired employees, criminals. But it also includes non-human threats, such as loss of power or hard drive malfunctions. Thus, threats are the entities we try to protect our systems against.

Threat assessment is how we determine how we can best secure a system. This is done by analysing what the attacker can do and how they would perform an attack. An example of how a threat assessment can be performed, is a (black-box) penetration test.

Vulnerability

A vulnerability is a weakness in our systems, a security flaw. An example of a vulnerability is an endpoint that neglects to check if the client has the appropriate permission to execute the requested command.

Risk and risk assessment

A risk is a combination of a vulnerability and a threat. TAG (Threat Analysis Group) defines a risk as "The potential for loss, damage or destruction of an asset as a result of a threat exploiting a vulnerability." (Threat Analysis Group, n.d.[3])

With a risk assessment, we analyse our systems to see where potential vulnerabilities lie and how they could be exploited by an attacker. The difference between risk assessments and threat assessments is best described by Ched Perrin: "Where risk assessments focus more on analysing the potential and tendency of one's resources to fall prey to various attacks, threat assessments focus more on analysing the attacker's resources." (2009 [4])

[3] Threat Analysis Group. (n.d.). Threat, vulnerability, risk – commonly mixed up terms. Retrieved from http://www.threatanalysis.com/2010/05/03/threat-vulnerability-risk-commonly-mixed-up-terms/

[4] Ched Perrin, M. R. (2009, July 7). Understanding risk, threat, and vulnerability. Retrieved from

The CIA model

The CIA model is comprised of the concepts of confidentiality, integrity and availability, which are central in information security. The CIA model is the simplest of many models in use in the information security business.

Confidentiality

Confidentiality means that our data is only available to people authorized to access it. For example, there is a breach of confidentiality when a system does not have appropriate authorisation checks, and allows anyone access to data.

Integrity

Integrity means that our data is complete: that no pieces of it are missing, and that it is not corrupted. For example, a hard drive failure could break the integrity property if it the data on the hard drive was not stored anywhere else.

Availability

Availability means that people who need access to data and have authorisation can access it when they need to. For example, the availability property of a system is not full-filled if the system is on an air gapped network, and an authorised person without access to the air gapped network needs access to the data stored on the system.

JavaScript

No paper on JavaScript is complete without an explanation and short history of JavaScript. Here's my take.

Brendan Eich, an experienced programming language writer, was hired by Netscape in 1995 to create a prototype language for Netscape, which he did in 10 days (Severance, 2010 [5]). The language that would later be known as JavaScript was first called Mocha, later LiveScript and ultimately JavaScript.

JavaScript as a client-side language for the browser started to grow and was standardised by ECMA

http://www.techrepublic.com/blog/it-security/understanding-risk-threat-and-vulnerability/

[5] Charles Severance, M. R. (2010, February). JavaScript: Designing a Language in 10 Days. Retrieved from https://www.computer.org/csdl/mags/co/2012/02/mco2012020007.pdf

in 1997 (ECMA International, 1997 [6]). In 2005, a new era of web applications started to bloom, thanks to AJAX: the XMLHttpRequest object allowed client-side JavaScript developers to refresh information and get content without a page refresh. In his influential paper, Jesse Garrett (2005 [7]) cites Google Suggest and Google Maps as "[...] two examples of a new approach to web applications that we at Adaptive Path have been calling Ajax. The name is shorthand for Asynchronous JavaScript + XML, and it represents a fundamental shift in what's possible on the Web." Unimaginable today, updating content on a web page without refreshing the page was a paradigm shift at the time.

Node.js brought even more momentum to JavaScript by providing a server-side platform for JavaScript. Node.js got its name in March 2009 (Dahl [8]), and it only got more popular from there. Express, the most popular framework for Node.js, had its initial commit in June 2009 (Holowaychuk [9]).

Interestingly, Node.js wasn't the first server-side JavaScript platform. To the contrary, JavaScript on the server had a much earlier start, even before the year 2000, people saw the potential. Microsoft had included Jscript (Microsoft's JavaScript implementation) as a server-side language in IIS in 1998 (Microsoft Corporation [10]) and Netscape's Enterprise Server supported server-side JavaScript in 1997 (Oracle Corporation [11]).

Node.js

Node.js is a runtime for server-side, event-driven JavaScript. It runs JavaScript on the V8 JavaScript engine, although other engines are now trying to create a runtime to allow Node.js to run on the Chakra JavaScript engine (Krill, 2016 [12]) and on the JVM, using Trireme or Rowboat. Node.js is

[6] ECMA International. (1997, June). ECMAScript: A general purpose, cross-platform programming language. Retrieved from http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%201st%20edition,%20June%201997.pdf

[7] Garrett, J. J. (2005, February 18). Ajax: A New Approach to Web Applications. Retrieved from http://adaptivepath.org/ideas/ajax-new-approach-web-applications/

[8] Dahl, R. (2009, March 3). Major refactoring: program name now "node". Retrieved from https://github.com/nodejs/node-v0.x-archive/commit/19478ed4b14263c489e872156ca55ff16a07ebe0

[9] Holowaychuk, M. R. (2009, June 26). Initial commit (expressjs/expressjs). Retrieved from https://github.com/expressjs/express/commit/9998490f93d3ad3d56c00d23c0aa13fac41c3f6b

[10] Microsoft Corporation. (1998, September). Using VBScript and JScript on a Web Page. Retrieved from https://msdn.microsoft.com/en-us/library/aa260861%28v=vs.60%29.aspx

[11] Oracle Corporation. (1997, December 19). Writing Server-Side JavaScript Applications with Enterprise Server 3.x. Retrieved from https://docs.oracle.com/cd/E19957-01/816-5653-10/816-5653-10.pdf

[12] Krill, P. (2016, January 20). Node.js welcomes Microsoft's Chakra JavaScript engine. Retrieved from http://www.infoworld.com/article/3024271/javascript/nodejs-welcomes-microsoft-chakra-javascript-engine.html

usually used to create web services, but it can also be used to create tooling, such as task runners and build tools, command line scripts, and even desktop applications.

Difference between JavaScript and Node.js

Node.js is the combination of Google's V8 JavaScript engine, and a bundle of JavaScript libraries, which provide streams, access to the file system, access to the network... Node.js uses an event-driven, asynchronous paradigm for I/O (Node.js Foundation, n.d. [13]).

More conceptually, Node.js can also be seen as just the bundle of libraries, and any JavaScript engine which supports these libraries.

Virtual machine

When people hear the term virtual machine, they will more often than not think of a system virtual machine, which is a computer application that can simulate the execution of an entire, virtual computer. Example system virtual machines include the VMware range of products, Oracle VirtualBox, and KVM.

There is also a different kind of virtual machine: a process virtual machine. Process virtual machines are used to execute a single program (usually a single process, hence the name). The Java Virtual Machine (JVM), HipHop Virtual Machine (HHVM) (Facebook, n.d. [14]) and Bogdan/Björn's Erlang Abstract Machine (BEAM) are examples of process virtual machines. They are usual made specifically for a programming language (the JVM for Java, HHVM for PHP and BEAM for Erlang) and specialized for the use cases of that language. Other language implementations may target different virtual machines, such as JRuby and Jython targeting the Java Virtual Machine, but that is beside the point here. In this research paper, “virtual machine” is used in the sense of a “process virtual machine”, unless otherwise noted.

[13] Node.js Foundation. (n.d.). Node.js. Retrieved March 17, 2016, from https://nodejs.org/en/
[14] Facebook. (n.d.). HHVM. Retrieved March 17, 2016, from http://hhvm.com/

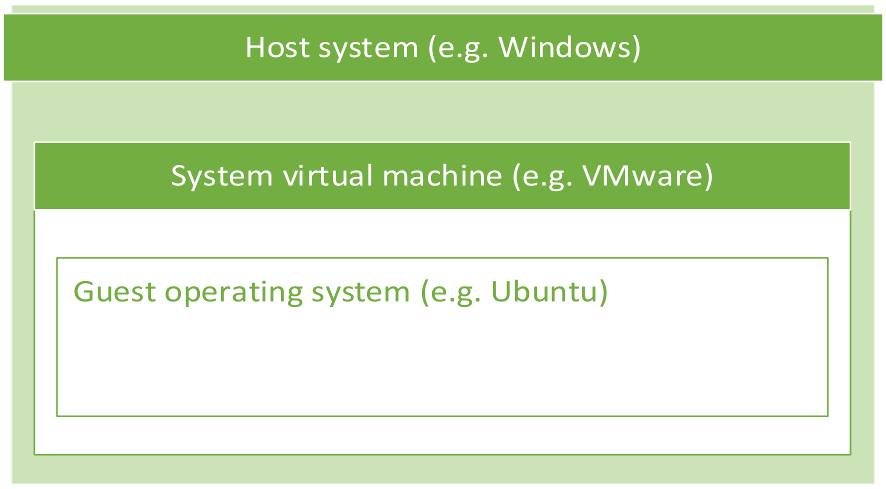
Host system (e.g. Windows)
System virtual machine (e.g. VMware)
Guest operating system (e.g. Ubuntu)

Figure 1 An example of a host running a system virtual machine running a guest operating system

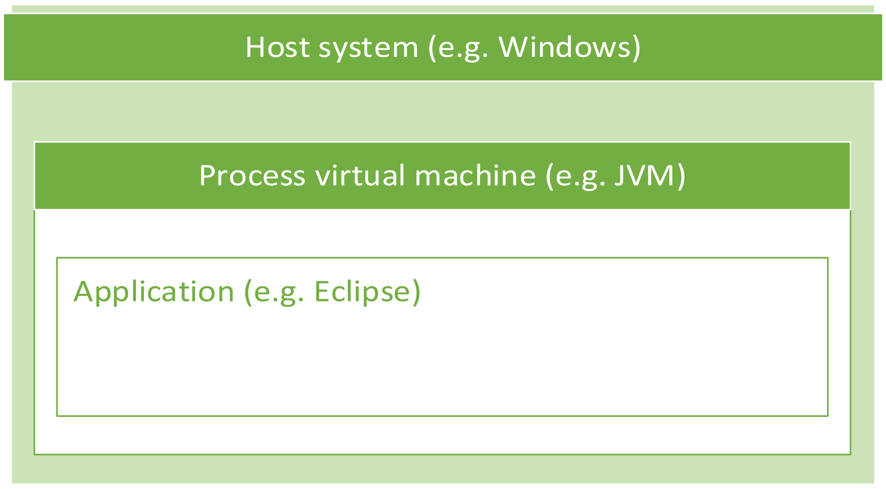
Host system (e.g. Windows)
Process virtual machine (e.g. JVM)
Application (e.g. Eclipse)

Figure 2 An example of a host running a process virtual machine running an application

Scala

Scala is a JVM programming language. It's the programming language that will be used in examples in this research paper, because of its expressiveness. While Scala can be written in a very functional, almost Haskell-like way, I choose to keep the few examples here readable and Java-like.

Automated testing

I wrote automated tests where possible and applicable to confirm claims I make in this paper, and to verify I correctly interpreted sources. These tests were written in Scala using Specs2 (“Software

Specifications for Scala”, a Scala testing framework (Torreborre, n.d. [15]) and are included in the appendices.

