Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 253 (2009) 145–151

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Jalapa: Securing Java with Local Policies

Tool Demonstration

Massimo Bartoletti

Dipartimento di Matematica e Informatica, Università degli Studi di Cagliari, Italy

Gabriele Costa

Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, Italy

Roberto Zunino

Dipartimento di Ingegneria e Scienza dell'Informazione, Università di Trento, Italy

Abstract

We present Jalapa, a tool for securing Java bytecode programs with history-based usage policies. Policies are defined by *usage automata*, that recognize the forbidden execution histories. Usage automata are expressive enough to allow programmers specify of many real-world usage policies; yet, they are simple enough to permit formal reasoning. Programmers can sandbox untrusted pieces of code with usage policies. The Jalapa tool rewrites the Java bytecode by adding the hooks for the mechanism that enforces the given policies at run-time.

Keywords: Usage control, history-based security, bytecode rewriting

1 Introduction

Security has been a major concern in the design and implementation of Java, starting from its early incarnations. Building upon the "safety pillars" of bytecode verification and secure class loading, new defence mechanisms have been developed over the years.

With the release of the JDK 1.0, a mechanism was provided to run untrusted mobile code into a *sandbox* with limited computational functionalities. The default sandbox prevented untrusted code from, e.g. accessing the local file sytem, from redefining the security manager (otherwise one could circumvent the sandbox), from connecting to (or accepting a connection from) any URL other than the one the code was downloaded, *etc.* Although these functionalities were completely customizable,

doi:10.1016/j.entcs.2009.11.020

this required to subclass the security manager, making it difficult to separate the functional aspects of programming from the security aspects.

While retaining the basic sandbox model of the JDK 1.0, the JDK 1.1 featured a "black or white" security model, based on digital signatures. Java-enabled browsers could be configured to trust digitally-signed mobile code, provided that the signature was put by a trusted entity. Trusted code were granted full privileges, while untrusted code were run without any privilege.

Starting with the JDK 1.2, a more fine-grained mechanism was devised, based on *stack inspection* [9]. This provides for associating methods with "protection domains" that reflect their provenance, and for defining a global security policy that grants each protection domain a set of permissions. Code includes local checks that guard access to critical resources. At run-time, an access authorization is granted when *all* the methods on the call stack have the required permission (a special case is that of privileged calls, that trust the methods below them in the call stack). Being strongly biased towards implementation, this mechanism suffers from some major shortcomings. For instance, since a method removed from the call stack no longer affects security, stack inspection does not offer any protection when trusted code uses objects supplied by untrusted code [8].

Although many security policies are not enforceable by stack inspection, at present Java offers no other facilities to specify and enforce user-defined policies. Therefore, it is common practice to renounce to separating duties between functionality and security, and to implement the needed enforcement mechanism with local checks explicitly inserted into the code by programmers. Since forgetting even a single check might compromise the security of the whole application, programmers have to inspect their code very carefully. This may be cumbersome even for small programs, and it may also lead to unnecessary checking.

History-based security has been repeatedly proposed as a replacement for stack inspection [1,7,11]. Clearly, the ability of checking the whole execution history, instead of the call stack only, places history-based mechanisms a step forward stack inspection, from the expressivity viewpoint. However, since many possible history-based models can be devised, it is crucial to choose one which wisely conciliates the expressive power with the theoretical properties enjoyed. It is also important that the security mechanism can be implementated in a way that makes it transparent to programmers, and with a negligible run-time overhead.

Jalapa advocates *local usage policies* [3] as a history-based model for securing Java applications. Some remarkable features of Jalapa are that:

- local usage policies are expressive enough to model security requirements of real-world applications. For instance, we used them to specify the typical set of policies of a realistic bulletin board system [10].
- at the same time, usage policies are simple enough to be statically amenable, e.g. they can be model-checked against abstractions of program usages [4].
- local usage policies generalise global policies and local checks. The ability of sandboxing a piece of code by localizing the scope of a policy is particularly

relevant, as the current programming methodologies provide for reusing code, and for exploiting services and components, offered by untrusted third parties.

