Evaluating the Flexibility of the Java Sandbox

By: Zack Coker, Michael Maass, Tianyuan Ding, Claire Le Goues, and Joshua Sunshine
Motivation

Detected Exploits

- Oracle Java
- Browsers
- Adobe Reader


Exploitation of application vulnerabilities
from survey of 1 million Trusteer customers, December 2013

- Oracle Java: 15%
- Adobe Reader: 13%
- Browsers: 22%
- Others: 50%

Figure 4. Exploitation of application vulnerabilities

Source: IBM X-Force® Research and Development
Java Sandbox

Untrusted Java Applications
How is the Java sandbox actually used?

Li Gong, the primary designer of the Java security architecture stated he didn’t know how extensively the “fined grained access control mechanism” (i.e. the Java sandbox) is used.

[ACSAC 2009 Keynote]
Importance of investigating how security tools are used in practice

• Design more intuitive security mechanisms
• Possible language features
• Differentiate between malicious and benign code
Sandbox Example

1. `Sandboxed Application` connects to `Socket` with `connect("internal.auth:80")`.
2. `Socket` checks connection with `checkConnect("internal.auth", 80)`.
3. `SecurityManager` checks permissions with `checkPermission(...)`.
4. `AccessController` evaluates permissions with `implies(...)`.
5. The result determines if the connection is allowed, resulting in either `SecurityException` or `Do Nothing`.
6. The overall policy is determined by the chain of these operations.
Turn off Sandbox Example

- Sandbox Application
  - System.setSecurityManager(null)
  - allowed?
    - SecurityException
    - Sandbox Removed
Study

In this work, we investigated how benign open source Java programs interact with the Java sandbox to:

• Characterize sandbox interactions
• Detect malicious applications
• Understand developer difficulties with the sandbox
Research Question

How do benign open source Java applications interact with the security manager?
Possible Answers

Four non-disjoint possibilities for all benign applications:

1. Benign applications never disable the security manager.
2. Benign applications never weaken a set security manager.
3. Benign applications do not change a set security manager.
4. Benign applications never modify the security manager if a self protecting security manager is set.
Not Self-protecting Permissions

<table>
<thead>
<tr>
<th>Risky Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>RuntimePermission(&quot;setSecurityManager&quot;)</td>
</tr>
<tr>
<td>ReflectPermission(&quot;suppressAccessChecks&quot;)</td>
</tr>
<tr>
<td>RuntimePermission(&quot;createClassLoader&quot;)</td>
</tr>
<tr>
<td>SecurityPermission(&quot;setPolicy&quot;)</td>
</tr>
<tr>
<td>FilePermission(&quot;&lt;&lt;ALL FILES&gt;&gt;&quot;, &quot;write, execute&quot;)</td>
</tr>
<tr>
<td>RuntimePermission(&quot;accessClassInPackage.sun&quot;)</td>
</tr>
<tr>
<td>SecurityPermission(&quot;setProperty.package.access&quot;)</td>
</tr>
</tbody>
</table>
Methodology

1. Collect large corpus of benign Java applications that use or interact with the security manager.
2. Static analysis to trace where the Security Manager was set and initialized
4. Dynamic Analysis to verify sandbox interactions

Goal: completely understand how benign applications interact with the security manager.
Qualitas Corpus

GitHub
## Security Manager Interaction Classification

<table>
<thead>
<tr>
<th>Type of Benign Interaction</th>
<th>Qualitas Corpus</th>
<th>GitHub</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sets a self-protecting manager</td>
<td>6</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>Changes set manager</td>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Supports being sandboxed</td>
<td>10</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>Interacts only in unit tests</td>
<td>3</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

28/36 (78%) never change the sandbox settings once the settings are initially set.
Possible Answers

Four Possibilities:

1. Benign applications never disable the security manager.
2. Benign applications never weaken a set security manager.
3. Benign applications do not change a set security manager.
4. Benign applications never modify the security manager if a self protecting security manager is set.
Research Question Evaluation

1. Benign applications do disable the security manager.
2. Benign applications do weaken a set security manager.
3. Benign applications do change a set security manager.
4. Benign applications never modify the sandbox if a self protecting security manager is set.
Examples of Disabling the Security Manager

```java
package jtimelag;

public class JTimelag {
    public static void main(String[] args) {
        System.setSecurityManager(null);
        Fenetre fen = new Fenetre();
        fen.setVisible(true);
    }
}
```
Not Sandboxed Requirement

• "//TODO Hack so that when the classloader loading the fwk is created we don't have funny permissions. This should be revisited."
Eclipse

JVM
Enforcing Architectural Constraints
Weakening the Security Manager?

Permission List  vs.  Forbidden List
<permissions>

<grant class="java.security.AllPermission"/>

<revoke class="java.util.PropertyPermission"/>

<permissions>

[https://ant.apache.org/manual/Types/permissions.html]
Takeaways from empirical study

• Applications interact with the sandbox for non security related reasons
  – Harder to design exploit mitigations

• Sandbox complexity and flexibility leads to insecurity

• A majority of applications in the dataset (78%) never change the security manager if a self-protecting security manager is set
Can we differentiate benign and malicious applications?

EMPIRICAL VALIDATION
Detection Rules

Privilege Escalation Rule: If a self-protecting security manager is set, a class cannot load a more privileged class.

Security Manager Rule: Once a self-protecting security manager is set, the manager cannot change
## Validation Results

<table>
<thead>
<tr>
<th>CVE-ID</th>
<th>Privilege Escalation</th>
<th>Security Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011-3544</td>
<td>Attack Succeeded</td>
<td>Attack Blocked</td>
</tr>
<tr>
<td>2012-0507</td>
<td>Attack Blocked</td>
<td>Attack Blocked</td>
</tr>
<tr>
<td>2012-4681</td>
<td>Attack Succeeded</td>
<td>Attack Blocked</td>
</tr>
<tr>
<td>2012-5076</td>
<td>Attack Succeeded</td>
<td>Attack Blocked</td>
</tr>
<tr>
<td>2013-0422</td>
<td>Attack Blocked</td>
<td>Attack Blocked</td>
</tr>
<tr>
<td>2013-0431</td>
<td>Attack Blocked</td>
<td>Attack Blocked</td>
</tr>
<tr>
<td>2013-1488</td>
<td>Attack Succeeded</td>
<td>Attack Blocked</td>
</tr>
<tr>
<td>2013-2423</td>
<td>Attack Succeeded</td>
<td>Attack Blocked</td>
</tr>
<tr>
<td>2013-2460</td>
<td>Attack Blocked</td>
<td>Attack Blocked</td>
</tr>
<tr>
<td>2013-2465</td>
<td>Attack Succeeded</td>
<td>Attack Blocked</td>
</tr>
</tbody>
</table>
Conclusion – Importance of Investigating Security Tools

• Improve security mechanisms: Java’s flexibility and complexity leads to misuse.

• Possible language features: People use the security manager for non-security reasons, such as preventing sub-applications from exiting; Java would benefit from supporting these features separately.

• Differentiate malicious and benign code: a language-level tool that notices when privilege escalation/weakening of security manager occurs could catch future exploits.