[15] Torreborre, E. (n.d.). specs2 User Guid. Retrieved March 17, 2016, from https://etorreborre.github.io/specs2/guide/SPECS2-3.7.2/org.specs2.guide.UserGuide.html

Body

The Java Virtual Machine

A Java Virtual Machine, or the JVM as it is colloquially known, is the runtime that allows to execute Java programs. There is no single Java Virtual Machine, there is only the Java Virtual Machine specification (Oracle Corp., 2015 [16]), and there are multiple JVM implementations, such as HotSpot and Jrockit. HotSpot is the JVM implementation that ships with OpenJDK

The Java Virtual machine is a process virtual machine originally developed for running Java. Later, other programming languages were created that targeted the JVM (Clojure, Scala, Kotlin...) and some existing programming languages had implementations written that ran on the JVM (Jython, JRuby...).

The JVM has its own instruction set, much like a real computer processor, which is called Java Bytecode. This bytecode is interpreted by the JVM. Bytecode is what the `javac` compiler produces, in the form of .class files. While the bytecode is initially interpreted, most JVM implementations use a Just-In-Time (JIT) compiler to speed up execution: commonly executed code paths are translated into native code, which is much faster to execute. The Java Virtual Machine Specification does not specify that a Just-In-Time compiler is mandatory (Oracle, 2015 [17]), this is up to the implementation.

[16] Oracle Corp.. (2015, February 13). The Java® Virtual Machine Specification Java SE 8 Edition. Retrieved from http://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf
[17] Oracle Corp.. (2015, February 13). The Java® Language Specification Java SE 8 Edition. Retrieved from http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

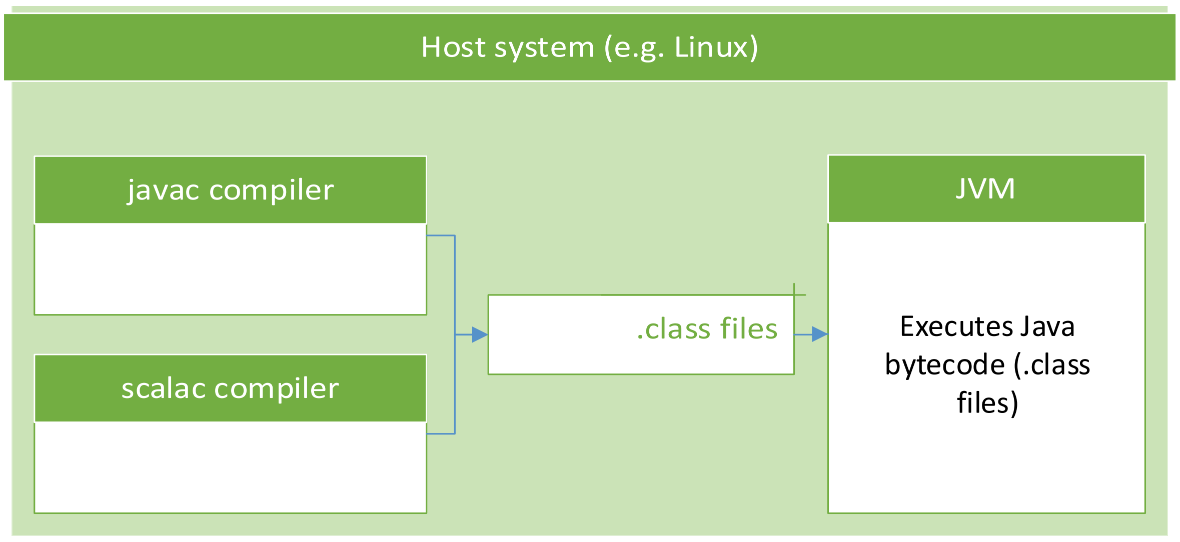

Figure 3 An illustration of the relationship between JVM languages and the JVM itself. Java and Scala sources get compiled to bytecode, which can then be executed on the JVM.

Difference between Java and the Java Virtual Machine

Java is a programming language. While Java does have a compiler (Oracle Corporation, n.d. [18]), it is not compiled down to machine code, like a C compiler does. Instead, it's compiled to Java bytecode. This bytecode cannot be run natively on a computer, it is meant to be executed by a virtual machine: the Java Virtual Machine, or JVM for short. The instruction set of this virtual machine is Java bytecode.

Even though Java and the Java Virtual Machine are very different, the terms do get mixed a lot. For example, the bytecode, which runs on the Java Virtual Machine, is called "Java bytecode", not "Java Virtual Machine bytecode", even though there are other languages, which also produce the same kind of bytecode.

Security on the Java Virtual Machine

History of security on the Java Virtual Machine

Historically, the Java Virtual Machine has had a very poor reputation of being insecure. JVM vulnerabilities and 0-days seemed like they were daily occurrences. People were told to switch "disable Java", because "[...] these days, Java is a favourite attack vector for hackers." (Rubenking,

[18] Oracle Corporation. (n.d.). javac - Java programming language compiler. Retrieved March 19, 2016, from http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html

2013 [19])

Applets and Java's bad reputation

It's important to make a distinction between client-side Java (or another JVM language) and server-side Java. What Rubenking was referring to, was the client-side JVM that allows web pages to run Java applets inside the browser. This technology was very popular in the early days of the web, when JavaScript did not have the capabilities it has today.

The problem with these Java applets was that code from third parties was executed inside a JVM on a client machine. This meant that the code running on the client could access the file system, access the network...

Java evolved to remove vulnerabilities like this. While the JVM had gotten a sandbox in JDK 1.0, the real security innovations started with Java 2, where a the permission model moved away from an “all or nothing” approach to a fine-grained model, the bytecode verifier was added and concrete implementation of the SecurityManager appeared (Srinivas, 2000 [20]).

JDK 1.2 was a major step forward, not only thanks to the already mentioned innovations, but also due to the stricter take on signed applets. Digitally signed JARs were introduced with JDK 1.1, but if the signature check failed, they would still run without any restrictions. This was changed in JDK 1.2: “A programmer bundles an applet and all related files in a JAR file and digitally signs it. The person who downloads the applet verifies the signature. If the verification succeeds, the applet runs with full access to system resources. If the verification fails, the applet is confined to a sandbox.” (Sun Microsystems, Inc., 1998 [21])

Even though the JVM implemented a sandbox, it's still got hit with a lot of vulnerabilities, as a sandbox is very hard to implement.

The demise of the Java Applet is not only due to security concerns, as Byrne (2016)[22] notes: his

[19] Rubenking, N. J. (2013, March 1). How to Disable Java. Retrieved from http://www.pcmag.com/article2/0,2817,2414191,00.asp

[20] Srinivas, R. N. (2000, July 28). Java security evolution and concepts, Part 2. Retrieved from http://www.javaworld.com/article/2076135/java-security/java-security-evolution-and-concepts–part-2.html

[21] Sun Microsystems, Inc.. (1998, January 15). Security Tools: Is the Only Really Secure Computer a Dead Computer? Retrieved from http://pawlan.com/monica/articles/sectools/

[22] Byrne, M. (2016, February 2). The Rise and Fall of the Java Applet: Creative Coding’s Awkward Little Square. Retrieved from http://motherboard.vice.com/read/a-brief-history-of-the-java-applet

takes on this is that the major reason Java applets fell was the fact that JavaScript in the browser gained more capabilities and better performance.

Java has gained a reputation of being insecure. However, server-side Java is usually unaffected: after all, you're running trusted code, and not third-party code like an applet does.

JVM security measures

Type safety

The Java Virtual Machine has built-in type checks. Contrary to languages like C and C++, type checks are also performed at run-time. In C and C++, types are only used by compiler checks. In Java, types are preserved in the bytecode that is run on the JVM.

When interpreting bytecode, the JVM will actively check bytecode, including the type of objects. This increases safety and provides guarantees against malformed or corrupted bytecode. For example, the JVM will complain at runtime when casting an object that is not of the correct type (Oracle, n.d. [23]).

Memory safety

The Java Virtual Machine uses automatic memory management. This means programmers don't have to manage computer memory themselves. In C and C++, programmers have to manually allocate and free memory. The JVM takes over this error-prone task from programmers.

In Java, when creating an object using the “new” operator, memory is automatically allocated for the new object. But manually freeing an object is not necessary (and is not even possible): objects automatically get garbage collected when no other object holds a reference to them.

This has major benefits: a whole range of possible errors disappear: in Java, there are no double frees (where trying to free the same object multiple times results in memory corruption), invalid frees (where passing the memory address of a non-existent object may result in memory corruption) and dangling pointers (where a pointer points to an address that does not hold an object).

The JVM also empowers the Java programming language with array bounds protection. Trying to

[23] Oracle Corporation. (n.d.). Chapter 4. The class File Format. Retrieved March 19, 2016, from http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.10.1.9.checkcast

get an index that is out of bounds (for example, trying to get the hundredth element of an array of size 10) will result in an ArrayIndexOutOfBoundsException (Oracle Corporation, n.d.)[24].

Apart from the security benefit discussed here, there is also an ergonomic benefit: programmers do not have to spend time tracing allocations and making sure they free all allocated memory. This raises programmer productivity.

Even though the Java Virtual Machine has automatic memory management, it does not prevent all classes of errors. For example, an object may have the special value null, which means the variable does not refer to an object. It's valid syntax to assign null to a reference type, but when you try to call a method on it, you will get a run-time NullPointerException.

```
String myString = null;
myString.charAt(1); // will throw java.lang.NullPointerException
```

The automatic memory management of the JVM does not come for free. Periodically, a garbage collector will run, and will clean up objects that are not referred to by any other objects. This means that you can prevent an object from being garbage collected by merely holding a reference to it, causing memory not to be reclaimed. Programs doing this on purpose will eventually make the JVM run out of memory.

```
scala> (1 to 100000000).map(_ + 20)
java.lang.OutOfMemoryError: GC overhead limit exceeded
  ... 31 elided
```

[24] Oracle Corporation. (n.d.). ArrayIndexOutOfBoundsException (Java Platform SE 8). Retrieved February 28, 2016, from https://docs.oracle.com/javase/8/docs/api/java/lang/ArrayIndexOutOfBoundsException.html

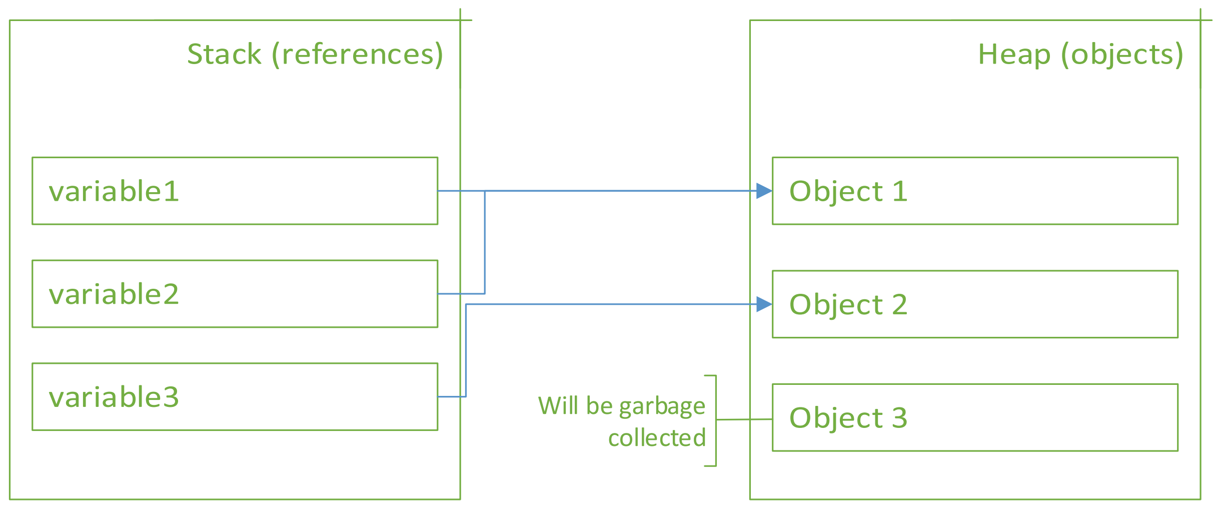

Figure 4 There are three variables on the stack, and three objects on the heap. Two variable refer to the same object. One object is not referred to, and will be collected by the garbage collector

Bytecode verifications

Bytecode is loaded into the Java Virtual Machine by class files. Apart from interpretable bytecode (the methods of the class), this file also contains other things, such as the Class File Format version, the class name, interfaces… I will not discuss the bytecode format here, as that is not really important here.