- apart from the localization of sandboxes, enforcing policies is completely transparent to programmers.
- since the enforcement mechanism is based on bytecode rewriting, it does not require a custom Java Virtual Machine.
- even when the program source code is unavailable, Jalapa allows for specifying and enforcing policies on its behavior, by directly modifying the bytecode.

This paper gives an overview of Jalapa. We start by presenting our methodology for securing Java applications through local usage policies, with the help of some examples. Then, we give some insights about the design and the implementation of our tool, and we summarise the artifacts supporting our tool. We conclude by highlighting some of the present and future challenges of Jalapa.

2 Securing Java with local usage policies

We illustrate our methodology for securing Java programs, as well as some key features of Jalapa, with the help of an example. Suppose you have a simple Web browser whose functionality can be extended with plugins, and with methods for handling connections and cookies. Since plugins can be downloaded from the network, possibly from untrusted sites, we want to control their behaviour, and block their execution at the moment they attempt some malicious action. In particular, we focus here on two confinement policies, that prevent plugins from transmitting data read from the local file system, either directly or by exploiting cookies to implement a covert communication channel (although stronger, these policies imply non-interference). Before formally specifying these policies, we consider a skeletal implementation of the classes `Browser` and `Plugin`.

```
public class Browser {
    private Map<URL,String> cookies;
    public Browser() { cookies = new HashMap<URL,String>(); }
    public void connect(URL url) throws Exception {
        URLConnection uc = url.openConnection();
        out = new BufferedWriter(new OutputStreamWriter(uc.getOutputStream()));...}
    public void writeCookie(URL u, String msg) { cookies.put(u,msg); }
    public String readCookie(URL u) { return cookies.get(u); }
}

public abstract class Plugin implements Runnable {
    Plugin(Browser browser, String name, URL codebase) { ... }
    public void doIt() { try {  // invokes this.run() within the sandbox
                                PolicyPool.sandbox("plugin-out", this);
                              } catch (Throwable e) { e.printStackTrace(); } }
}
```

We assume that browser plugins extend the `Plugin` abstract class, by implementing the method `run()`. The browser starts a plugin by invoking the method `doIt()`, which is quite peculiar. Actually, it defines a `sandbox`, which will enforce the policy `plugin-out` throughout the run of the plugin. This means that all the security-relevant methods called while executing the method `run()` will be monitored, and blocked if not conformant to the policy. This policy is specified by the usage automaton `plugin-out` below on the left, to be discussed in a while.

```
name: plugin-out
aliases:
read  := FileInputStream.<init>()
read  := Browser.readCookie(URL u)
write := Socket.getOutputStream()
states: q0 q1 fail
start: q0
final: fail
trans:
q0 -- read --> q1
q1 -- write --> fail
```

```
name: plugin-cookies
aliases:
cookie(u) := Browser.writeCookie(URL u,String m)
cookie(u) := Browser.readCookie(URL u)
init(p,u) := (p:Plugin).<init>(URL u)
start(p)  := (p:Plugin).doIt()
states: q0 q1 q2 fail
start: q0
final: fail
trans:
q0 -- init(p,u) --> q1
q1 -- start(p) --> q2
q2 -- cookie(u') --> fail when u'!=u
q2 -- start(p') --> q1 when p'!=p
```

A usage automaton closely resembles a finite state automaton. The field tagged `name` just defines the name of the policy. The tag `aliases` defines a mapping from the signatures of security-relevant methods to *events* that trigger the transitions of the usage automaton. E.g. in the usage automaton `plugin-out` above, the event `read` is fired whenever a new object of the class `FileInputStream` is created, or a cookie is accessed through the method `readCookie`. Similarly, the event `write` is fired when the method `getOutputStream` is invoked on a `Socket`. The remaining fields describe the logics of the automaton. The tag `states` is for the set of states, `start` is for the initial state, and `final` is for the final state, denoting a policy violation. The tag `trans` preludes to the transition relation of the automaton. In our example, a transition from `q0` to `q1` occurs upon reading any file or cookie. A transition from `q1` to `fail` occurs upon opening an output stream on a socket. Since `fail` is offending, this indeed implements the first confinement policy.

