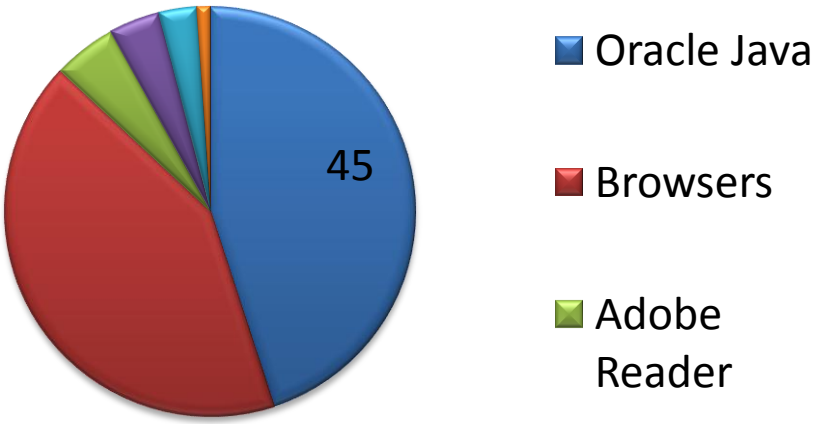
Evaluating the Flexibility of the Java Sandbox

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

Motivation

Detected Exploits



Kaspersky Security Bulletin 2014

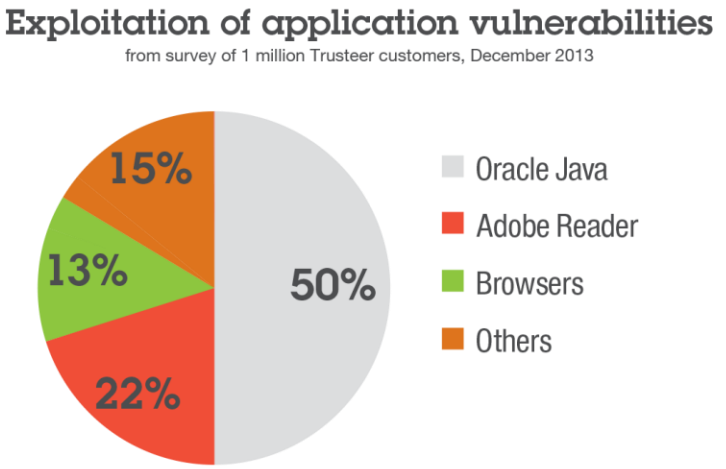
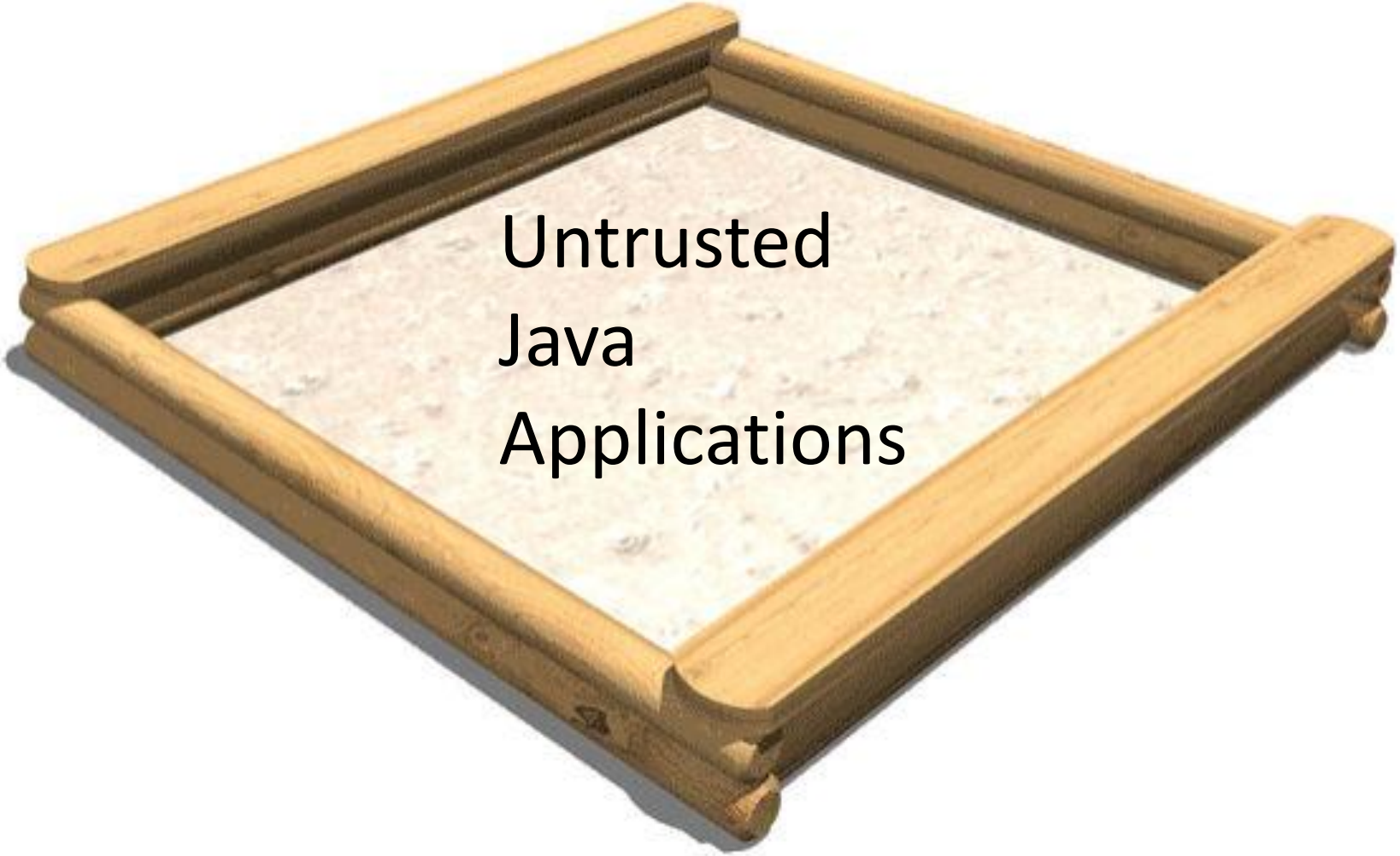