The Java Virtual Machine performs checks on the bytecode before it even begins interpreting it. These are some examples of checks being performed (not a complete list):

"

- Branches must be within the bounds of the code array for the method. […]
- No instruction can access or modify a local variable at an index greater than or equal to the number of local variables that its method indicates it allocates.
- All references to the constant pool must be to an entry of the appropriate type. (For example, the instruction getfield must reference a field.)
- The code does not end in the middle of an instruction.
- Execution cannot fall off the end of the code.

[...]

” (Section 4.10.2.2. The Bytecode Verifier, Oracle, 2013 [25])

Security Manager

Calling System.exit() allows a Java program to exit the Java Virtual Machine. This may be unwanted behaviour; imagine running a third-party plugin for a Java application: if this plugin calls the System.exit() method, it brings down the entire JVM, and thus the entire application.

The SecurityManager (java.lang.SecurityManager) was created to address problems like this. It allows restricting what Java code can do. For example, a SecurityManager implementation may stop a Java application from exiting the JVM:

```
class MySecurityManager extends SecurityManager {
  override def checkExit(status: Int): Unit = throw new SecurityException()
}
System.setSecurityManager(new MySecurityManager)
```

If code were to call System.exit() now, a SecurityException would be thrown and the Java Virtual Machine would not exit.

checkExit() is just one of the many methods in the SecurityManager. Other methods allow restricting access to files, the network and the class loader.

It's important to note this the Java SecurityManager is not a Java Virutal Machine feature, but a Java feature. The Security Manager is fully implemented in Java. For example, System.exit() will ask the SecurityManager if it's safe to exit. This is implemented in Java, and the JVM does not have a notion of the SecurityManager.

Scripting languages on the Java Virtual Machine: JSR-223

Java Specification Request 223 (Oracle Corporation, 2006 [26]) proposed a new Java bytecode to better support dynamically typed languages, such as JavaScript. This proposal, which was accepted

[25] Oracle Corporation. (2013, February 28). Java Virtual Machine Specification, Java SE 7 Edition. Retrieved from http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.10.2.2

[26] Oracle Corporation. (2006, December 11). JSR 223: Scripting for the JavaTM Platform. Retrieved from https://www.jcp.org/en/jsr/detail?id=223

and is now implemented, allows more flexible invocation of methods in the JVM. An important benefit is that it makes the JVM able to optimize dynamic languages.

Applying a security model to code running in a virtual machine

In this section, we will apply a security model to code running in a virtual machine. While the focus is on process virtual machines, this can also be applied to system virtual machines. Although a virtual machine tries to encapsulate code, there are still some risks associated with running code on a virtual machine.

Threats when running code in a virtual machine

A threat is anything that can exploit vulnerabilities in a system. In this specific case, the system is a virtual machine, and logically, the primary threat is the code running inside the virtual machine.

There are also secondary threats: code running on the same level as the virtual machine could modify the virtual machine, power could fail... But I will concentrate on the code running inside the virtual machine and how it can try to exploit its environment in this research paper.

Vulnerabilities of running code in a virtual machine

Confidentiality

Network access

Network access can lead to a breach of confidentiality. For example, a lot of applications "phone home" after installation. This allows the application's developers to know where the application was installed. But it can go further than that, the application may collect statistics on the system it is running on, and send these. Given no network access, it would be next to impossible to send information like that.

You do not always want to give hosts of a VM unlimited access to the network. Code running in a virtual machine can connect to external networks, which can bring a whole slew of issues with it, as guest code can use the host network connection, making it look like the host made the requests. For example, rogue code could try to perform a denial of service attack on a server.

File system access

File system access may lead to a breach of confidentiality. Files on a system may include sensitive information that should be safe-guarded. For example, UNIX-like operating systems store password information in the `/etc/shadow` file and users may store their SSH keys in the `~/.ssh` directory. If an attacker were to steal the user's password or SSH key, they may impersonate the user.

An example of this vulnerability is CVE-2007-1744. This vulnerability allowed a system virtual machine running on VMware to write arbitrary files on the host.

Integrity

File system access

I already discussed that file system access may constitute a breach of confidentiality, but it may also be threaten the integrity of data. For example, code running inside a virtual machine may corrupt files by writing random data to them.

Availability

Denial of Service through CPU resource starvation

When the virtual machine runtime does not limit the CPU time the guest code gets, the guest can effectively cause a denial of service for the host by drawing all processing power to itself. An example would be running `while(1){}` on Node.js (on the V8 JavaScript engine): this will take up the processing power of one CPU (as Node.js on V8 is single-threaded).

Denial of Service through RAM resource starvation

If the virtual machine runtime does not properly limit the memory allocations the code running in the virtual machine can make, the code may cause a denial of service attack by allocating huge amounts of memory without releasing them.

Denial of Service through file system resource starvation

When the virtual machine runtime does not properly limit the file system to the code running in the virtual machine, there is a possibility to cause a denial of service attack.

One course of action is to open a lot of files. This will cause a lot of file descriptors to be created, one for each opened file. Most operating systems have a limit on the number of file descriptors. When the limit is reached, no more file descriptors can be created, so opening files will fail. Luckily, modern operating systems have a per-process limit on the number of file descriptors, so a single process can not cause a denial of service attack for the whole system. But a virtual machine runtime may be implemented as a single process running multiple virtual machines, so one virtual machine may cause a denial of service attack for others.

Another and easier possibility is for code running on a virtual machine to read or write a lot of data to a disk. This cause I/O-bound applications that read or write data and the same disk to slow down.

JavaScript on the JVM

Rhino

Rhino is a mature implementation of JavaScript on the JVM. It's developed by Mozilla and was started in 1997 (Mozilla, 2015 [27]). On its information site, Mozilla says “[Rhino] is typically embedded into Java applications to provide scripting to end users” (Mozilla, n.d.)[28].

Rhino has been the traditional choice for running JavaScript on the Java Virtual Machine. As the oldest JavaScript engine for the JVM, there is a ton of information available on it.

To run a script on Rhino, you must first create a `Context` and initialize the standard JavaScript objects (such as `Date`, `String`, `Array`…). Then you can evaluate a script, by passing it either a Java string, or a Java `Reader` (an abstract class used for reading character streams). An explanation of the most common Rhino classes is further in this section.

```
val cx: Context = Context.enter()
val scope: Scriptable = cx.initStandardObjects()
val result: Any = cx.evaluateString(scope, "var test = 1;", "MySource", 1, null)
```

Context

Context is a class representing the runtime context of an executing script (Brail, 2016 [29]). You do not create a `Context` (although you can, but the constructor is deprecated), instead you use the `enter()` factory method. You cannot share Context's across threads, but you can instantiate new ones.

[27] Mozilla. (2015, July 26). Rhino history. Retrieved February 22, 2016, from https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino/History
[28] Mozilla. (n.d.). Rhino. Retrieved February 29, 2016, from https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino
[29] Brail, G. (2016, March 19). Context.java. Retrieved from https://github.com/mozilla/rhino/blob/15f57d5785fe3da878eb269ebcb6c3a4873c4a98/src/org/mozilla/javascript/Context.java#L29

Scriptable

Scriptable is an interface that is implemented by all JavaScript object in the Rhino source code (Boyd & Bukanov, 2011 [30]). It provides methods that can be called on all JavaScript objects. For example, it is possible to use brackets on all JavaScript objects. In other languages, this is usually reserved for use on arrays, but they can be used on JavaScript objects, arrays, strings... This can be implemented by implementing the get() method.

An example implementation of Scriptable is NativeString (Brail, 2016 [31]), which maps a Java string to a JavaScript string.

Initializing standard objects

JavaScript has some objects available in the global scope. Examples include the eval function, the Array object, undefined... These have to be initialized, which can be done using the Context#initStandardObject() method, which returns a ScriptableObject containing all these properties. This can then be passed to evaluateString/evaluateReader when you want to execute a script.

It's possible to get a full list of what Rhino provides by creating a scope with all the standard objects (using Context#initStandardObjects()), then calling getAllIds() on the resulting ScriptableObject:

```
val all = Context.enter().initStandardObjects().getAllIds
```

This returned an object with 55 elements on the system I tested on.

Nashorn

Nashorn is the new kid on the block when it comes to JavaScript engines on the JVM. Its goals explicitly mention high performance and the use of newer technologies (OpenJDK, n.d. [32]) Like Rhino, Nashorn aims to make it possible to easily embed JavaScript in JVM applications, but also make it possible to run standalone JavaScript on the Java Virtual Machine.

[30] Boyd, N., & Bukanov, I. (2011, June 1). Scriptable.java. Retrieved from https://github.com/mozilla/rhino/blob/master/src/org/mozilla/javascript/Scriptable.java#L12
[31] Brail, G. (2016, February 19). Context.java. Retrieved from https://github.com/mozilla/rhino/blob/master/src/org/mozilla/javascript/NativeString.java
[32] OpenJDK. (n.d.). OpenJDK: Nashorn. Retrieved February 29, 2016, from http://openjdk.java.net/projects/nashorn/

Others

Rhino and Nashorn are the two big JavaScript engines on the Java Virtual Machine. But there are also other implementations. However, these are mostly research projects, incomplete or abandoned projects. The most promising alternative JavaScript engine was DynJS. At nearly 2500 commits (DynJS, n.d. [33]), I would call this a mature project. It does not completely comply to the JavaScript specification however (Campos, 2016 [34]) and the project was sadly abandoned (Campos, 2016 [35]).

Node.js on the JVM

Trireme

Trireme is a project that implements the Node.js APIs on top of Rhino. Trireme is a mature project, and Apigee, the company that develops it, uses it in production (Brail, n.d. [36]). It has some security built in, including HTTP and filesystem sandboxing and execution time limiting.

To run a script on Trireme, several classes should be understood.

