Un modelo dinámico de nointerferencia

Maximiliano Cristiá
Universidad Nacional de Rosario
Flowgate Security Consulting
Argentina
Email: mcristia@fceia.unr.edu.ar

12 de octubre 2007

Introducción

- El concepto de nointerferencia es una formalización *muy* abstracta orientada a resolver el problema de la confidencialidad en sistemas de cómputo.
- Originalmente fue propuesto por Goguen y Meseguer en 1982.
- Básicamente dice que un grupo de usuarios H no interfiere con un grupo de usuarios L si la actividad de H sobre el sistema no puede ser percibida por L.
  - En otras palabras, para los miembros de L no existen los miembros de H.

Seguridad militar

- Los militares fueron los primeros en preocuparse en cómo preservar un secreto dentro de un sistema de cómputo multiusuario.
- Ellos clasifican cada documento y cada persona con una *clase de acceso*, y autorizan a una persona a leer un documento si la clase de acceso de la primera es mayor que la del segundo.
  - Para simplificar, podemos pensar que las clases de acceso son números naturales; en cuyo caso se los suele llamar *niveles de seguridad*.

Seguridad militar (cont.)

- La idea es lograr eso, *eficientemente*, en un sistema de cómputo multiusuario; no es ni por asomo tan fácil como podría parecer.
  - Eficientemente: preservar compatibilidad de aplicaciones, transparencia para el programador y el usuario, número indefinido de niveles de seguridad.
- El concepto de nointerferencia se adapta muy bien a esa política de seguridad pues podemos dividir a cada grupo de usuarios según su clase de acceso y pedir no interferencia entre los superiores y los inferiores.
- Actualmente se considera que un modelo de nointerferencia es la solución al problema de la seguridad militar.

Aproximaciones actuales a la nointerferencia

- En la actualidad la nointerferencia se estudia según dos aproximaciones:
  - Estática. Análisis estático de código, sistemas de tipos, compiladores que certifican programas.
  - Dinámica. Un módulo subyacente controla la ejecución de cada proceso de manera tal de asegurar la nointerferencia.
- La forma estática es actualmente la línea dominante.
- Nosotros creemos que no es aplicable, eficientemente, a sistemas operativos de propósito general como UNIX o Windows, razón por la cual trabajamos sobre la forma dinámica.

Etiquetado de variables

- Una de las posibles soluciones es etiquetar cada variable de un proceso con la clase de acceso que tiene en cada momento.
- Para ello se recurre al modelo de control de flujo de la información de Denning (1976) o sus derivados.
- Sin embargo, no es posible hacerlo eficientemente en lenguajes de máquina.
- El problema aparece con los punteros.

Ejemplo

`x` guarda un valor secreto entre 1 y 5; este programa permite descubrirlo. La etiqueta `L` es no secreto y `H` secreto.

```
a := malloc_zip(5);
b : = a;
read(x);
for i := 0 to x - 1 do
  *(a + i) := 1;
endfor
for j := 1 to 5 do
  print(*b);
  b := b + 1;
endfor
```

Inicialmente:

`(0[L],0[L],0[L],0[L],0[L])`

Luego del primer `for`:

`(1[H],1[H],1[H],0[L],0[L])`

Si asumimos que la salida estándar es `L` los tres primeros "prints" fallarán pero los restantes no, por lo que un atacante puede deducir el valor de `x` contanto los ceros que ve.

Un modelo dinámico basado en virtualización

- El programa anterior muestra que no es posible controlar, eficientemente, el flujo de información de un proceso que debe acceder valores secretos y no secretos etiquetando variables.
- Por lo tanto, el modelo que nosotros proponemos se basa en "forkear" un proceso cuando quiere acceder información de un nivel superior al que accede hasta ese momento.
- De esta forma cada proceso accede información *hasta* cierto nivel.
- Además, todos los procesos que se originan de esta forma comparten la entrada según el nivel que esta tenga.

¿Virtualización?

Decimos que el modelo se basa en virtualización porque el sistema se comporta como si hubiera una computadora por cada nivel de seguridad.

Por ejemplo, si uno ejecuta `vi secret_file public_file`, el sistema crea dos procesos para `vi`: uno accede a ambos archivos pero solo puede escribir en el archivo secreto (`H`), en tanto que el otro accede únicamente al archivo público (`L`) pero puede modificarlo.

Además, cuando el usuario escriba algo, si la entrada es pública, ambos procesos la recibirán.