Figure 4. Exploitation of application vulnerabilities

Source: IBM X-Force® Research and Development

Java Sandbox



How is the Java sandbox actually used?

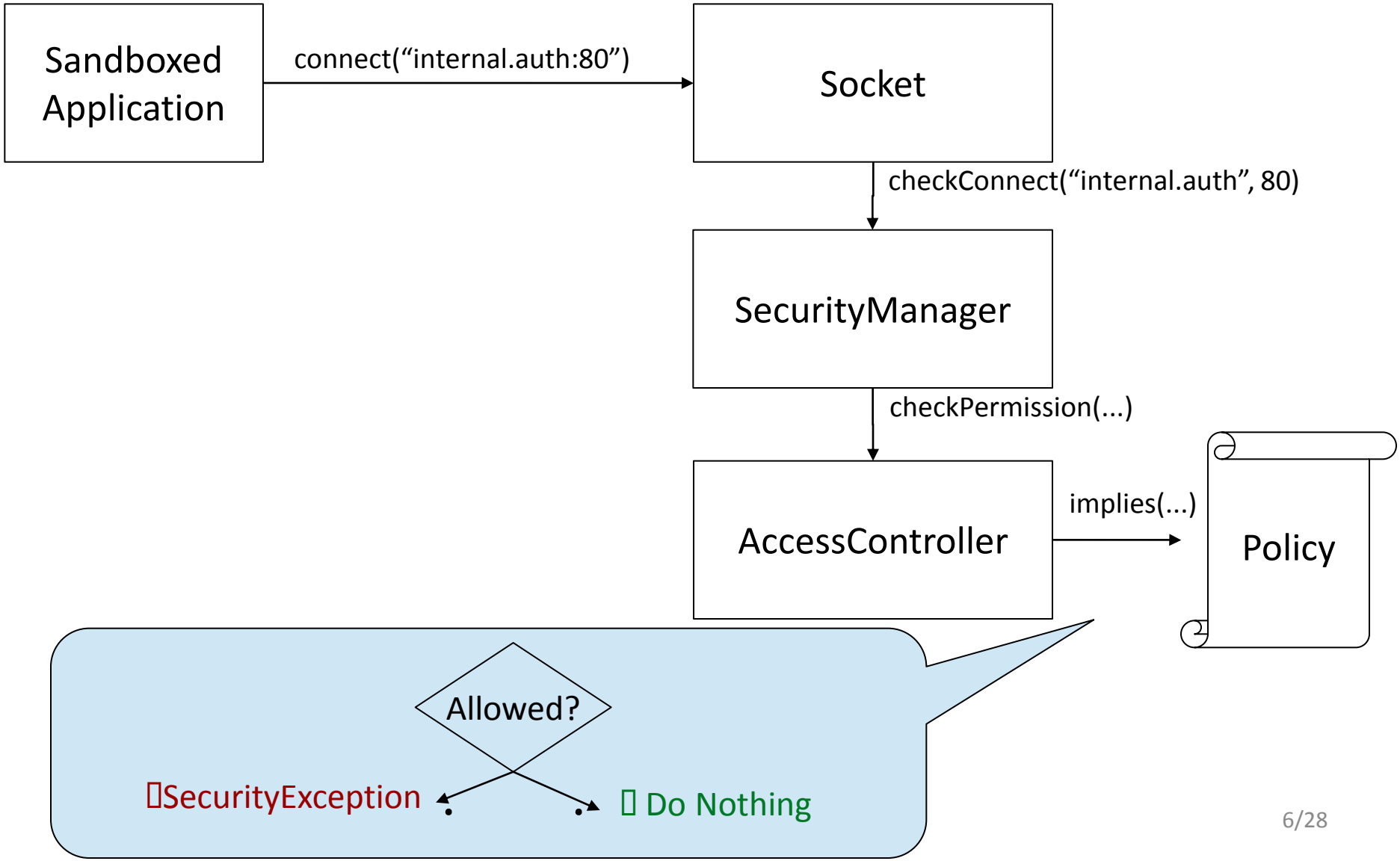
Li Gong, the primary designer of the Java security