Nashorn script security permissions

Nashorn script security permissions

When you call "eval" method on ScriptEngine passing a String or a Reader, the script is treated as untrusted and so it gets only permissions given to "sandbox" code. This is true for eval ECMAScript builtin function as well. The evaluated nashorn script does **not** inherit permissions of the calling Java code! This is because nashorn script engine receives script whose origin URL is unknown to the engine!

So, how can we grant security permissions to specific scripts? We may have trusted local scripts - for which we may want to grant more permissions compared to sandbox scripts.

URLReader

Instead of passing a String or any other Reader to "eval" method, you can pass an instance of jdk.nashorn.api.scripting.URLReader. URLReader constructor accepts a URL. You can then grant permissions to a specific script by using URL of the script in your security policy file. The following sample code demonstrates the use of URLReader. The following files Main.java, test.js and test.policy are assumed to stored under "D:\test" directory on a Windows machine. You may want to adjust security policy file – if you store these under a different directory (or in a different OS).

Main.java

```
import java.io.*;
import java.nio.file.*;
import javax.script.*;
import jdk.nashorn.api.scripting.*;

public class Main {
  public static void main(String[] args) throws Exception {
    ScriptEngineManager m = new ScriptEngineManager();
    ScriptEngine e = m.getEngineByName("nashorn");
    if (args.length == 0) {
      System.err.println("Usage: java Main <script_file>");
      return;
    }

    // args[0] is script file to which permissions are granted
    // in security policy
    File file = new File(args[0]);

    // read the file content and pass a String to 'eval'
    // The script is untrusted as nashorn does not know the origin!
    try {
        e.eval(new String(Files.readAllBytes(file.toPath())));
    } catch (SecurityException se) {
        System.out.println(se);
    }

    // create a Reader over the file and pass to 'eval'
    // The script is untrusted as nashorn does not know the origin!
    try {
        e.eval(new FileReader(file));
    } catch (SecurityException se) {
        System.out.println(se);
    }

    // pass a URLReader on file - script will get permissions
    // configured in security policy!
    e.eval(new URLReader(file.toURL()));
  }
}
```

test.js

```
var File = java.io.File;
// list contents of the current directory!
for each (var f in new File(".").list())
   print(f)
```

test.policy

```
// give AllPermission for Main class (or any class in that directory!)
grant codeBase "file:///d:/test" {
    permission java.security.AllPermission;
};

// give AllPermission to test.js script
grant codeBase "file:///d:/test/test.js" {
    permission java.security.AllPermission;
};
```

$ javac Main.java

$ java -Djava.security.manager -Djava.security.policy=./test.policy Main test.js

java.security.AccessControlException: access denied ("java.io.FilePermission" "." "read")
java.security.AccessControlException: access denied ("java.io.FilePermission" "." "read")
Main.class
Main.java
test.js
test.policy

As you can see from the above example, SecurityException is thrown when "eval" was called with a String or a FileReader. But, if you pass a URLReader, nashorn will associate that URL with the script and therefore security permissions are granted as per your security policy. This allows trusted scripts be granted with more permissions.

load and loadWithNewGlobal builtin functions

Nashorn supports 'load' builtin extension function. This can be called from a script to load another script from a URL or a File. When script is loaded with "load" call, Nashorn associates URL/File origin to the script and therefore permissions are granted as per the current security policy. This is another way to grant security permissions to specific scripts. loadWithNewGlobal is another nashorn builtin extension function. This can also be used to load script from a URL or a File. Nashorn will associate script URL/File origin to the script and so permissions are granted as per the current security policy.

Summary of various ways of loading/evaluating scripts and security implications:

- **javax.script** APIs **engine.eval(String)** and **engine.eval(Reader)**: These scripts are treated as sandbox code - except for **jdk.nashorn.api.scripting.URLReader**. If you pass **URLReader**, script origin based on that **URL** associated is used. So, security permissions are based on the script origin URL.
- ECMAScript **eval** builtin function: Script is treated as "sandbox" and hence gets only sandbox permissions.
- **load** function with a file File/URL: This method and command line method both associate a URL/File origin for the script and hence script URL /File based fine-grained permission can be used. When you run with security manager on, you can specify permissions for specific script URLs or file: URLs
- **load** from a script object like **load({ name: "foo", script: str}):** This is equivalent to "eval" - but it associates a name with script and so stack traces will have nice readable name instead of <eval>. "str" may be computed or a literal. It does not matter. But, script is treated as 'sandbox'.
- **loadWithNewGlobal** function: This is similar to load [all options are load available]. The difference is that it creates a new EMCAScript global scope and loads your code into that global. This avoids global namespace pollution. Note that security access permission is based on script origin URL or File if you pass a URL or a File.

- **loadWithNewGlobal** from a script object like **loadWithNewGlobal({ name: "foo", script: str})**: The script is treated as a sandbox.
- **javax.script** APIs **engine.eval(Reader, Bindings)** and **engine.eval(String, Bindings)**: This is similar to the other **engine.eval** methods in that these are sandbox script evaluations unless Reader is a **URLReader**. But, these methods create/associate a fresh ECMAScript global and load code there [similar to **loadWithNewGlobal** in that sense]