Language-based access control

February 1, 2007
Access control for the Java platform

- Codes from different trust levels execute within the same runtime. Java Cards cardlets, J2ME midlets, J2SE applets
- Security architectures use dynamic monitoring checks
  - Java Card firewall
  - J2ME interactive permissions
  - J2SE stack inspection
Local checks vs global security property

A study of the major Java security architectures:

- Analysis of Java Card firewall [TSI’04]
- Inference of security interfaces for stack-inspection [JFP’05]
- New security model for interactive devices [Esorics’06]

Are local checks sufficient to ensure a global security property?
Stack inspection
The stack inspection mechanism

Dynamic access control mechanism (Java, .NET)

- Security policy $\text{Code} \rightarrow \mathcal{P}(\text{attr})$
  origin, signature

- Stack inspection primitive $\square \land \blacklozenge$

success  success  failure
Control graph model for libraries

\[\begin{align*}
n_0 &: \text{call} \\
n_1 &: \text{call} \\
n_2 &: \text{return} \\
n_3 &: \text{check}(\gamma) \\
n_4 &: \text{return}
\end{align*}\]

\[\gamma = F(\text{Manager}) \land F(\text{Accountant})\]
\[\varphi = \text{Crit} \implies F(\text{Manager}) \land F(\text{Accountant})\]

\[n_0, n_1 \mapsto \{\text{System}\}\]
\[n_2 \mapsto \{\text{System, Crit}\}\]
\[n_3, n_4 \mapsto \{\text{Manager}\}\]
Specification of secure contexts

- Secure call contexts

\[ \text{secure}(s, n_0) = \forall (s' : \text{Stack}). s : n_0 \xrightarrow{[s]} s' \Rightarrow s' \models \varphi \]

- Call contexts that permit node traversal

\[ \text{transits}(s, n) = \exists n', s : n \xrightarrow{[s]} + s : n' \]

- Call contexts that permit method returns

\[ \text{returns}(s, n) = \exists r, \text{is}(r) = \text{return} \land s : n \xrightarrow{[s]} s : r \]
Symbolic computation of secure contexts

- Constraint solving over a lattice of LTL formulae
- A weakest condition operator $\delta : LTL \rightarrow LTL$
  
  $$s \models \delta_n(\phi) \iff s:n \models \phi$$

- Flavor of the constraints to solve
  - Traversal of check nodes
    
    $$is(n) = check(\gamma)$$
    $$\tau_n \leftarrow \delta_n(\gamma)$$
  
  - Traversal of method calls
    
    $$n^{\text{inter}} \rightarrow m$$
    $$\tau_n \leftarrow \delta_n(\rho_m)$$

- Secure contexts for method calls
  
  $$n^{\text{inter}} \rightarrow m$$
  $$\sigma_n \Rightarrow \delta_n(\sigma_m)$$
Inference of secure call contexts

Weakest precondition over the call stack

\[ \text{secure}(s, n_0) = \forall (s' : \text{Stack}). s : n_0 \xrightarrow{[s]} s' \Rightarrow s' \models \varphi \]

is given by a LTL formulae

\[ \sigma_{n_0} = \neg F(\text{Accountant}) \lor F(\text{Manager}) \]

\[ \begin{align*}
\gamma &= F(\text{Manager}) \land F(\text{Accountant}) \\
\varphi &= \text{Crit} \Rightarrow F(\text{Manager}) \land F(\text{Accountant})
\end{align*} \]
Stack inspection: a long-standing effort


Access control for interactive devices
Current security model for interactive devices

Resources accesses is protected by permissions

- Signed applications
  permissions granted \textit{forever}
- unsigned applications
  permissions \textit{granted} & \textit{consumed} at resource access time

Drawback: a coarse-grained control of permissions

- Unsigned applications may flood the user with security screens
- Operators are reluctant to sign
Resource usage scenario (current model)
Inflexible usage of permissions

- permission is granted
- permission is consumed (resource access)
Resource usage scenario (enhanced model)
Towards a fine-grained control of permissions

- Permissions are granted in advance before resource access
- Permissions are assigned quotas
- Permissions denote sets of resources
- Permissions of different kinds are independent
Resource usage scenario (enhanced model)
Towards a fined-grained control of permissions

- Permissions are granted in advance before resource access
- Permissions are assigned quotas
- Permissions denote sets of resources
- Permissions of different kinds are independent
Resource usage scenario (enhanced model)
Towards a fined-grained control of permissions

▶ Permissions are granted in advance before resource access
▶ Permissions are assigned quotas
▶ Permissions denote sets of resources
▶ Permissions of different kinds are independent
Resource usage scenario (enhanced model)
Towards a fined-grained control of permissions

- Permissions are granted in advance before resource access
- Permissions are assigned quotas
- Permissions denote sets of resources
- Permissions of different kinds are independent
Resource usage scenario (enhanced model)
Towards a fined-grained control of permissions

- Permissions are granted in advance before resource access
- Permissions are assigned quotas
- Permissions denote sets of resources
- Permissions of different kinds are independent
Enforcement of the enhanced model

Programs will not use more permissions than they have been granted.