NodeEnvironment

The `NodeEnvironment` is the class that sets up a thread pool, initializes a context, set the stack trace format, and initializes a few other things (Brail & Whitlock, n.d. [37]). The Trireme developers remark that you usually create one `NodeEnvironment` per thread, but you can create more if you want to.

NodeScript

A `NodeScript` represents a single JavaScript source that can be run on Trireme. To create a `NodeScript`, you always need a `NodeEnvironment`. There are constructors provided for the creation of a script from a Java File, a source represented by a Java String, and even a constructor which allows the creation of a REPL.

To execute a `NodeScript`, two methods are provided on `NodeScript` instances: `execute()` and

[33] DynJS. (n.d.). dynjs/dynjs. Retrieved March 19, 2016, from https://github.com/dynjs/dynjs
[34] Campos, D. (2016, August 6). Get > 90% spec compliance on IR #128. Retrieved March 19, 2016, from https://github.com/dynjs/dynjs/issues/128
[35] Campos, D. (2016, January 28). this project is unmaintained atm. Retrieved March 19, 2016, from https://github.com/dynjs/dynjs/commit/d89b684c317f5668cb5e982aade35ec39b235599
[36] Brail, G. (n.d.). trireme/samples/apigee-edge-like-runner/. Retrieved March 19, 2016, from https://github.com/apigee/trireme/tree/master/samples/apigee-edge-like-runner
[37] Brail, G., & Whitlock, J. (n.d.). NodeEnvironment.java. Retrieved March 19, 2016, from https://github.com/apigee/trireme/blob/master/core/src/main/java/io/apigee/trireme/core/NodeEnvironment.java

executeModule(). Both will run the script on the Rhino JavaScript engine, but execute() runs the script, while executeModule() is used to execute Node.js modules. After executeModule() is done executing the script, it will return the value of module.exports. This is how Node.js modules export functions, objects and other values for inclusion in other scripts.

NodeStatus

After executing a script, you get back a NodeStatus. NodeStatus is a very simple class, containing just an exit code and possible a Throwable (an exception) representing the cause of the failure, and a few helper methods. The helper methods allow to check if the script completed successfully (isOk(), isCancelled() and getExitCode()) and allow check for failures (hasCause() and getCause()).

NodeStatus is what the execute() and executeModule() functions of NodeScript return.

Faking asynchronicity

Java is traditionally a synchronous programming language, a lot of operations block execution until they are done. With the introduction of java.nio in Java 1.4, Java got non-blocking I/O support. However, there are still a lot of blocking methods in the Java APIs.

This presents a problem when thinking about Node.js interoperability: Node.js is built on the principle that nothing should block. For example, file access and network access are completely asynchronous in Node.js.

When implementing the Node.js APIs, the Trireme developers found this to be a problem: you can't make synchronous interfaces asynchronous (bar completely rewriting them). The solution is simple: a thread pool is provided, in which these operations can block. By off-loading blocking to a thread pool, the operations can appear asynchronous to the code running in Trireme. In the source code, the operations that utilise the thread pole are specified as "[...] file I/O, at least in Java 6, plus DNS queries and certain SSLEngine functions.". (Brail & Whitlock, n.d. [38])

There's an obvious limit: the size of the thread pool. If the number of operations is higher than the

[38] Brail, G., & Whitlock, J. (n.d.). NodeEnvironment.java. Retrieved March 19, 2016, from https://github.com/apigee/trireme/blob/master/core/src/main/java/io/apigee/trireme/core/NodeEnvironment.java

thread pool size, the operations will fail.

Rowboat

Rowboat is a project started by the same company that developed Trireme, Apigee (Apigee, n.d. [39]). Its goal is to eventually replace Trireme. It uses Nashorn (unlike Rowboat, which uses Rhino). The reason Rowboat is being developed is twofold: it's creators believe depending on the newer Nashorn engine will be more future-proof because a team is working on Nashorn, which ensures it keeps up with the latest JavaScript features, and because Nashorn is "[...] supposed to be faster" (Apigee, n.d. [40]).

The project is very incomplete (Apigee, n.d. [41]) and has not been updated in 9 months. One author commented "I haven't done a ton of work on Rowboat and neither has anyone else that I know of" (Taggart & Brail, 2015 [42]). For these reasons, I believe Rowboat is still too immature to be reviewed.

Nodyn

Nodyn is a Node.js implementation on top of the DynJS JavaScript engine, so it's not built on Rhino or Nashorn. It's a project that was sponsored by Red Hat (Yegulalp, 2014 [43]), but unfortunately, it's no longer being maintained (Ball, 2015 [44]). Therefore, I will not discuss it here.

Avatar.js

Avatar.js is a Node.js implementation on top of the Nashorn JavaScript engine. (Oracle Corporation, n.d. [45]). Unfortunately, it seems like it's no longer being maintained, the last commit on the Git repository was over a year ago and Oracle officially pronounced the project as being "on hold" (Köbler, 2015 [46])

[39] Apigee. (n.d.). apigee/rowboat. Retrieved March 19, 2016, from https://github.com/apigee/rowboat
[40] Apigee. (n.d.). apigee/rowboat: Performance. Retrieved March 19, 2016, from https://github.com/apigee/rowboat#performance
[41] Apigee. (n.d.). apigee/rowboat. Retrieved March 19, 2016, from https://github.com/apigee/rowboat#status
[42] Taggart, T., & Brail, G. (2015, December 18). Can rowboat run the TypeScript Compiler? #1. Retrieved March 19, 2016, from https://github.com/apigee/rowboat/issues/1
[43] Yegulalp, S. (2014, March 3). Node.js arrives for the JVM. Retrieved from http://www.infoworld.com/article/2610123/javascript/node-js-arrives-for-the-jvm.html
[44] Ball, L. (2015, June 12). Update README.md. Retrieved from https://github.com/nodyn/nodyn/commit/7e73bd692664110c381f663561767e78645dd23b
[45] Oracle Corporation. (n.d.). Project Avatar Essentials. Retrieved March 19, 2016, from https://avatar.java.net/essentials.html
[46] Köbler, N. (2015, February 12). Current Status of Oracle's Project Avatar. Retrieved from http://www.n-k.de/2015/02/current-status-of-oracles-project-avatar.html

Untrusted JavaScript on the JVM

Plain JavaScript

Rhino

I will study the Rhino Sandbox repository, created by Java Delight (Java Delight, n.d. [47]).

This project aims to create a sandbox around Rhino, so you can run untrusted code on the JVM. It adds some features on top of Rhino to provide a secure environment.

Preventing access to and creation of Java classes

To investigate secure execution of JavaScript on the JVM with Rhino, I will investigate both features provided by Nashorn itself, and the Delight Rhino Sandbox (Java Delight, n.d. [48]).

Vanilla Rhino

Rhino has both a built-in way to access Java classes from JavaScript, and a built-in way to limit this behaviour. Preventing access is very flexible, and works by the means of a ClassShutter class. Such a class implements the ClassShutter interface. This is a very simple interface that has only one method: visibleToScripts(), which a string representing the full class name (including packages), and returns a boolean. If it returns true, then JavaScript running on Rhino may access and create new objects of the specified class, if false, it may not.

The following ClassShutter implementation prevents access to all classes from JavaScript:

```
import org.mozilla.javascript.ClassShutter
class MyClassShutter extends ClassShutter {
  override def visibleToScripts(fullClassName: String): Boolean = false
}
```

When a JavaScript script being executed by a Rhino context which has this ClassShutter, Rhino will throw an exception, and the execution of the Script will be interrupted.

[47] Java Delight. (n.d.). javadelight/delight-rhino-sandbox. Retrieved March 19, 2016, from https://github.com/javadelight/delight-rhino-sandbox
[48] Java Delight. (n.d.). javadelight/delight-rhino-sandbox. Retrieved March 19, 2016, from https://github.com/javadelight/delight-rhino-sandbox

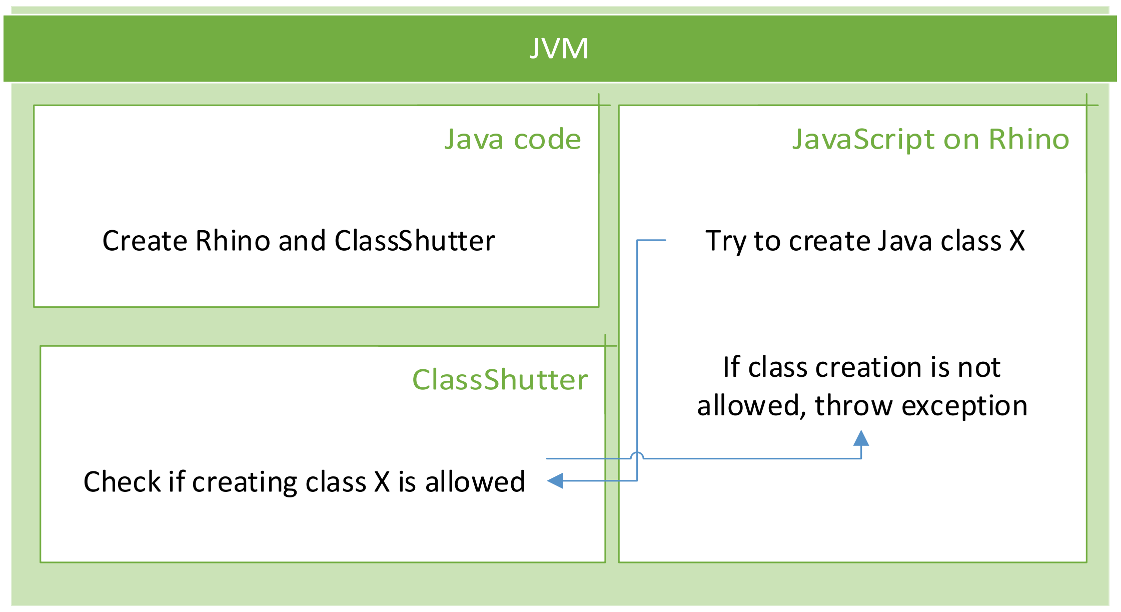
JVM
Java code
Create Rhino and ClassShutter
JavaScript on Rhino
Try to create Java class X
ClassShutter
Check if creating class X is allowed
If class creation is not allowed, throw exception

Figure 5 Java code sets up Rhino with a ClassShutter. When JavaScript on Rhino wants to create a Java class, Rhino asks the ClassShutter if creating the class is allowed. If not, an exception is thrown.

Java Delight Rhino Sandbox

The Java Delight Rhino Sandbox blocks access to all Java classes by default. That means you don't have to take any extra steps to prevent JavaScript running on the Java Delight Rhino Sandbox to access Java classes.

If you do want to allow JavaScript access to Java classes, you need to add them to an internal white-list, by using the inject() method on the RhinoSandbox instance you will execute the JavaScript on. The inject method takes two parameters: a name and a Java object instance. The name, which must be a string, will be the name of the variable that will made be available to the JavaScript executing on the sandbox. For example, if you pass in myJavaList as the name, and an ArrayList instance as the object instance, you will be able to call myJavaList.add() from JavaScript.

Preventing thread creation

To prevent Rhino from creating threads, you just have to prevent JavaScript running on Rhino access to classes that can create threads. The obvious one is java.lang.Thread. But that's not all: threads can also be created by other classes. Rhino will only disallow the JavaScript running from accessing the Thread class, but it won't prevent the JavaScript from calling another class, that then creates a thread.

