Sécurité des applications

Josée Desharnais, Nadia Tawbi

Université Laval, Québec, Canada

Données massives 2015

Étudiants

- Andrew Bedford
- Sébastien Garvin
- Théophane Godonou
- Erwanne Kenyabwero
- Jean-Philippe Lachance
- David Landry
- Jean-Claude Simo
- Neda Torabian

Contexte

- Les applications informatiques sont ominiprésentes
- Le traitement des données passe par des applications informatiques
- Nous exécutons inévitablement des applications provenant de source non fiables.
- Dans un contexte de données massives, la sécurité est un enjeu majeur
  - confidentialité
  - intégrité
- Les mécanismes existants sont insuffisants
  - Anti-virus : limités
  - Mécanismes de contrôle d'accès : pas de suivi

Sécurité des applications

- Concevoir des mécanismes permettant de
  - reconnaître un code malicieux non répertorié
  - garder une flexibilité : ne pas imposer des contraintes trop restrictives.
  - ne pas rejeter trop d'applications
  - ne pas alourdir l'exécution (en insérant trop de vérification)
  - ne pas modifier la sémantique des applications sécuritaires.
- Prouver formellement que ces mécanismes garantissent la sécurité.

Axe de recherche: Sécurité basée sur les langages de programmation, "Language based security".

Sécurité des applications

- Concevoir des mécanismes permettant de
  - reconnaître un code malicieux non répertorié
  - garder une flexibilité : ne pas imposer des contraintes trop restrictives.
  - ne pas rejeter trop d'applications
  - ne pas alourdir l'exécution (en insérant trop de vérification)
  - ne pas modifier la sémantique des applications sécuritaires.
- Prouver formellement que ces mécanismes garantissent la sécurité.

Axe de recherche: Sécurité basée sur les langages de programmation, "Language based security".

Sécurité des applications

- Concevoir des mécanismes permettant de
  - reconnaître un code malicieux non répertorié
  - garder une flexibilité : ne pas imposer des contraintes trop restrictives.
  - ne pas rejeter trop d'applications
  - ne pas alourdir l'exécution (en insérant trop de vérification)
  - ne pas modifier la sémantique des applications sécuritaires.
- Prouver formellement que ces mécanismes garantissent la sécurité.

Axe de recherche: Sécurité basée sur les langages de programmation, "Language based security".

Que veut-on garantir?

Définir formellement ce qu'on veut garantir

- La non-interférence : une information d'une source privée ne doit pas influencer le contenu d'une destination publique.
- Principe de Bell-La Padula
  Confidentialité = "no read up, no write down "
  Intégrité nécessite "no read down, no write up".

Comment?

- Classer les sources et destinations d'information par niveaux de sécurité, par exemple privé, public.
- Inspecter le code ou surveiller l'exécution.

Que veut-on garantir?

Définir formellement ce qu'on veut garantir

- La non-interférence : une information d'une source privée ne doit pas influencer le contenu d'une destination publique.
- Principe de Bell-La Padula
  Confidentialité = "no read up, no write down "
  Intégrité nécessite "no read down, no write up".

Comment?

- Classer les sources et destinations d'information par niveaux de sécurité, par exemple privé, public.
- Inspecter le code ou surveiller l'exécution.

Que veut-on garantir?

Définir formellement ce qu'on veut garantir

- La non-interférence : une information d'une source privée ne doit pas influencer le contenu d'une destination publique.
- Principe de Bell-La Padula
  Confidentialité = "no read up, no write down "
  Intégrité nécessite "no read down, no write up".

Comment?

- Classer les sources et destinations d'information par niveaux de sécurité, par exemple privé, public.
- Inspecter le code ou surveiller l'exécution.

Mécanismes de contrôle du flot d'information

- Mécanismes statiques, souvent basés sur l'analyse de code par systèmes de types: très conservateurs, en cas de doute rejet de l'application, faux positifs.
- Mécanismes dynamiques, surveiller le programme durant l'exécution: ne peuvent pas tout détecter à cause de la nature de la propriété, faux négatifs.
- Nous nous intéressons aux approches hybrides basées sur le typage et sensibles au flot de données
  - L'application est sécuritaire, pour toute exécution possible: accepter.
  - L'application pourrait être non sécuritaire: instrumenter.
  - L'application n'est certainement pas sécuritaire: rejeter.

Mécanismes de contrôle du flot d'information

- Mécanismes statiques, souvent basés sur l'analyse de code par systèmes de types: très conservateurs, en cas de doute rejet de l'application, faux positifs.
- Mécanismes dynamiques, surveiller le programme durant l'exécution: ne peuvent pas tout détecter à cause de la nature de la propriété, faux négatifs.
- Nous nous intéressons aux approches hybrides basées sur le typage et sensibles au flot de données
  - L'application est sécuritaire, pour toute exécution possible: accepter.
  - L'application pourrait être non sécuritaire: instrumenter.
  - L'application n'est certainement pas sécuritaire: rejeter.