architecture stated he didn't know how extensively the "fined grained access control mechanism" (i.e. the Java sandbox) is used.

[ACSAC 2009 Keynote]

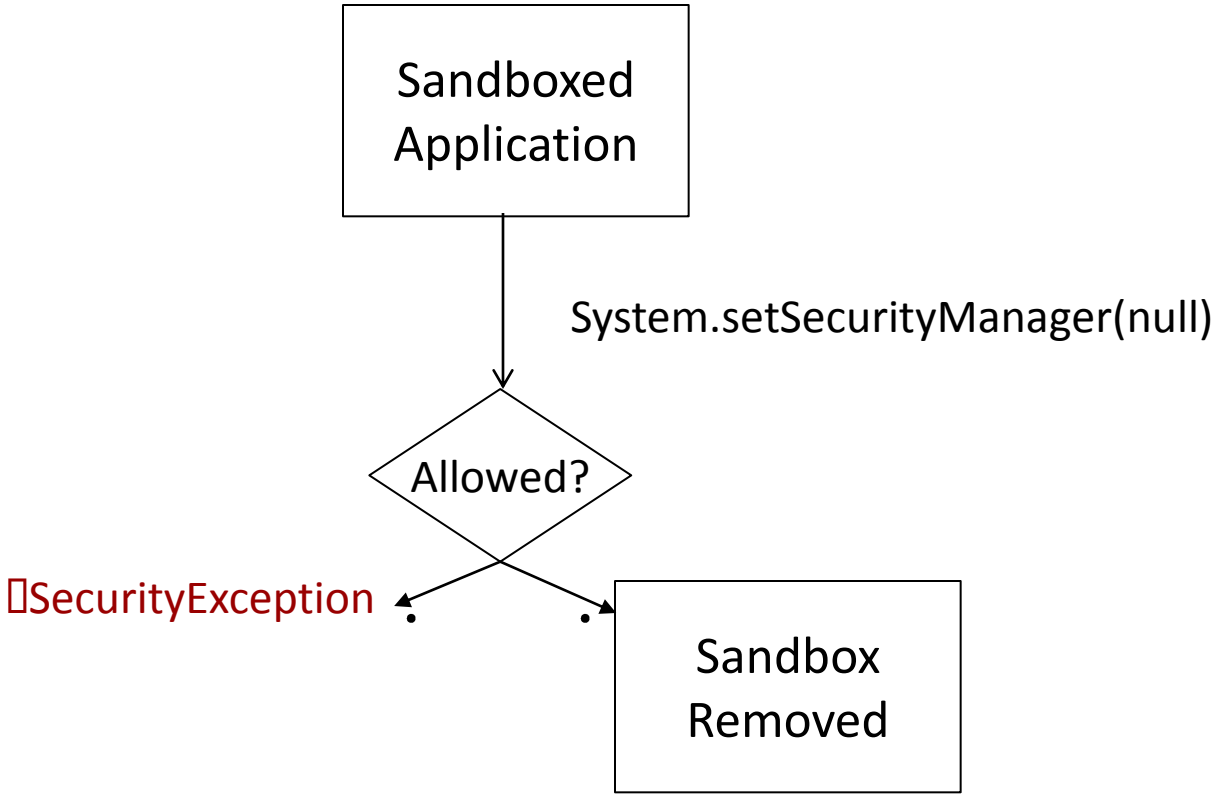
Importance of investigating how security tools are used in practice