De esta forma, parece como si cada proceso `vi` ejecutara en una computadora diferente, cada una de las cuales procesa información hasta cierto nivel, pero ambas conectadas a la misma E/S.

El modelo – La gramática del lenguaje

El modelo formal presentado en esta charla es muy abstracto y el único fin es mostrar sus elementos primordiales.

$$Expr ::= \mathbb{N} \mid VAR \mid *VAR \mid \&VAR \mid Expr \boxplus Expr$$

$$BasicSentence ::=$$

$$\quad \mathsf{skip} \mid var := Expr \mid *var := Expr \mid \mathsf{syscall}(\mathbb{N}, arg_1, \ldots, arg_n)$$

$$ConditionalSentence ::=$$

$$\quad \mathsf{if}\ Expr\ \mathsf{then}\ Program\ \mathsf{fi} \mid \mathsf{while}\ Expr\ \mathsf{do}\ Program\ \mathsf{done}$$

$$BasicAndConditional ::= BasicSentence \mid ConditionalSentence$$

$$Program ::= BasicAndConditional \mid Program\ ;\ Program$$

El modelo – Dos llamadas al sistema

Name	**System call**	**Meaning**
read(dev, var)	syscall(0, dev, var)	Reads var from input device dev
write($dev, expr$)	syscall(1, $dev, expr$)	Writes $expr$ to output device dev

El modelo – La semántica del lenguaje

$LS : Memory \to Program \to Memory$ donde $Memory : VAR \to \mathbb{N}$, $addr : \mathbb{N} \to VAR$, $\overrightarrow{x, M} = addr \circ M(x)$.

$$LS(M, \text{skip}) = M \qquad \text{(LS-skip)}$$

$$LS(M, x := e) = M \oplus \{x \mapsto eval(M, e)\} \qquad \text{(LS- :=)}$$

$$LS(M, *x := e) = M \oplus \{\overrightarrow{x, M} \mapsto eval(M, e)\} \qquad \text{(LS-}*\text{)}$$

$$LS(M, \text{if } e \text{ then } P \text{ fi}) = \mathbf{if}\ eval(M, e)\ \mathbf{then}\ LS(M, P)\ \mathbf{else}\ M \qquad \text{(LS-if)}$$

$$LS(M, \text{while } e \text{ do } P \text{ done}) = \mathbf{if}\ eval(M, e)\ \mathbf{then}\ LS(M, P \text{ ; while } e \text{ do } P \text{ done})\ \mathbf{else}\ M \qquad \text{(LS-while)}$$

$$LS(M, P_1 \text{ ; } P_2) = LS(LS(M, P_1), P_2) \qquad \text{(LS- ;)}$$

El modelo – Evaluación de expresiones

$M \in Memory$, $n \in \mathbb{N}$, $x \in VAR$, $e_1, e_2 \in Expr$

$$eval(M, n) = n \qquad \text{(eval-}\mathbb{N}\text{)}$$

$$eval(M, x) = M(x) \qquad \text{(eval-}VAR\text{)}$$

$$eval(M, *x) = M(\overrightarrow{x, M}) \qquad \text{(eval-}*\text{)}$$

$$eval(M, \&x) = addr^{-1}(x) \qquad \text{(eval-\&)}$$

$$eval(M, e_1 \boxplus e_2) = eval(M, e_1) \boxplus eval(M, e_2) \qquad \text{(eval-}\boxplus\text{)}$$

El modelo – Máquina controladora

$$SM : SState \times Env \times Memory \times Program \\ \rightarrow SState \times Env \times Memory$$

where $SState$ and Env are defined by:

$$DEV ::= il \mid ih \mid ol \mid oh$$
$$LEVEL ::= L \mid H$$
$$SState \triangleq [m : Memory, dl : DEV \rightarrow LEVEL]$$
$$Env == DEV \rightarrow \text{ seq } \mathbb{N}$$

La intención es que en la memoria de SM se van a procesar datos H mientras que en la memoria de $SState$ se van a procesar los datos L.