Mécanismes de contrôle du flot d'information

- Mécanismes statiques, souvent basés sur l'analyse de code par systèmes de types: très conservateurs, en cas de doute rejet de l'application, faux positifs.
- Mécanismes dynamiques, surveiller le programme durant l'exécution: ne peuvent pas tout détecter à cause de la nature de la propriété, faux négatifs.
- Nous nous intéressons aux approches hybrides basées sur le typage et sensibles au flot de données
  - L'application est sécuritaire, pour toute exécution possible: accepter.
  - L'application pourrait être non sécuritaire: instrumenter.
  - L'application n'est certainement pas sécuritaire: rejeter.

Prendre en compte les flots explicites et implicites

Flot explicite:

```
x := secret
send x to publicFile;    (* envoi non sécuritaire *)
```

Flot implicite:

```
if secret > 0 then
    send 42 to publicFile  (* envoi non sécuritaire *)
end
```

Prendre en compte les flots explicites et implicites

Flot explicite:

```
x := secret
send x to publicFile;    (* envoi non sécuritaire *)
```

Flot implicite:

```
if secret > 0 then
    send 42 to publicFile  (* envoi non sécuritaire *)
end
```

Prendre en compte le flot d'information

Ce programme risque d'être rejeté

```
1.   x := secret
     ⋮
5.   x := 0
6.   send x to publicFile  (* envoi sécuritaire *)
```

Et si on ne sait pas ?

```
if x > 0
    then d := privateFile
     else d := publicFile
end;
send secret to d      (* on ne peut dire *)
```

Les flux dus au progrès de l'exécution

Une boucle qui termine, ou non, peut révéler de l'information

```
while secret > 0 do
    skip
end
send 42 to publicFile      (* envoi non sécuritaire *)
```

The core language

$$
\begin{array}{lll}
\textit{(instructions)} & p ::= & e \mid c \\
\textit{(expressions)} & e ::= & x \mid n \mid \mathit{nch} \mid e_1 \ \mathbf{op} \ e_2 \mid \text{read } x \\
\textit{(commands)} & c ::= & x := e \mid \\
 & & \mathbf{skip} \mid \\
 & & \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{end} \mid \\
 & & \mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{end} \mid \\
 & & c_1; c_2 \mid \\
 & & \mathbf{send}\ x_1\ \mathbf{to}\ x_2
\end{array}
$$

Quelques règles de sémantique

(comment le langage est exécuté)

$$(\textsc{Assign})\ \frac{m(e) = r}{\langle x := e, m, o\rangle \longrightarrow \langle \textbf{stop}, m[x \mapsto r], o\rangle}$$

$$(\textsc{Send})\ \frac{m(x_1) = v \in \mathbb{Z} \qquad m(x_2) = ch \in \mathcal{C}}{\langle \textbf{send } x_1 \textbf{ to } x_2, m, o\rangle \longrightarrow \langle \textbf{stop}, m[ch \mapsto v], o :: (v, ch))\rangle}$$

Figure: Sémantique du langage I

Quelques règles de sémantique

(comment le langage est exécuté)

$$\text{(If)}\quad \frac{m(e) \neq 0 \Longrightarrow i = 1 \qquad m(e) = 0 \Longrightarrow i = 2}{\langle \textbf{if } e \textbf{ then } cmd_1 \textbf{ else } cmd_2 \textbf{ end}, m, o\rangle \longrightarrow \langle cmd_i, m, o\rangle}$$

$$\text{(Loop1)}\quad \frac{m(e) \neq 0}{\langle \textbf{while } e \textbf{ do } cmd \textbf{ end}, m, o\rangle \longrightarrow \langle cmd; \textbf{while } e \textbf{ do } cmd \textbf{ end}, m, o\rangle}$$

$$\text{(Loop2)}\quad \frac{m(e) = 0}{\langle \textbf{while } e \textbf{ do } cmd \textbf{ end}, m, o\rangle \longrightarrow \langle \textbf{stop}, m, o\rangle}$$

Figure: Sémantique du langage II

Quelques règles de typage et instrumentation

Vérification du code et insertion de tests si nécessaire

$$(\text{S-ASSIGN})\ \frac{\Gamma \vdash e : \sigma_{\ell_e}}{\Gamma, pc, hc \vdash x := e : \ T, hc, \Gamma[x \mapsto \sigma_{pc \sqcup \ell_e}],\ \text{genassign}}$$

$$(\text{S-SEND})\ \frac{\Gamma(x) = int_{\ell_x} \qquad \Gamma(c) = (int_{\ell}\ chan)_{\ell_c} \qquad (pc \sqcup hc \sqcup \ell_x \sqcup \ell_c) \sqsubseteq_m \ell}{\Gamma, pc, hc \vdash \textbf{send}\ x\ \textbf{to}\ c : T, hc \sqcup \ell_c, \Gamma,\ \text{gensend}}$$