- Design more intuitive security mechanisms
- Possible language features
- Differentiate between malicious and benign code

Sandbox Example



Turn off Sandbox Example



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 manger
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

Risky Permissions
RuntimePermission(“setSecurityManager")
ReflectPermission(“suppressAccessChecks")
RuntimePermission(“createClassLoader")
SecurityPermission(“setPolicy")
FilePermission(“<<ALL FILES>>", “write, execute")
RuntimePermission(“accessClassInPackage.sun")
SecurityPermission(“setProperty.package.access")

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
3. Manual code inspection (two annotators, check for agreement).
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

Type of Benign Interaction	Qualitas Corpus	GitHub	Total
Sets a self-protecting manager	6	1	7
Changes set manager	5	3	8
Supports being sandboxed	10	3	13
Interacts only in unit tests	3	5	8

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 manger
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 manger
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

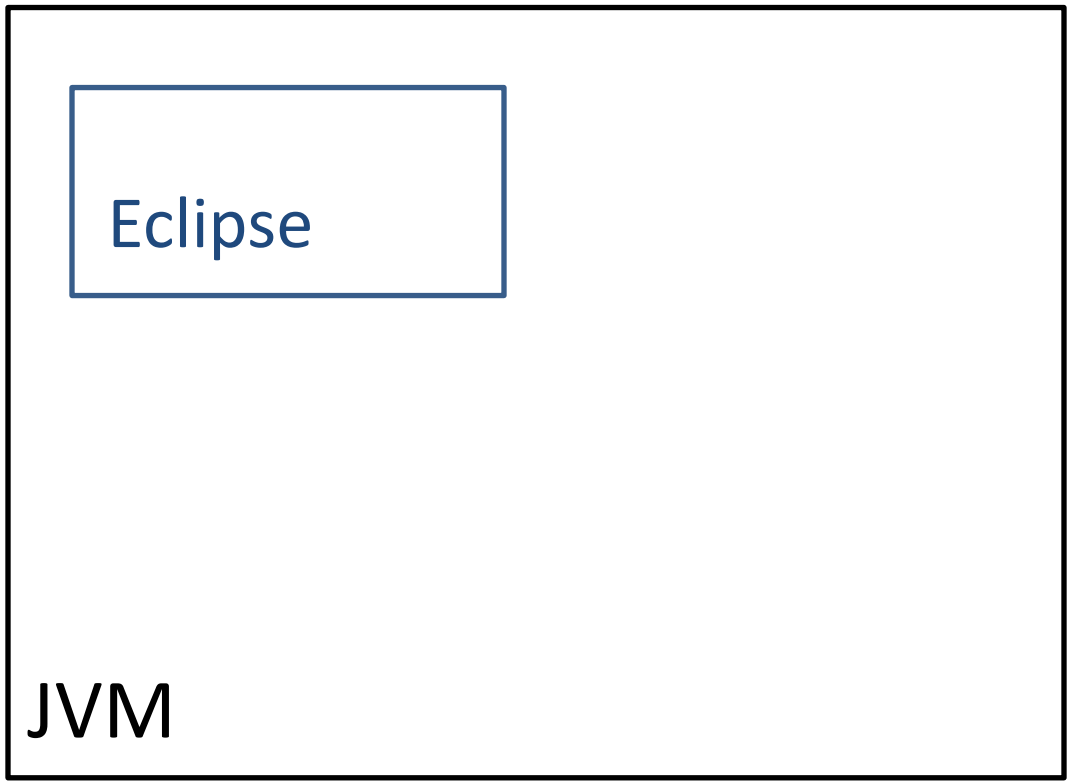
```
package jtimelag;

public class JTimelag {

    public static void main(String[] args) {

        System.setSecurityManager(null);

        Fenetre fen = new Fenetre();
        fen.setVisible(true);
    }

}
```