- Dynamic monitoring
  - runtime overhead
  - security exception

- Static enforcement
  - no runtime overhead
  - no security exception
### Enforcement of the enhanced model

Programs will not use more permissions than they have been granted.

- **Dynamic monitoring**
  - runtime overhead
  - security exception

- **Static enforcement**
  - no runtime overhead
  - no security exception
Enforcement of the enhanced model

Programs will not use more permissions than they have been granted.

- Dynamic monitoring
  - runtime overhead
  - security exception
- Static enforcement
  - no runtime overhead
  - no security exception
Program as control-flow graphs

- A permission centric control-flow graph
  - Permission nodes:
    \[
    \text{grant} : \text{Kind} \times \mathcal{P}(\text{Permission}) \times \mathbb{N} \cup \infty \\
    \text{consume} : \text{Permission}
    \]
  - Control-flow nodes: call, return, throw
- A model of execution

\[
\text{State} = \text{Stack}(\text{Node}), \text{Exception}?, \text{BagOf}(\text{Permission})
\]

\[
\text{Kind}(n) = \text{grant}(p, m) \quad n \xrightarrow{\text{intra}} n' \\
\frac{n:s, \epsilon, \pi \rightarrow n':s, \epsilon, \text{grant}(\pi)(p, m)}{}
\]

\[
\text{Kind}(n) = \text{call} \quad n \xrightarrow{\text{inter}} m \\
\frac{n:s, \epsilon, \pi \rightarrow m:n:s, \epsilon, \pi}{}
\]

\[
\text{Kind}(n) = \text{throw}(\text{ex}) \quad \forall h, n \xrightarrow{\text{ex}} h \\
\frac{n:s, \epsilon, \pi \rightarrow n:s, \text{ex}, \pi}{}
\]
Safe traces and permissions

1. Formalise the notion of safe traces
   
   Safe traces do not use more permissions than they have been granted

2. Prove the soundness theorem (Coq proof)

Theorem

$$\forall n \in \text{Node}, P_n \neq \text{Err} \Rightarrow \forall tr \in \text{Trace}, \text{Safe}(tr)$$
Static analysis of permission usage

- Compute an under-approximation of the permissions

  \[ P : \text{Node} \rightarrow \text{BagOf(Permission)} \]

- Greatest solution of a set of recursive constraints

  \[
  \begin{align*}
  \text{Kind}(n) = \text{grant}(p) & \quad n \xrightarrow{\text{intra}} n' \\
  P_{n'} \sqsubseteq_p \text{grant}(P_n)(p)
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Kind}(n) = \text{call} & \quad n \xrightarrow{\text{inter}} m \quad n \xrightarrow{\text{intra}} n' \\
  P_{n'} \sqsubseteq_p \text{R}_m(P_n)
  \end{align*}
  \]

⇒ Iterative constraint solving
Inter-procedural analysis

Constraints summarise the effect of method calls

$$\begin{align*}
  \text{Kind}(n) &= \text{grant}(p, m) \quad n \xrightarrow{\text{intra}} n' \\
  R_n^e &\sqsubseteq \text{grant}(p, m); R_{n'}^e
\end{align*}$$

$$\begin{align*}
  \text{Kind}(n) &= \text{return} \\
  R_n &\sqsubseteq \lambda\rho.\rho
\end{align*}$$

$$\begin{align*}
  \text{Kind}(n) &= \text{call} \quad n \xrightarrow{\text{inter}} m \quad n \xrightarrow{\text{intra}} n' \\
  R_n^e &\sqsubseteq R_m; R_{n'}^e
\end{align*}$$

$$\Rightarrow$$ Amenable to symbolic resolution
Further enhancements

- Language features
  permission objects, multi-threading
- Precise program models
  dataflow analyses, integer analyses
  (nasty interaction with multi-threading)
- Strengthened security policy
  Beyond *enforceable security properties* (eventually, all the permissions are consumed)
- Bytecode verifier for the security model
  trade-off verification power/efficiency