Figure: Règles de typage et instrumentation

Quelques règles de typage et instrumentation

Vérification du code et insertion de tests si nécessaire

$$(\text{S-If})\ \frac{\begin{array}{c}\Gamma \vdash e : int_{\ell_e} \qquad h_3 = \sqcup_{j\in\{1,2\}} d(\Gamma, pc \sqcup \ell_e, cmd_j) \\ \bot \notin \text{ran}(\Gamma_1 \sqcup \Gamma_2) \qquad h = (h_1 \sqcup h_2 \sqcup h_3 \sqcup level(t_1 \oplus_{\ell_e} t_2)) \\ \Gamma, pc \sqcup \ell_e, hc \vdash cmd_j : t_j, h_j, \Gamma_j, [\![cmd_j]\!] \quad j \in \{1,2\}\end{array}}{\Gamma, pc, hc \vdash \textbf{if } e \textbf{ then } cmd_1 \textbf{ else } cmd_2 \textbf{ end} : (t_1 \oplus_{\ell_e} t_2), h, \Gamma_1 \sqcup \Gamma_2, \text{genif}}$$

$$(\text{S-Loop})\ \frac{\begin{array}{c}O(e, cmd, \Gamma \sqcup \Gamma') = t_o \qquad h = d(\Gamma, pc \sqcup \ell_e, cmd) \qquad \ell_t = level(t) \\ \ell_o = level(t_o) \qquad \bot \notin \text{ran}(\Gamma \sqcup \Gamma') \qquad \Gamma \sqcup \Gamma' \vdash e : int_{\ell_e} \\ \Gamma \sqcup \Gamma', (pc \sqcup \ell_e), (hc \sqcup \ell_t \sqcup h') \vdash cmd : t, h', \Gamma', [\![cmd]\!]\end{array}}{\Gamma, pc, hc \vdash \textbf{while } e \textbf{ do } cmd \textbf{ end} : t_o, h \sqcup h' \sqcup \ell_o, \Gamma \sqcup \Gamma', \text{genwhile}}$$

Figure: Règles de typage et instrumentation (suite)

Exemple

Instrumentation d'un envoi dangereux

gensend =
  if _pc $\sqcup$ _hc $\sqcup$ x_{lev} $\sqcup$ c_{lev} $\sqsubseteq$ c_{ch} **then**
    (**send** x **to** c)
  else
    fail
  end;
  _hc := _hc $\sqcup$ c_{lev};

```
if  lVal > 0  then
  c := lChan
else
  c := hChan
end;
x := 1;
uv := read c;
if  uv > 0 then
  send  lVal to lChan
else
  x := 2
end;
send lVal to lChan;
send  x to lChan
```

Here is its instrumentation :

```
(*initialization*)
_pc := L;
_hc := L;

(*if*)
_oldpc1 := _pc;
if lVal > 0 then
  _pc := _pc ⊔ lVal_lev ⊔ L;
  _hif1 := _pc;
  (*assign*)
  (c, c_ch, c_lev) := (lChan, lChan_ch, _pc ⊔ lChan_lev);

  _hc := _hc ⊔ _hif1;
  _hc := _hc ⊔ L;
  (c, c_lev) := (c, c_lev ⊔ _pc)
else
  _pc := _pc ⊔ lVal_lev ⊔ L;
  _hif1 := _pc;
  (*assign*)
  (c, c_ch, c_lev) := (hChan, hChan_ch, _pc ⊔ hChan_lev);

  _hc := _hc ⊔ _hif1;
  _hc := _hc ⊔ L;
  (c, c_lev) := (c, c_lev ⊔ _pc)
end
_pc := _oldpc1;
```

```
else
  _hif6 := (_pc⊓(¯lChan_ch⊔lChan_lev));
  (*assign*)
  (x, x_ch, x_lev) := (2, L, _pc ⊔ L)

  _hc := _hc ⊔ _hif6;
  _hc := _hc ⊔ L;
end
_pc := _oldpc6;

(*send*)
if _hc⊔_pc⊔lVal_lev⊔lChan_lev ⊑ lChan_ch then
  (send lVal to lChan)
else fail end;
_hc := _hc ⊔ lChan_lev;

(*send*)
if _hc⊔_pc⊔x_lev⊔lChan_lev ⊑ lChan_ch then
  (send x to lChan)
else fail end;
_hc := _hc ⊔ lChan_lev;
```

Divergence: Commands after a loop that always diverges are ignored. Hence, the following program is statically safe.

```
while 1 do
  skip
end;
send hVal to lChan
```

Even if it is statically safe, it is instrumented :

```
(*initialization*)
_pc := L;
_hc := L;

(*while*)
_oldpc1 := _pc;
while 1 do
  _pc := _pc ⊔ L;
  _hwhile1 := (_pc ⊓ (L)) ;
```

Transfert de technologie

Collaboration avec Thalès

- Détection de Malware sous Android

Conclusion et Travaux futurs

- Assurer la confidentialité de façon précise n'est pas facile
- Faire des compromis : efficacité versus précision
- Étendre pour traiter la concurrence, les canaux temporels, la déclassification, ...