In fact, you could even create a Java class that extends Thread, and call that from JavaScript:

```
public class MyThread extends java.lang.Thread {}
```

Even if you prevent access to `java.lang.Thread`, you will be able to create `MyThread`, which is essentially the same thing.

That's why allowing access to classes from JavaScript should be used very sparingly when used with untrusted JavaScript code. I suggest a white-list of classes, if you absolutely must provide access to Java classes. These classes should be vetted to ensure that they allow possibility for harmful behaviour.

As mentioned in the previous section, the Java Delight Rhino Sandbox blocks access to all Java classes by default. That means you don't have to take any extra steps to prevent JavaScript running on the Java Delight Rhino Sandbox from creating threads.

Preventing CPU resource starvation

Rhino has a built-in way to prevent scripts from taking too much CPU time. It can limit the number of instructions a script can execute. This is fairer than setting an execution time limit: I/O-bound applications do not take much CPU time, but they do take a lot of real execution time.

Setting an instruction limit is as easy as calling `setInstructionObserverThreshold` on a `Context`:

```
val cx: Context = Context.enter()
cx.setInstructionObserverThreshold(5000)
```

Scripts executing using this `Context` will now only be able to execute a maximum of 5000 instructions.

The Java Delight Rhino Sandbox also supports this. The instruction limit is set on the sandbox:

```
RhinoSandbox sandbox = RhinoSandboxes.create();
sandbox.setInstructionLimit(5000);
```

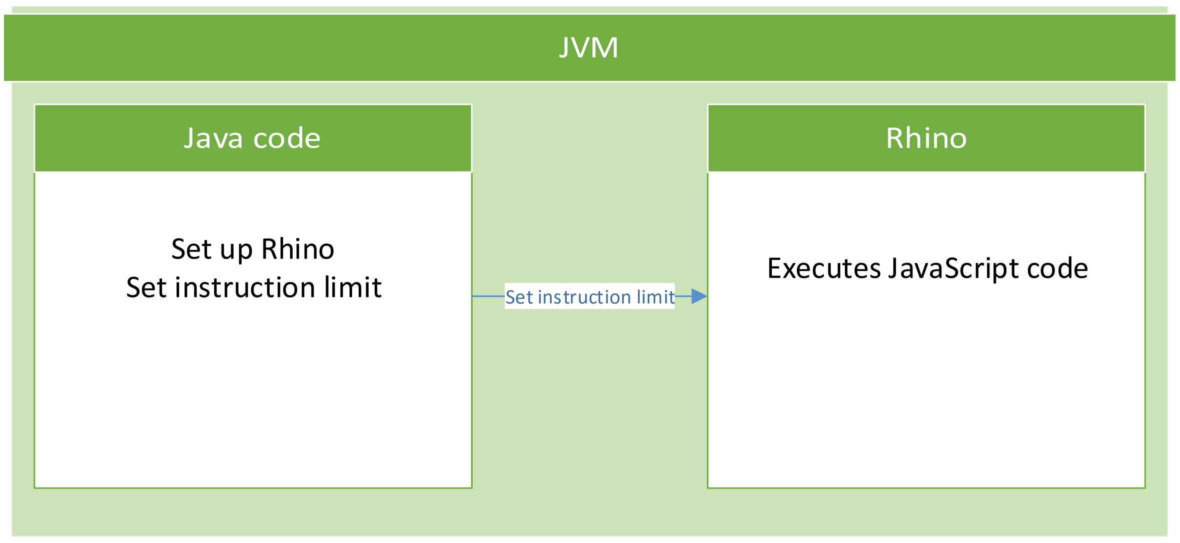

Figure 6 Java code sets up Rhino and specifies and instruction limit. When the JavaScript running on Rhino exceeds the limit, it gets terminated.

Preventing file system access

There are no built-in reliable ways to limit access to the file system. As explained in the section on thread creation, you have to rely on a white-list of classes that JavaScript running on Rhino is allowed to access.

Preventing network access

There are no built-in reliable ways to limit access to the network. As explained in the section on thread creation, you have to rely on a white-list of classes that JavaScript running on Rhino is allowed to access.

Preventing RAM resource starvation

If JavaScript running on Rhino allocates a lot of objects, the Java Virtual Machine may run out of memory. There is no reliable way to prevent this.

An example script that will make the JVM run out of heap space:

```
var a = [];
while(true) { a.push("aoeu") }
```

This will start an endless loop, that grows the array a with every loop body execution. Eventually, the JVM will run out of heap space, and give up with an `OutOfMemoryException`.

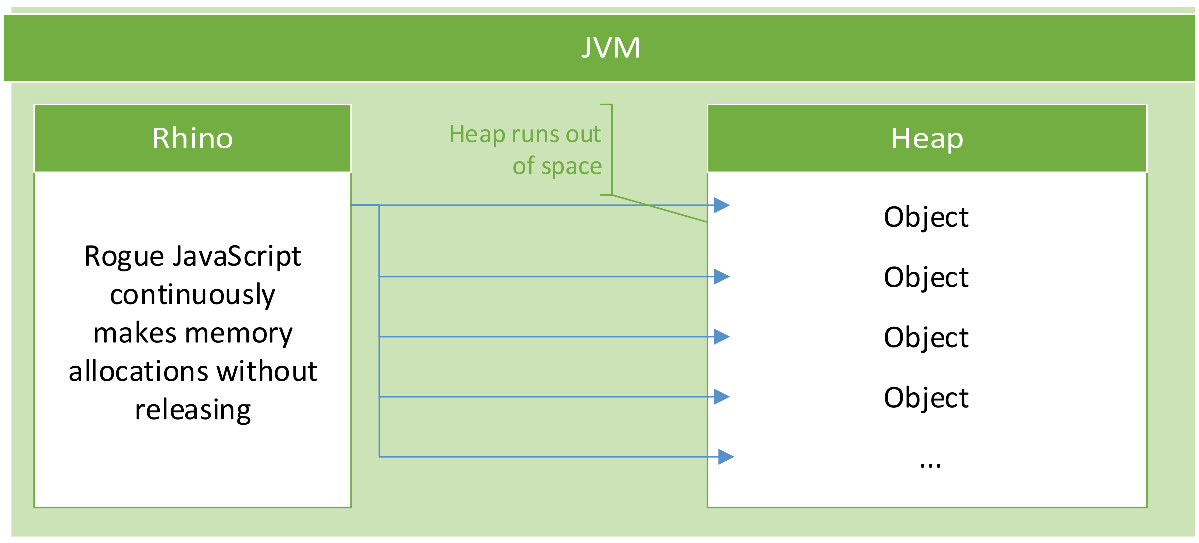
JVM
Rhino
Rogue JavaScript continuously makes memory allocations without releasing
Heap runs out of space
Heap
Object
Object
Object
Object
...

Figure 7 Malicious JavaScript running on Rhino makes lots of memory allocations without releasing them. This makes the heap run out of space. Eventually, the JVM crashes.

Conclusions on running JavaScript on Rhino

Vanilla Rhino is not built from the ground up to run untrusted JavaScript. However, using a `ClassShutter` that blocks access to all Java classes will get you far. You still have to limit the instructions that Rhino can run, but you cannot limit allocations, so it's possible for JavaScript to bring down the JVM. For this reason, I do not deem it safe to run untrusted JavaScript code on Rhino.

Nashorn

To investigate secure execution of JavaScript on the JVM with Nashorn, I will investigate both features provided by Nashorn itself, and the Delight Nashorn Sandbox (Java Delight, n.d. [49]).

The Delight Nashorn Sandbox is not complete, but it's a very active project, and I believe it will grow to replace the Delight Rhino Sandbox.

Preventing access to and creation of Java classes

Nashorn has a built-in way to limit JavaScript scripts access to Java classes. Java classes can be created from within Nashorn:

```
js> new java.lang.String("Hello from JavaScript!")
```

[49] Java Delight. (n.d.). javadelight/delight-nashorn-sandbox. Retrieved March 19, 2016, from https://github.com/javadelight/delight-nashorn-sandbox

```
Hello from JavaScript!
```

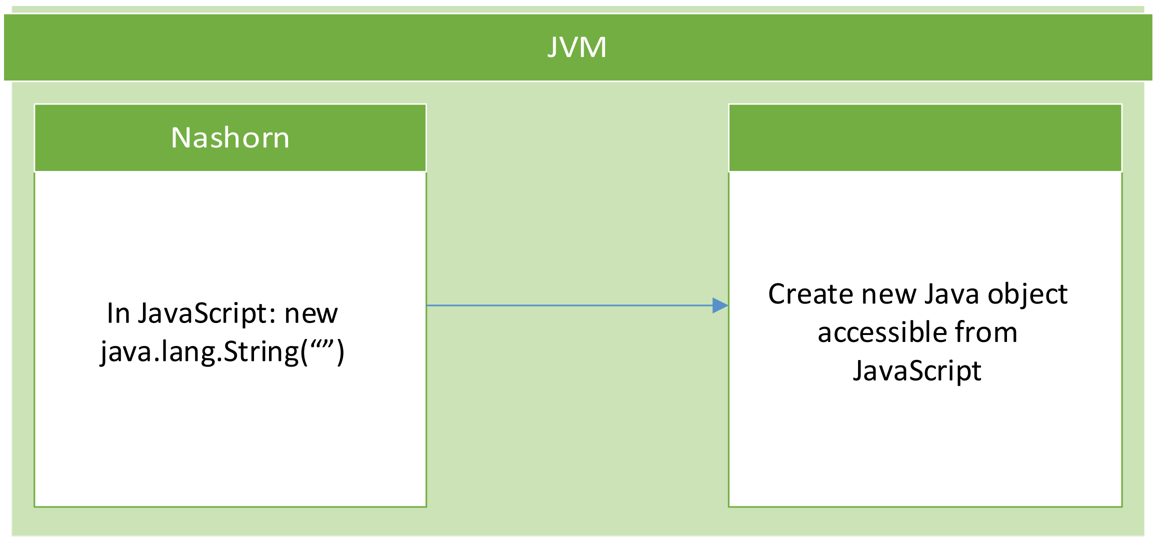
JVM
Nashorn
In JavaScript: new java.lang.String("")
Create new Java object accessible from JavaScript

Figure 8 Creating a Java object from JavaScript on Nashorn

We want to limit this ability for the reasons explained previously. Nashorn provides a built-in way to do this: the ClassFilter interface. This interface contains a single method you should implement. The method is `boolean exposeToScripts(String className)`. This pretty self-explanatory method takes in the fully qualified name of a class (for example, `java.lang.String`) and returns `true` if the class should be accessible from JavaScript, and `false` otherwise.