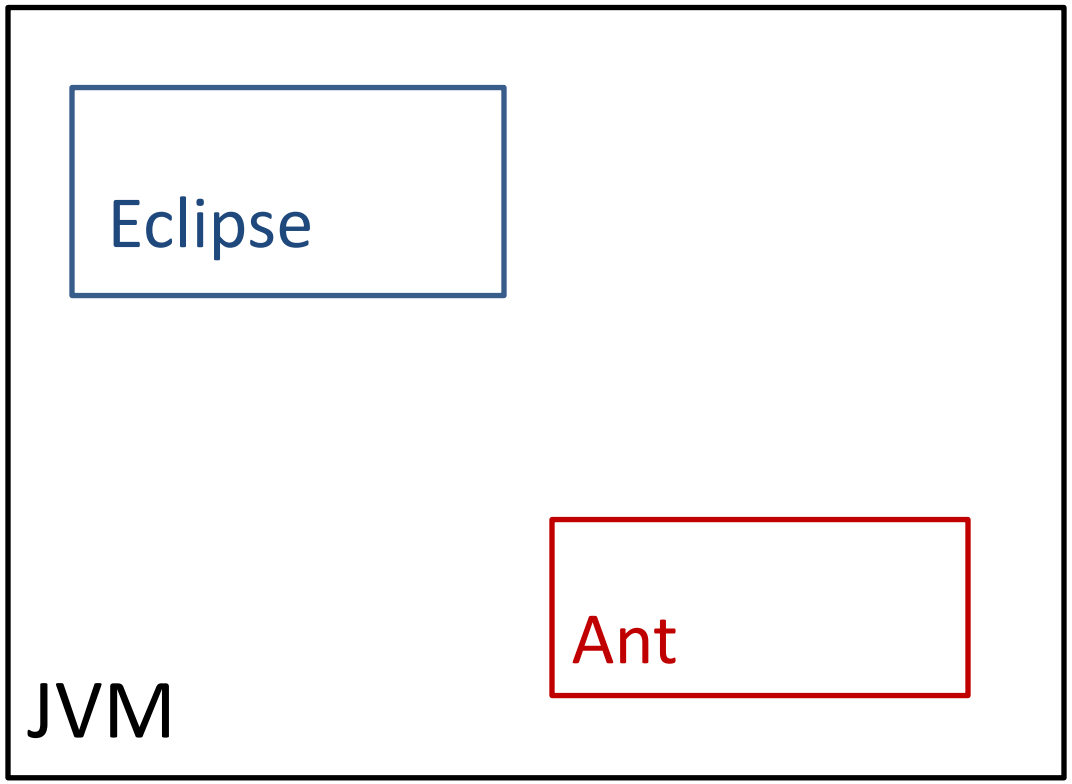
?