The second policy is specified by the usage automaton `plugin-cookie`, above on the right, which introduces further peculiar features of Jalapa: parameters and guards. We start from the state `q0`. The event `init(p,u)`, signalling the creation of a new plugin `p` with codebase URL `u`, causes a transition to `q1`. Upon a `start(p)`, i.e. when `p` is launched by the browser, we reach `q2`. There, all the accesses to a cookie having a URL different from `u` lead to the offending state. When the control is transferred to another plugin, we reset the state to `q1`. At run-time, the policy `plugin-cookie` is enforced for all the possible instantiations of the formal parameters `p`, `u` and `u'`. Since this policy spans over multiple activations of plugins, we enforce it globally throughout the execution of the browser.

Once the needed policies and sandboxes have been defined, the next step is to instrument the compiled program with the hooks from the security-relevant methods to the execution monitor. Our tool implements this step as a bytecode transformation, discussed in more detail below. The resulting bytecode will respect all the usage policies at hand, within their scopes (see [10] for usage details). In [2] we formally prove that the run-time mechanism implemented by Jalapa is sound and complete w.r.t. the specification of policy compliance.

The Jalapa bytecode instrumentator. Our approach to code instrumentation is based on class wrapping, at the bytecode level. Since this solution suffers from some known issues, when moving to a production implementation we plan to follow a bytecode rewriting approach *à la* Kava [12]. First, we detect the set $\mathcal{M}$ of all the methods involved in policies. We inspect the bytecode, starting from the methods used in the aliases, and then computing a transitive closure through the inheritance graph. We create a *wrapper* for each of these methods. A wrapper $\mathtt{W}_\mathtt{C}$ for the class `C` declares exactly the same methods of `C`, implements all the interfaces of `C`, and extends the superclass of `C`. Indeed, $\mathtt{W}_\mathtt{C}$ can replace `C` in any context, in that it admits the same operations of `C`. A method `m` of $\mathtt{W}_\mathtt{C}$ can be either monitored or not. If the corresponding method `m` of `C` does not belong to $\mathcal{M}$, then $\mathtt{W}_\mathtt{C}$`.m` simply calls `C.m`. Otherwise, $\mathtt{W}_\mathtt{C}$`.m` calls the `PolicyPool.check` method that controls whether `C.m` can actually be executed without violating the active policies. A further step substitutes (the references to) the newly created classes for (the references to) the original classes. Finally, the instrumented code is linked to the Jalapa run-time support, i.e. a library that contains the resources necessary to the monitoring process. Note that our instrumentation produces a stand-alone application, requiring no custom JVM and no further external components.

The Jalapa runtime environment. The core of the enforcement mechanism is the method `PolicyPool.check()` that, for each active policy, tracks the states of all the needed usage automata. The state of the monitor is a mapping from policies to sets of pairs $((\mathtt{O}_1, \ldots, \mathtt{O}_\mathtt{k}), \mathtt{Q})$, where $(\mathtt{O}_1, \ldots, \mathtt{O}_\mathtt{k})$ is a tuple of weak references to the objects that substitute the formal parameters of the usage automaton, and `Q` is the current state of the usage automaton. Dummy instantiations are also maintained, to be concretized when new objects are discovered in the execution trace. When an object is garbage-collected, its occurrences in the mechanism state are reverted to dummies. If no usage automaton reaches an offending state, the intercepted method call is forwarded to the actual target; otherwise, a security exception is thrown.

Supporting artifacts. Jalapa is an open-source project. The sources are available through a Subversion repository at SourceForge [10]. Some further supporting material is accessible through the project Web page:

- the Jalapa Tutorial, that provides programmers with a step-by-step guide for securing Java programs with local usage policies.
- a repository of example programs and policies, including a prototype implementation of a secure bulletin board system.
- the manual page of policies, that defines their syntax and semantics.
- the manual page of the Jalapa rewriter, that defines its command-line syntax.

3 Discussion: present and future challenges

The Jalapa project started as an applicative branch of more foundational work on history-based access control [3,4,5]. Porting this theoretical machinery to a concrete setting like Java posed several issues. While our original goals have been achieved to a fair degree by the current release of Jalapa, there is room for future improvements. We devise three main research directions: (1) increasing the expressive power of usage policies, (2) reducing the run-time overhead of the enforcement mechanism, and (3) developing programming tools and methodologies to facilitate writing secure programs with Jalapa.