El modelo – Reglas para SM

Se definen inductivamente según la gramática del lenguaje.

$$SM(S, E, M, \text{skip}) = (S, E, M) \qquad \text{(SM-skip)}$$

$$SM(S, E, M, x := expr) = ([m \leftarrow LS(S.m, x := expr), dl \leftarrow S.dl], E, LS(M, x := expr)) \qquad \text{(SM- :=)}$$

$$SM(S, E, M, *x := expr) = ([m \leftarrow LS(S.m, *x := expr), dl \leftarrow S.dl], E, LS(M, *x := expr)) \qquad \text{(SM-*)}$$

$$SM(S, E, M, P_1 \ ; \ P_2) = SM(SM(S, E, M, P_1), P_2) \qquad \text{(SM- ;)}$$

$$SM(S, E, M, \mathsf{read}(il, x)) =$$
$$([m \leftarrow S.m \oplus \{x \mapsto head \circ E(il)\}, dl \leftarrow S.dl], \qquad (\text{SM-read}(il))$$
$$E \oplus \{il \mapsto tail \circ E(il)\}, M \oplus \{x \mapsto head \circ E(il)\})$$

$$SM(S, E, M, \mathsf{read}(ih, x)) =$$
$$(S, E \oplus \{ih \mapsto tail \circ E(ih)\}, M \oplus \{x \mapsto head \circ E(ih)\})$$
$$(\text{SM-read}(ih))$$

$$SM(S, E, M, \mathsf{write}(ol, e)) =$$
$$(S, E \oplus \{ol \mapsto \langle eval(S.m, e)\rangle \frown E(ol)\}, M) \qquad (\text{SM-write}(ol))$$

$$SM(S, E, M, \mathsf{write}(oh, e)) =$$
$$(S, E \oplus \{oh \mapsto \langle eval(M, e)\rangle \frown E(oh)\}, M) \qquad (\text{SM-write}(oh))$$

$$SM(S, E, M, \text{if } e \text{ then } P \text{ fi}) =$$

$$\begin{cases} SM(S, E, M, P) & \textbf{if } eval(S.m, e) \wedge eval(M, e) \\ (SM(S, E, M, P).1, \\ \quad SM(S, E, M, P).2, M) & \textbf{if } eval(S.m, e) \wedge \neg eval(M, e) \\ (S, E', SM(S, E, M, P).3) & \textbf{if } \neg eval(S.m, e) \wedge eval(M, e) \\ (S, E, M) & \textbf{if } \neg eval(S.m, e) \wedge \neg eval(M, e) \end{cases}$$

$$\text{where } E' = E \oplus \{ih \mapsto SM(S, E, M, P).2(ih), \\ oh \mapsto SM(S, E, M, P).2(oh)\}$$

(SM-if)

Let $\mathcal{P}While$ be P ; while e do P done

$SM(S, E, M,$ while e do P done$) =$

$$\begin{cases} SM(S, E, M, \mathcal{P}While) & \textbf{if } eval(S.m, e) \wedge eval(M, e) \\ (SM(S, E, M, \mathcal{P}While).1, \\ \quad SM(S, E, M, \mathcal{P}While).2, M) & \textbf{if } eval(S.m, e) \wedge \neg eval(M, e) \\ (S, E', SM(S, E, M, \mathcal{P}While).3) & \textbf{if } \neg eval(S.m, e) \wedge eval(M, e) \\ (S, E, M) & \textbf{if } \neg eval(S.m, e) \wedge \neg eval(M, e) \end{cases}$$

where $E' = E \oplus \{ih \mapsto SM(S, E, M, \mathcal{P}While).2(ih),$
$oh \mapsto SM(S, E, M, \mathcal{P}While).2(oh)\}$

(SM-while)

Teorema de nointerferencia

El modelo presentado verifica el siguiente teorema.

Theorem. $Noninterference$

$$\forall S \in SState; E_1, E_2 \in Env; M_1, M_2 \in Memory; P \in Program \bullet$$
$$P \text{ terminates}$$
$$\wedge S.dl = \{il \mapsto L, ol \mapsto L, ih \mapsto H, oh \mapsto H\}$$
$$\wedge E_1(il) = E_2(il)$$
$$\wedge E_1(ol) = E_2(ol)$$
$$\wedge SM(S, E_1, M_1, P) = (S_1', E_1', M_1')$$
$$\wedge SM(S, E_2, M_2, P) = (S_2', E_2', M_2')$$
$$\Longrightarrow E_1'(ol) = E_2'(ol)$$

Implementación

El modelo está siendo implementado por Pablo Mata como tesis de Licenciatura sobre Linux 2.6 como un módulo de seguridad utilizando Linux Security Modules.

En la implementación las dos memorias corresponden a las memorias de dos procesos que surgen al dividir un proceso que accede información H. La división se lleva a cabo cuando el proceso L solicita al núcleo un read() sobre un recurso H.

A partir de ese momento el núcleo comunica la entrada L a ambos procesos.

La implementación permite trabajar con un número mayor de niveles de seguridad.