Not Sandboxed Requirement

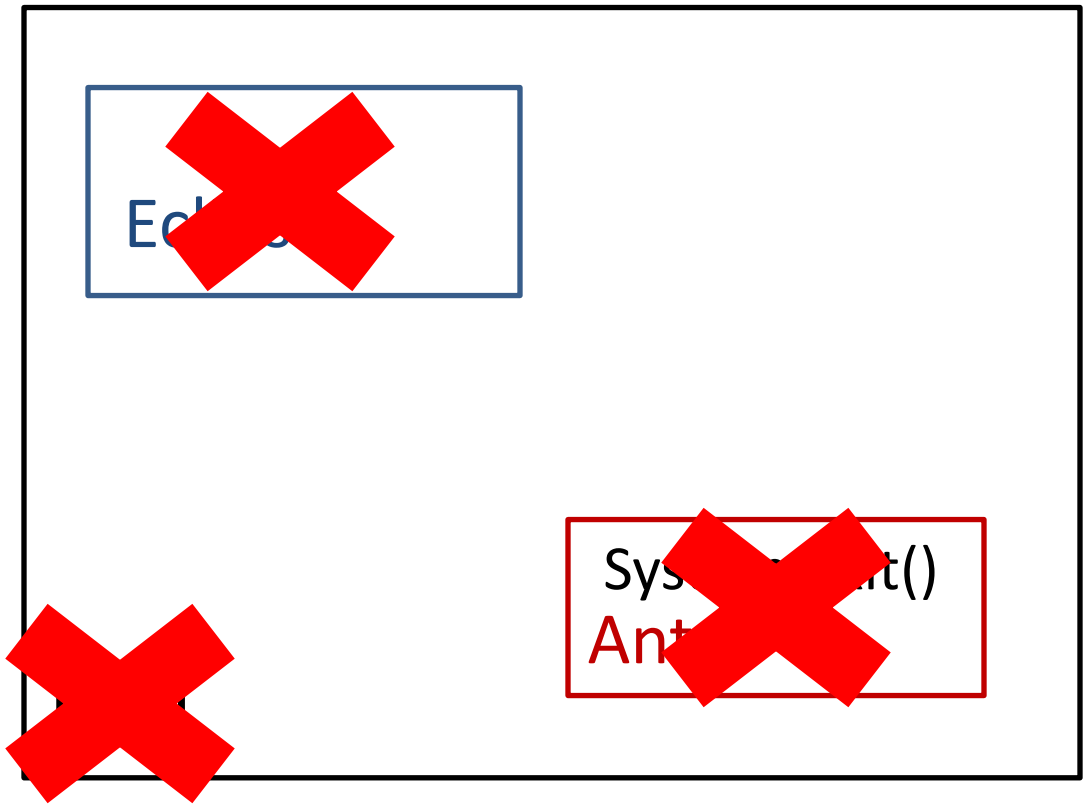
- "//TODO Hack so that when the classloader loading the fwk is created we don't have funny permissions. This should be revisited."

[eclipse]


Eclipse
JVM


Eclipse
Ant
JVM

Enforcing Architectural Constraints



Weakening the Security Manager?

vs.

Risky Permissions
RuntimePermission(“setSecurityManager")
ReflectPermission(“suppressAccessChecks")
RuntimePermission(“createClassLoader")
SecurityPermission(“setPolicy")
FilePermission(“<<ALL FILES>>", “write, execute")
RuntimePermission(“accessClassInPackage.sun")
SecurityPermission(“setProperty.package.access")

17/21

```
<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

CVE-ID	Privilege Escalation	Security Manager
2011-3544	Attack Succeeded	Attack Blocked
2012-0507	Attack Blocked	Attack Blocked
2012-4681	Attack Succeeded	Attack Blocked
2012-5076	Attack Succeeded	Attack Blocked
2013-0422	Attack Blocked	Attack Blocked
2013-0431	Attack Blocked	Attack Blocked
2013-1488	Attack Succeeded	Attack Blocked
2013-2423	Attack Succeeded	Attack Blocked
2013-2460	Attack Blocked	Attack Blocked
2013-2465	Attack Succeeded	Attack Blocked

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.