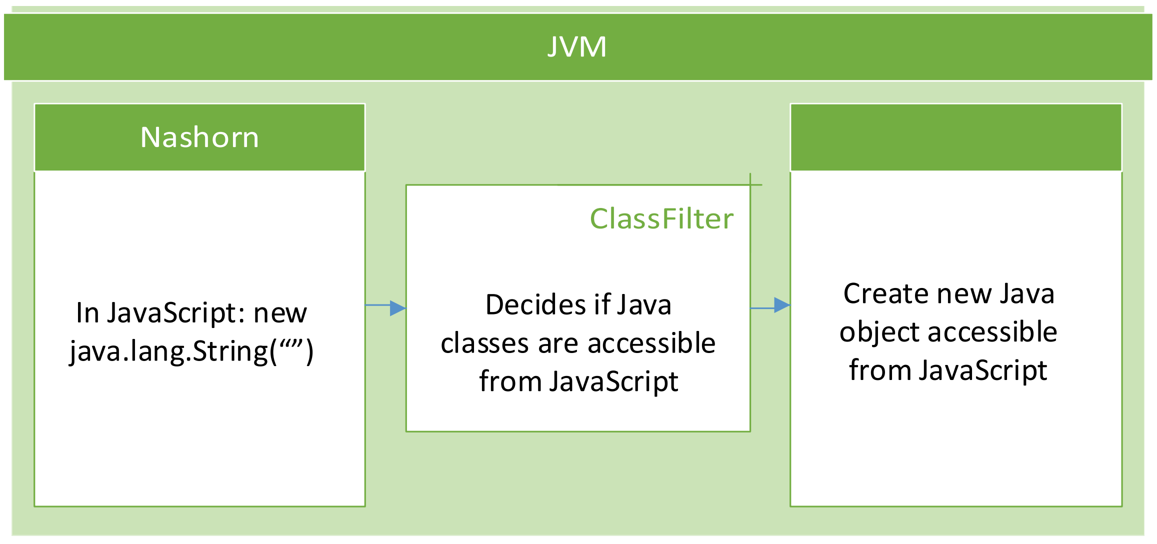
JVM
Nashorn
In JavaScript: new java.lang.String("")
ClassFilter
Decides if Java classes are accessible from JavaScript
Create new Java object accessible from JavaScript

Figure 9 A ClassFilter decides whether or not JavaScript on Nashorn may access Java classes

The example `ClassFilter` implementation below allows JavaScript running on Rhino to create and access all classes in the java.lang packages (including subpackages).

```
import jdk.nashorn.api.scripting.ClassFilter
class MyClassFilter extends ClassFilter {
  def exposeToScripts(className: String): Boolean = className.startsWith("java.lang.")
}
```

When trying to access a class that is not in (a subpackage of) the java.lang package, Nashorn will throw a RuntimeException wrapping a ClassNotFoundException.

The Delight Nashorn Sandbox provides its own wrapper for this. Instead of implementing your own ClassFilter, you add allowed classes to the sandbox.

```
sandbox.allow(java.io.File.class);
```

Internally, the sandbox keeps a list of all the allowed classes. When it creates the Nashorn engine, it creates a new SandboxClassFilter with all the Java classes that have been allowed. The SandboxClassFilter is Delight Nashorn Sandbox' implementation of ClassFilter, it's a very simple class that takes a list of classes, and exposeToScripts() will return true if the asked class is in the list of allowed classes. This means the sandbox is rather inflexible: since it does not allow you to pass in your own ClassFilter, you can not specify custom logic (such as the "if the class name starts with java.lang." we used above) and you have to pass in every class you want to be accessible individually.

Preventing thread creation

To prevent Nashorn from creating threads, you just have to prevent JavaScript running on Nashorn access to classes that can create threads. The obvious one is java.lang.Thread. But that's not complete: threads can also be created by other classes. Nashorn will only disallow the JavaScript running from accessing the Thread class, but it won't prevent the JavaScript from calling another class, that then creates a thread.

In fact, you could even create a Java class that extends Thread, and call that from JavaScript:

```
public class MyThread extends java.lang.Thread {}
```

Even if you prevent access to java.lang.Thread, you will be able to create MyThread, which is essentially the same thing.

That's why allowing access to classes from JavaScript should be used very sparingly when used with untrusted JavaScript code. I suggest a white-list of classes, if you absolutely must provide access to Java classes. These classes should be vetted to ensure that they allow possibility for harmful behaviour.

As mentioned in a previous section, the Delight Nashorn Sandbox protects you against this by requiring you to explicitly white-list allowed classes.

Preventing CPU resource starvation

Nashorn does not have a built-in way to limit CPU time spent in the engine like Rhino does. This is a significant disadvantage. That means executing a script containing an endless loop (e.g. `while(1) {}`) will block the execution indefinitely.

The Delight Nashorn Sandbox luckily provides a solution for this. On the Sandbox class, there is a method called `setMaxCPUTime()` that allows to set a maximum execution time in milliseconds for the script. When executing the script, the Delight Nashorn Sandbox will execute the script on one thread, and monitor the executing thread on another thread. If the executing thread has been executing for too long, the Delight Nashorn Sandbox will kill it.

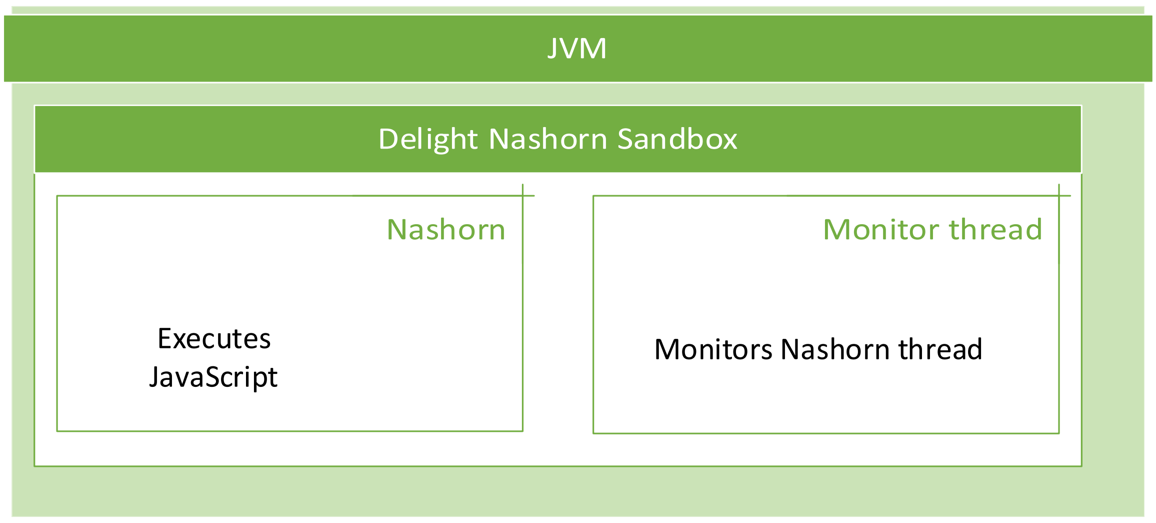

Figure 10 The Delight Nashorn Sandbox can monitor Nashorn. If JavaScript execution on Nashorn takes too long, the monitor thread will kill it.

Preventing file system access

There are no built-in reliable ways to limit access to the file system. As explained in the section on thread creation, you have to rely on a white-list of classes that JavaScript running on Nashorn is allowed to access.

Preventing network access

There are no built-in reliable ways to limit access to the network. As explained in the section on thread creation, you have to rely on a white-list of classes that JavaScript running on Nashorn is allowed to access.

Preventing access to Nashorn's built-in functions

Nashorn adds in extra functions, in addition to JavaScript built-in objects. These functions can be used to read from and print to the standard output, to load a script from a remote location, and to exit the current running process.

There is no way to turn this off when using Nashorn programmatically from Java or another JVM language. One way to only to disable these functions is prepend a script to be run with a script that assigns `undefined` to these variables. This way, they cannot be called. Another way is to implement a `SecurityManager` that disallows access to what these functions call.

Conclusions on running JavaScript on Nashorn

Just like Rhino, Nashorn has a built-in way to limit access to classes. Unlike Rhino, there is no built-in way to limit the maximum number of instructions. If you use the Delight Nashorn Sandbox, you can set a time limit. Nashorn also adds in extra methods, next to the default JavaScript objects. These methods allow to perform dangerous operations, e.g. shut down the JVM. This leads me to conclude that Nashorn is not ready to run untrusted JavaScript.

Node.js

If you want to run Node.js on the Java Virtual Machine, you bring in more risks than plain JavaScript. The reason is that Node.js defines a set of core APIs, including APIs for file system and network access (Node.js Foundation, n.d. [50]). Plain JavaScript does not provide such access to the

[50] Node.js Foundation. (n.d.). Node.js v5.9.0 Documentation. Retrieved March 19, 2016, from https://nodejs.org/api/fs.html

file system or to the network.

Trireme

Preventing access to and creation of Java classes

Trireme has a default ClassShutter implementation which prevents access to all classes. This is a sensible default: no Java classes can be accessed or created from JavaScript running on Trireme.

If you do want to provide JavaScript running on Trireme access to Java classes, you need to implement your own ClassShutter implementation and provide this to Trireme. To provide this ClassShutter to Trireme, you must create a Sandbox. A Sandbox holds information on what Trireme scripts are allowed to access.

An example implementation is shown below.

```
val env: NodeEnvironment = new NodeEnvironment
env.setSandbox((new Sandbox).setExtraClassShutter(new ClassShutter {
  override def visibleToScripts(fullClassName: String): Boolean =
fullClassName.startsWith("java.io.")
}))
```

A new Sandbox is created, on which a ClassShutter is set up. We create a custom ClassShutter that allows access to all classes in java.io and its subpackages. The new Sandbox is then set on the NodeEnviroment, scripts that are now executed in this new environment will have access to the java.io packages.

Preventing thread creation

As Trireme prevents access to Java classes from JavaScript code by default, no other steps should be taken to prevent JavaScript access to thread creation.

While the JavaScript running on Trireme cannot create threads directly, it can create them indirectly. This risk will be explained in a section below.

Preventing CPU resource starvation

If processes are not limited, they can take up a lot of CPU time. Worse, they may not even stop. Even a simple while loop with a conditional that always evaluates to true can make the server take up a lot of CPU.

This means other processes running on the system get less CPU time. If the process executes a lot of instructions for a long period of time, this essentially means that other processes will be starved: they will get very little CPU time, if any.

To prevent this, we can limit the process. Two popular ways include limiting the CPU time and limiting the number of instructions.

Trireme only supports time limits for JavaScript scripts. There is a method called `setScriptTimeLimit()` on `NodeEnvironment` that does exactly that: it sets a time limit for the script. As arguments, it takes in a time value and a unit (e.g. seconds). When a time limit is set, it will abort execution if executing the script takes too long: execution will be interrupted and an `ExecutionException` (wrapping a `JavaScriptException` with details) will be thrown. An example of setting a script time limit:

```
env.setScriptTimeLimit(1, TimeUnit.SECONDS)
```

Preventing file system access

As Node.js includes an fs module, to access the file system, it makes sense to limit file system access. Trireme has built-in support to do this. It provides a chroot-like environment: a directory of the host system is made available to the running JavaScript code, to which it appears as the only directory that it is able to access.