For the first point, although our usage policies are quite general, they do not cover all the possible real-world scenarios. We would like to require e.g. that a given low-level method (e.g. a write-file) can only be invoked within the scope of some high-level method that securely manages the low-level one. This is the case e.g. of a change-password method that calls write-file to update passwords. The challenge is to improve the expressive power of usage policies, while keeping them clean and formally sound. A promising solution seems that of introducing aliases of the form `ev := C1.m1(...) { C2.m2(...) }`, meaning that the event `ev` is fired whenever the method `m1` of class `C1` is invoked within the scope of the method `m2` of class `C2`. Another improvement would be to allow policies to mention the values *returned* by methods. This can be done by generating "return" events, exposing these values.

For the second point, we are currently developing a static analyser for Java bytecode, to detect those policies that are always respected in all the possible executions of the application. The run-time enforcement can then be optimized, by discarding the wrappers, and the associated execution monitoring, for the methods involved in policies that are always respected. This static analysis can be split in two phases:

- in the first phase, we extract from the bytecode a *control flow graph*, and we transform it into a *history expression* [4]. This is a sort of context-free grammar, the language of which over-approximates all the possible traces of events that the analysed program can generate at run-time.
- in the second phase, we reduce the infinite-state system given by the history expression to an *equivalent* finite one, and check it against the usage policies mentioned by the sandboxes used in the program. This is done through a model-checker. Only the policies that do not pass model checking need to be enforced at run-time. This phase has been implemented by our LocUsT tool, which runs in polynomial time in the size of the extracted history expression. Further details about this phase can be found in [4,6].

For the third point, we are developing an Eclipse plugin that combines the previous items into a programming environment, with facilities for writing policies, sandboxing code, and for running the static analyses to discover which policies can be disregarded by the security monitor. The LocUsT model checker, a prototype of this first analysis phase, and a prototype of the Eclipse plugin are distributed along with the Jalapa sources through the SourceForge Subversion repository.

Acknowledgement

Research partially supported by EU-FETPI Project IST-2005-16004 Sensoria (Software Engineering for Service-Oriented Overlay Computers), by PRIN Project SOFT (Tecniche Formali Orientate alla Sicurezza), and by EU project IST-033817 GridTrust (Trust and Security for Next Generation Grids).

References

[1] Abadi, M. and C. Fournet, *Access control based on execution history*, in: *Proc. 10th Annual Network and Distributed System Security Symposium*, 2003.

[2] Bartoletti, M., *Usage automata* (2009), to appear in ARSPA-WITS.

[3] Bartoletti, M., P. Degano, G. L. Ferrari and R. Zunino, *Types and effects for resource usage analysis*, in: *Proc. Fossacs*, 2007.

[4] Bartoletti, M., P. Degano, G. L. Ferrari and R. Zunino, *Model checking usage policies*, in: *Proc. Trustworthy Global Computing*, 2008.

[5] Bartoletti, M., P. Degano, G. L. Ferrari and R. Zunino, *Semantics-based design for secure web services*, IEEE Transactions on Software Engineering **34** (2008).

[6] Bartoletti, M. and R. Zunino, *LocUsT: a tool for checking usage policies*, Technical Report TR-08-07, Dip. Informatica, Univ. Pisa (2008).

[7] Edjlali, G., A. Acharya and V. Chaudhary, *History-based access control for mobile code*, in: *Secure Internet Programming*, Lecture Notes in Computer Science **1603**, 1999.

[8] Fournet, C. and A. D. Gordon, *Stack inspection: theory and variants*, ACM Transactions on Programming Languages and Systems **25** (2003), pp. 360–399.

[9] Gong, L., "Inside Java 2 platform security: architecture, API design, and implementation," Addison-Wesley, 1999.

[10] *Jalapa: Securing Java with Local Policies*, `http://jalapa.sourceforge.net`.

[11] Skalka, C. and S. Smith, *History effects and verification*, in: *Proc. APLAS*, 2004.

[12] Welch, I. and R. J. Stroud, *Kava - using byte code rewriting to add behavioural reflection to Java*, in: *USENIX Conference on Object-Oriented Technology*, 2001.