Enabling this chroot-like environment is done by creating a sandbox and calling the method `setFilesystemRoot()` on it. This method takes one parameter: the path to what the root exposed to the script should be. The script will then only be able to access that root and the folders in it, it will not be able to get files or folders in a parent directory of the root. The following three lines create a `NodeEnvironment`, a sandbox that is configured to let scripts only access the target directory and applies that sandbox to the environment:

```
val env: NodeEnvironment = new NodeEnvironment
val sandbox = (new Sandbox).setFilesystemRoot("./target")
env.setSandbox(sandbox)
```

If scripts executed in this environment try to access a parent directory, an exception will be thrown. For example, the following JavaScript will throw an exception:

```
var fs = require('fs');
```

```
console.log(fs.readdirSync('..'));
```

Preventing network access

Trireme makes it possible to create a custom HTTP adapter. That way, you can provide a custom Java implementation that will be used for network access. This can be used to limit HTTP requests, one could implement an HTTP adapter that will refuse access to a certain domain, limit that number of requests, or throttle the bandwidth. Since this is not trivial, I have chosen not to inline the code to do this.

Conclusions on running JavaScript on Trireme

Trireme has the sensible default of blocking access to all Java classes from within JavaScript. This prevents a whole range of issues. It also supports CPU time-outs.

Since Trireme is built to support Node.js, it also supports Node.js libraries. This opens a range of vulnerabilities. Luckily, Trireme has supports to mitigate these. For example, file system access can be restricted to a single directory. Network access can be restricted through the use of a custom HTTP adapter. However, writing such an adapter is hard and prone to error.

Conclusion

JavaScript is a wildly popular language and is usually executed in a browser. Server-side JavaScript has enjoyed massive popularity, but few implementations. It’s possible to execute JavaScript on the Java Virtual Machine, but at this moment, it’s not safe to execute untrusted JavaScript on it.

If you want to execute JavaScript on the Java Virtual Machine, the best choice is Nashorn, as it’s officially part of Java 8 and supported by Oracle, followed by Rhino, which is older and slower. There are no other adequate JavaScript engines on the Java Virtual Machine, other engines are unsupported, deprecated or unfinished. Both Rhino and Nashorn are not suited to run untrusted JavaScript.

If you want to execute Node.js on the Java Virtual Machine, the only choice is Trireme, based on Rhino. There are no other adequate Node.js runtimes on the Java Virtual machine. Trireme is not suited to run untrusted JavaScript.

References

Apigee. (n.d.). apigee/rowboat. Retrieved March 19, 2016, from https://github.com/apigee/rowboat

Apigee. (n.d.). apigee/rowboat: Performance. Retrieved March 19, 2016, from https://github.com/apigee/rowboat#performance

Apigee. (n.d.). apigee/rowboat. Retrieved March 19, 2016, from https://github.com/apigee/rowboat#status

Ball, L. (2015, June 12). Update README.md. Retrieved from https://github.com/nodyn/nodyn/commit/7e73bd692664110c381f663561767e78645dd23b

Boyd, N., & Bukanov, I. (2011, June 1). Scriptable.java. Retrieved from https://github.com/mozilla/rhino/blob/master/src/org/mozilla/javascript/Scriptable.java#L12

Brail, G. (n.d.). trireme/samples/apigee-edge-like-runner/. Retrieved March 19, 2016, from https://github.com/apigee/trireme/tree/master/samples/apigee-edge-like-runner

Brail, G. (2016, March 19). Context.java. Retrieved from https://github.com/mozilla/rhino/blob/15f57d5785fe3da878eb269ebcb6c3a4873c4a98/src/org/mozilla/javascript/Context.java#L29

Brail, G. (2016, February 19). Context.java. Retrieved from https://github.com/mozilla/rhino/blob/master/src/org/mozilla/javascript/NativeString.java

Brail, G., & Whitlock, J. (n.d.). NodeEnvironment.java. Retrieved March 19, 2016, from https://github.com/apigee/trireme/blob/master/core/src/main/java/io/apigee/trireme/core/NodeEnvironment.java

Byrne, M. (2016, February 2). The Rise and Fall of the Java Applet: Creative Coding's Awkward Little Square. Retrieved from http://motherboard.vice.com/read/a-brief-history-of-the-java-applet

Campos, D. (2016, August 6). Get > 90% spec compliance on IR #128. Retrieved March 19, 2016, from https://github.com/dynjs/dynjs/issues/128

Campos, D. (2016, January 28). this project is unmaintained atm. Retrieved March 19, 2016, from https://github.com/dynjs/dynjs/commit/d89b684c317f5668cb5e982aade35ec39b235599

Charles Severance, M. R. (2010, February). JavaScript: Designing a Language in 10 Days. Retrieved from https://www.computer.org/csdl/mags/co/2012/02/mco2012020007.pdf

Ched Perrin, M. R. (2009, July 7). Understanding risk, threat, and vulnerability. Retrieved from http://www.techrepublic.com/blog/it-security/understanding-risk-threat-and-vulnerability/

Dahl, R. (2009, March 3). Major refactoring: program name now "node". Retrieved from https://github.com/nodejs/node-v0.x-archive/commit/19478ed4b14263c489e872156ca55ff16a07ebe0

DynJS. (n.d.). dynjs/dynjs. Retrieved March 19, 2016, from https://github.com/dynjs/dynjs

ECMA International. (1997, June). ECMAScript: A general purpose, cross-platform programming language. Retrieved from http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%201st%20edition,%20June%201997.pdf

Facebook. (n.d.). HHVM. Retrieved March 17, 2016, from http://hhvm.com/

Garrett, J. J. (2005, February 18). Ajax: A New Approach to Web Applications. Retrieved from http://adaptivepath.org/ideas/ajax-new-approach-web-applications/

Holowaychuk, M. R. (2009, June 26). Initial commit (expressjs/expressjs). Retrieved from https://github.com/expressjs/express/commit/9998490f93d3ad3d56c00d23c0aa13fac41c3f6b

Java Delight. (n.d.). javadelight/delight-rhino-sandbox. Retrieved March 19, 2016, from https://github.com/javadelight/delight-rhino-sandbox

Java Delight. (n.d.). javadelight/delight-nashorn-sandbox. Retrieved March 19, 2016, from https://github.com/javadelight/delight-nashorn-sandbox

Krill, P. (2016, January 20). Node.js welcomes Microsoft's Chakra JavaScript engine. Retrieved from http://www.infoworld.com/article/3024271/javascript/nodejs-welcomes-microsoft-chakra-javascript-engine.html

Köbler, N. (2015, February 12). Current Status of Oracle's Project Avatar. Retrieved from http://www.n-k.de/2015/02/current-status-of-oracles-project-avatar.html

Microsoft Corporation. (1998, September). Using VBScript and JScript on a Web Page. Retrieved from https://msdn.microsoft.com/en-us/library/aa260861%28v=vs.60%29.aspx

Mozilla. (n.d.). Rhino. Retrieved February 29, 2016, from https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino

Mozilla. (2015, July 26). Rhino history. Retrieved February 22, 2016, from https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino/History

Node.js Foundation. (n.d.). Node.js. Retrieved March 17, 2016, from https://nodejs.org/en/

Node.js Foundation. (n.d.). Node.js v5.9.0 Documentation. Retrieved March 19, 2016, from https://nodejs.org/api/fs.html

OpenJDK. (n.d.). OpenJDK: Nashorn. Retrieved February 29, 2016, from http://openjdk.java.net/projects/nashorn/

Oracle Corp.. (2014, December 12). Nashorn Architecture and Performance Improvements in the Upcoming JDK 8u40 Release (Nashorn) [Blog post]. Retrieved from https://blogs.oracle.com/nashorn/entry/nashorn_performance_work_in_the

Oracle Corp.. (2015, February 13). The Java® Virtual Machine Specification Java SE 8 Edition. Retrieved from http://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf

Oracle Corp.. (2015, February 13). The Java® Language Specification Java SE 8 Edition. Retrieved from http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

Oracle Corporation. (n.d.). JSRs: Java Specification Requests. Retrieved February 22, 2016, from https://jcp.org/en/jsr/overview

Oracle Corporation. (n.d.). ArrayIndexOutOfBoundsException (Java Platform SE 8). Retrieved February 28, 2016, from https://docs.oracle.com/javase/8/docs/api/java/lang/ArrayIndexOutOfBoundsException.html

Oracle Corporation. (n.d.). javac - Java programming language compiler. Retrieved March 19, 2016, from http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html

Oracle Corporation. (n.d.). Chapter 4. The class File Format. Retrieved March 19, 2016, from http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.10.1.9.checkcast

Oracle Corporation. (n.d.). Project Avatar Essentials. Retrieved March 19, 2016, from https://avatar.java.net/essentials.html

Oracle Corporation. (1997, December 19). Writing Server-Side JavaScript Applications with Enterprise Server 3.x. Retrieved from https://docs.oracle.com/cd/E19957-01/816-5653-10/816-5653-10.pdf

Oracle Corporation. (2006, December 11). JSR 223: Scripting for the JavaTM Platform. Retrieved from https://www.jcp.org/en/jsr/detail?id=223

Oracle Corporation. (2013, February 28). Java Virtual Machine Specification, Java SE 7 Edition. Retrieved from http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.10.2.2

Rubenking, N. J. (2013, March 1). How to Disable Java. Retrieved from http://www.pcmag.com/article2/0,2817,2414191,00.asp

Srinivas, R. N. (2000, July 28). Java security evolution and concepts, Part 2. Retrieved from http://www.javaworld.com/article/2076135/java-security/java-security-evolution-and-concepts--part-2.html

Sun Microsystems, Inc.. (1998, January 15). Security Tools: Is the Only Really Secure Computer a Dead

Computer? Retrieved from http://pawlan.com/monica/articles/sectools/

Taggart, T., & Brail, G. (2015, December 18). Can rowboat run the TypeScript Compiler? #1. Retrieved March 19, 2016, from https://github.com/apigee/rowboat/issues/1

Threat Analysis Group. (n.d.). Threat, vulnerability, risk – commonly mixed up terms. Retrieved from http://www.threatanalysis.com/2010/05/03/threat-vulnerability-risk-commonly-mixed-up-terms/

Torreborre, E. (n.d.). specs2 User Guid. Retrieved March 17, 2016, from https://etorreborre.github.io/specs2/guide/SPECS2-3.7.2/org.specs2.guide.UserGuide.html

Yegulalp, S. (2014, March 3). Node.js arrives for the JVM. Retrieved from http://www.infoworld.com/article/2610123/javascript/node-js-arrives-for-the-jvm.html

Appendices

Code

src/build.sbt

```
name := "Research paper examples"

version := "1.0"

scalaVersion := "2.11.7"

libraryDependencies ++= Seq(
  "io.apigee.trireme" % "trireme-core" % "0.8.9",
  "io.apigee.trireme" % "trireme-net" % "0.8.9",
  "io.apigee.trireme" % "trireme-node10src" % "0.8.9",
  "org.specs2" %% "specs2-core" % "3.7.2" % "test"
)

scalacOptions in Test ++= Seq("-Yrangepos")
```

src/main/scala/nashorn/MyClassFilter.scala

```
package nashorn

import jdk.nashorn.api.scripting.ClassFilter

class MyClassFilter extends ClassFilter {
  def exposeToScripts(className: String): Boolean = className.startsWith("java.lang.")
}
```

src/main/scala/rhino/MyClassShutter.scala

```
package rhino

import org.mozilla.javascript.ClassShutter

class MyClassShutter extends ClassShutter {
  override def visibleToScripts(fullClassName: String): Boolean = false
}
```

src/main/scala/nashorn/MyClassFilter.scala

```
package nashorn

import jdk.nashorn.api.scripting.ClassFilter

class MyClassFilter extends ClassFilter {
  def exposeToScripts(className: String): Boolean = className.startsWith("java.lang.")
}
```

src/test/scala/test/SecurityManagerSpec.scala

```
package test

import org.specs2.mutable._
```

```
/* This causes issues with sbt, therefore skipped */
class SecurityManagerSpec extends Specification {
  "The SecurityManager" should {
    "prevent the JVM from exiting" in pending {
      class MySecurityManager extends SecurityManager {
        override def checkExit(status: Int): Unit = throw new SecurityException()
      }

      System.setSecurityManager(new MySecurityManager)

      System.exit(0) should throwA(new SecurityException())
    }
  }
}
```

src/test/scala/test/nashorn/ClassFilterSpec.scala

```
package test.nashorn

import org.specs2.mutable._

import nashorn.MyClassFilter

import javax.script.ScriptEngine
import jdk.nashorn.api.scripting.NashornScriptEngineFactory

class ClassFilterSpec extends Specification {
  "The custom class filter" should {
    "Allow access to java.lang.String" in {
      val factory: NashornScriptEngineFactory = new NashornScriptEngineFactory()
      val engine: ScriptEngine = factory.getScriptEngine(new MyClassFilter())

      val result = engine.eval(
        """
          |var res = new java.lang.String("My new test string")
          |res
        """.stripMargin).asInstanceOf[String]

      result must be equalTo "My new test string"
    }

    "Disallow access to java.util.ArrayList" in {
      val factory: NashornScriptEngineFactory = new NashornScriptEngineFactory()
      val engine: ScriptEngine = factory.getScriptEngine(new MyClassFilter())

      engine.eval(
        """
          |var res = new java.util.ArrayList()
          |res.add("test")
          |res
        """.stripMargin) must throwA(new RuntimeException(new ClassNotFoundException("java.util.ArrayList")))
    }
  }
}
```

src/test/scala/test/rhino/BasicUsageSpec.scala

```
package test.rhino

import org.mozilla.javascript._

import org.specs2.mutable._

class BasicUsageSpec extends Specification {
  "Rhino" should {
    "run basic JavaScript code" in {
      // Source: https://github.com/mozilla/rhino/blob/master/examples/Control.java

      val cx: Context = Context.enter()

      // Set version to JavaScript1.2 so that we get object-literal style
      // printing instead of "[object Object]"
      cx.setLanguageVersion(Context.VERSION_1_2)

      // Initialize the standard objects (Object, Function, etc.)
      // This must be done before scripts can be executed.
      val scope: Scriptable = cx.initStandardObjects()

      // Now we can evaluate a script. Let's create a new object
      // using the object literal notation.
      val result: Any = cx.evaluateString(scope, "obj = {a:1, b:['x','y']}", "MySource", 1, null)

      val obj: Scriptable = scope.get("obj", scope).asInstanceOf[Scriptable]

      val b: Scriptable = obj.get("b", obj).asInstanceOf[Scriptable]

      // Should print {a:1, b:["x", "y"]}
      val fn: Function = ScriptableObject.getProperty(obj, "toString").asInstanceOf[Function]
      fn.call(cx, scope, obj, Array(new Object))  mustEqual """{a:1, b:["x", "y"]}"""

      Context.exit()

      // Since the result of an assignment expression is the value that was assigned, these must be equal
      obj mustEqual result

      obj.get("a", obj) mustEqual 1

      b.get(0, b) mustEqual "x"
      b.get(1, b) mustEqual "y"


    }
  }
}
```

src/test/scala/test/rhino/ClassAccessSpec.scala

```
package test.rhino

import org.mozilla.javascript._

import org.specs2.mutable._
import rhino.MyClassShutter

class ClassAccessSpec extends Specification {
  "Rhino class access" should {
    "be able to access classes without a ClassShutter" in {
      val cx: Context = Context.enter()
      val scope: Scriptable = cx.initStandardObjects()

      val result: Any = cx.evaluateString(scope,
        """
          |a = new java.lang.String("aoeu");
        """.stripMargin, "MySource", 1, null)

      Context.exit()
      1 must be equalTo 1 // this block may not throw
    }

    "not be able to access classes forbidden by a ClassShutter" in {

      val cx: Context = Context.enter()
      cx.setClassShutter(new MyClassShutter)
      val scope: Scriptable = cx.initStandardObjects()

      val result: Any = cx.evaluateString(scope,
        """
          |a = new java.lang.String("aoeu");
        """.stripMargin, "MySource", 1, null) should throwA[EcmaError]

      Context.exit()

      1 must be equalTo 1 // Context.exit() can't be last line, last line should be Specs2 expression
    }
  }
}
```

src/test/scala/test/rhino/InstructionLimitSpec.scala

```
package test.rhino

import org.mozilla.javascript._

import org.specs2.mutable._

class InstructionLimitSpec extends Specification {
  "Rhino instruction limit" should {
    "run basic JavaScript code" in {
      val cx: Context = Context.enter()
      cx.setInstructionObserverThreshold(5000)
      val scope: Scriptable = cx.initStandardObjects()

      val result: Any = cx.evaluateString(scope, """""", "MySource", 1, null)

      val obj: Scriptable = scope.get("obj", scope).asInstanceOf[Scriptable]

      val b: Scriptable = obj.get("b", obj).asInstanceOf[Scriptable]

      // Should print {a:1, b:["x", "y"]}
      val fn: Function = ScriptableObject.getProperty(obj, "toString").asInstanceOf[Function]
      fn.call(cx, scope, obj, Array(new Object))  mustEqual """{a:1, b:["x", "y"]}"""

      Context.exit()

      // Since the result of an assignment expression is the value that was assigned, these must be equal
      obj mustEqual result

      obj.get("a", obj) mustEqual 1

      b.get(0, b) mustEqual "x"
      b.get(1, b) mustEqual "y"


    }
  }
}
```

src/test/scala/test/rhino/OutOfMemorySpec.scala

```
package test.rhino

import org.mozilla.javascript._

import org.specs2.mutable._

class OutOfMemorySpec extends Specification {
  "Rhino" should {
    "run basic JavaScript code" in {
      val cx: Context = Context.enter()
      val scope: Scriptable = cx.initStandardObjects()

      cx.evaluateString(scope,
        """
          |var a = [];
          |while(true) { a.push("aoeu") }
        """.stripMargin, "MySource", 1, null) must throwA[OutOfMemoryError]

      1 must be equalTo 1
    }
  }
}
```

src/test/scala/test/trireme/BasicUsageSpec.scala

```
package test.trireme

import java.util.concurrent.ExecutionException

import org.specs2.mutable._

import io.apigee.trireme.core.{ScriptStatus, NodeScript, NodeEnvironment}

class BasicUsageSpec extends Specification{
  "Trireme" should {
    "run basic JavaScript code" in {
      val env: NodeEnvironment = new NodeEnvironment
      val script: NodeScript = env.createScript(
        "my-test-script.js",
        """
          |var test = 1 + 2;
          |var math = Math.sqrt(test);
        """.stripMargin,
        null
      )

      val status: ScriptStatus = script.execute().get()

      status.isOk should be equalTo true
    }
  }
}
```

src/test/scala/test/trireme/FileSystemAccessSpec.scala

```
package test.trireme

import java.util.concurrent.{TimeUnit, ExecutionException}

import org.mozilla.javascript.{JavaScriptException, EcmaError, ClassShutter}
import org.specs2.mutable._

import io.apigee.trireme.core.{Sandbox, ScriptStatus, NodeScript, NodeEnvironment}

class FileSystemAccessSpec extends Specification {
  "Trireme file access" should {
    "not allow scripts to access to the parent directory in a chroot-like jail" in {
      val env: NodeEnvironment = new NodeEnvironment
      val sandbox = (new Sandbox).setFilesystemRoot("./target")
      env.setSandbox(sandbox)

      val script: NodeScript = env.createScript(
        "my-test-script.js",
        """var fs = require('fs');
          |fs.readdirSync('..');
        """.stripMargin,
        null
      )

      script.execute().get() should throwA(new ExecutionException(new JavaScriptException("Error: ENOENT")))
    }

    "allow scripts to access the current directory in a chroot-like jail" in {
      val env: NodeEnvironment = new NodeEnvironment
      val sandbox = new Sandbox
      sandbox.setFilesystemRoot("./target")
      env.setSandbox(sandbox)

      val script: NodeScript = env.createScript(
        "my-test-script.js",
        """var fs = require('fs');
          |fs.readdirSync('.');
        """.stripMargin,
      null
    )

    val status = script.execute().get()

    status.isOk must be equalTo true
  }
 }
}
```

src/test/scala/test/trireme/JavaClassAccessSpec.scala

```
package test.trireme

import java.util.concurrent.ExecutionException

import org.mozilla.javascript.{EcmaError, ClassShutter}
import org.specs2.mutable._

import io.apigee.trireme.core.{Sandbox, ScriptStatus, NodeScript, NodeEnvironment}

class JavaClassAccessSpec extends Specification{
 "Trireme" should {
  "allow scripts access to Java classes specified in an extra class shutter" in {
   val env: NodeEnvironment = new NodeEnvironment
   env.setSandbox((new Sandbox).setExtraClassShutter(new ClassShutter {
    override def visibleToScripts(fullClassName: String): Boolean = fullClassName.startsWith("java.io.")
   }))

   val script: NodeScript = env.createScript(
    "my-test-script.js",
     """
      |var test = java.lang.String("aoeu");
      |var f = new java.io.File("test.txt");
      |f
     """.stripMargin,
    null
   )

   val status: ScriptStatus = script.execute().get()

   status.isOk must be equalTo true
  }

  "disallow scripts access to Java classes" in {
   val env: NodeEnvironment = new NodeEnvironment
   env.setSandbox((new Sandbox).setExtraClassShutter(new ClassShutter {
    override def visibleToScripts(fullClassName: String): Boolean = false
   }))

   val script: NodeScript = env.createScript(
    "my-test-script.js",
     """
      |var test = java.lang.String("aoeu");
      |var f = new java.io.File("test.txt");
      |f
     """.stripMargin,
      null
    )

    script.execute().get() should throwA[ExecutionException]
  }
 }
}
```

src/test/scala/test/trireme/ResourceStarvationSpec.scala

```
package test.trireme

import java.util.concurrent.{TimeUnit, ExecutionException}

import org.mozilla.javascript.{JavaScriptException, EcmaError, ClassShutter}
import org.specs2.mutable._

import io.apigee.trireme.core.{Sandbox, ScriptStatus, NodeScript, NodeEnvironment}

class ResourceStarvationSpec extends Specification{
 "Trireme script time limits" should {
  "not allow scripts to abuse the CPU" in {
    val env: NodeEnvironment = new NodeEnvironment
    env.setScriptTimeLimit(1, TimeUnit.SECONDS)

    val script: NodeScript = env.createScript(
      "my-test-script.js",
       """
         |while(1) {}
       """.stripMargin,
      null
    )

    script.execute().get() must throwA(new ExecutionException(new JavaScriptException("Script timed out")))

  }
 }
}
```