Sécurité par le typage

Le problème

- L'exécution d'un programme comporte un *danger* inhérent, car celui-ci peut être incorrect ou mal intentionné (cheval de Troie).
- On souhaite garantir le bon fonctionnement du système, ainsi que *l'intégrité* ou la *confidentialité* des données qu'il contient.
- Pour cela, on édicte des *politiques* de sécurité, qui restreignent l'ensemble des comportements considérés comme acceptables.
- Toute politique de sécurité repose typiquement sur une exigence élémentaire, dite de *memory safety*.

Approche traditionnelle: le *reference monitor*

- Le *noyau* d'un OS contrôle l'accès aux ressources (mémoire, périphériques), empêchant les processus d'interférer directement.
- On peut également instrumenter le programme que l'on souhaite contrôler (*inline reference monitor*), ce qui est potentiellement plus efficace.
- La politique ainsi imposée peut être uniquement *memory safety* – cf. l'outil *software fault isolation* – ou bien plus précise, par exemple décrite par un *automate de sécurité*.
- Toutefois, les politiques de flot d'information *échappent* à cette approche.

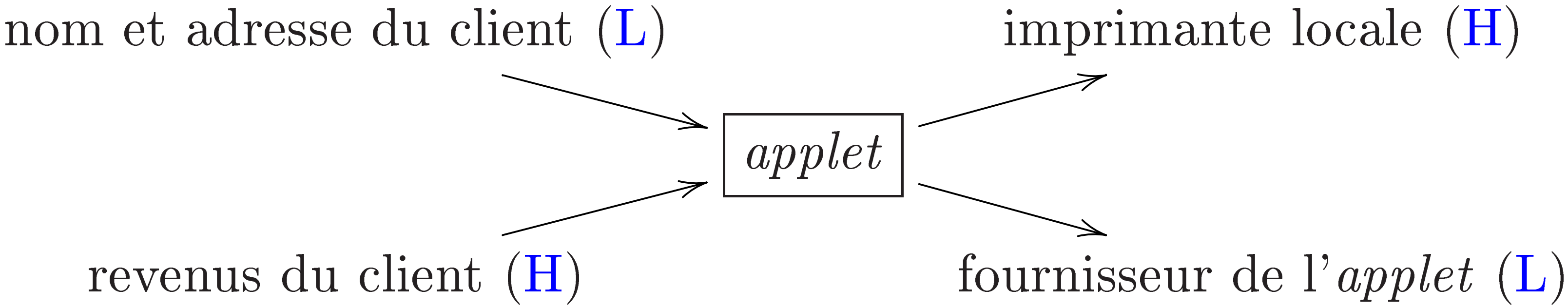
Approche à base de types: *language-based security*

- Si le programme est initialement exprimé dans un langage de haut niveau, on peut en contrôler le comportement à l'aide d'une discipline de *typage*.
- Tout code produit par un compilateur *correct* à partir d'un programme bien typé sera alors acceptable. (Si le code est fourni sous forme binaire, un mécanisme *d'authentification* sera nécessaire.)
- Mieux, un compilateur *certifiant* peut produire du code explicitement typé (*proof-carrying code, Java, TAL*).
- Cette approche *statique* permet de détecter le problème avant l'exécution (et même avant la distribution) du programme.

Que garantit usuellement le typage?

- Tout système de types *correct* (ou *fort*) garantit *memory safety*. Par exemple, la conversion d'un entier en un pointeur est interdite.
- La *structure lexicale* du langage permet de ne donner accès qu'à un sous-ensemble de l'environnement.
- Ainsi, une structure de données interne au système peut être rendue accessible uniquement à travers certaines fonctions, dont l'implantation peut alors imposer divers contrôles (*abstraction procédurale*).
- Enfin, un objet dont le type est *abstrait* pour le programme ne peut être ni créé, ni modifié par celui-ci, et se comporte donc comme une *capacité*.

Le flot d'information



Dans cette situation typique,

- Interdire l'un des deux flots situés au bas du diagramme serait trop restrictif.
- Les flots d'information autorisés sont décrits par un treillis de *niveaux de sécurité* $\{L \leq H\}$; la politique souhaitée est alors la *non-interférence*.
- Cette politique ne peut pas être imposée pendant l'exécution et requiert une approche *statique*.

Contrôle de flot d'information par le typage

- On utilise des types annotés par des niveaux d'information. La politique de sécurité est alors constituée par la donnée d'un *type*, que le programme doit satisfaire; ici, $\mathsf{string}^L \times \mathsf{string}^H \rightarrow \mathsf{string}^H \times \mathsf{string}^L$.
- Les règles de typage enrichies effectuent une série de vérifications, destinées à garantir que tout flot induit par le programme est conforme au treillis des niveaux, ici $\{L \leq H\}$.
- On démontre effectivement un énoncé de *non-interférence*: si le programme P admet le type $\mathsf{string}^H \rightarrow \mathsf{string}^L$, alors pour toutes chaînes de caractères x et y, les résultats $P(x)$ et $P(y)$ sont identiques. (Énoncé simplifié.)

État de l'art

- Andrew C. Myers (Cornell University) a publié une version de Java enrichie d'une analyse de flot d'information basée sur le typage (*JFlow/Jif*).
- Vincent Simonet et moi-même (INRIA) travaillons sur une version d'*OCaml* dotée d'une telle analyse. Le langage dispose de structures de données allouées dynamiquement et modifiables en place, de fonctions de première classe, d'exceptions, d'un système de types polymorphe, et d'un algorithme d'inférence de types.
- Notre étude théorique est essentiellement terminée (l'analyse et l'algorithme d'inférence ont été prouvés corrects). Un premier prototype est en passe de voir le jour.

Comment utilisera-t-on un tel outil?

- La librairie standard du langage fournira des structures de données, par exemple les listes, ainsi qu'une série de fonctions associées, par exemple `length`, de type $\forall \alpha\beta.(\alpha\ \mathsf{list}^{\beta}) \rightarrow \mathsf{int}^{\beta}$, ou encore `read`, de type $\forall\beta.\mathsf{file}^{\beta} \rightarrow \mathsf{string}^{\beta}$.
- Le type de certains des objets fournis par la librairie devra mentionner des niveaux d'information fixes; par exemple, `open` pourra avoir le type $\forall\beta \leq \mathsf{disk}.\mathsf{string}^{\beta} \rightarrow \mathsf{file}^{\mathsf{disk}}$.
- L'utilisateur spécifiera une relation d'ordre sur ces niveaux, et l'inférence déterminera (de façon conservative) si les flots induits par le programme la respectent. Par exemple, ouvrir un fichier dont le nom peut refléter un secret, c'est-à-dire dont le nom est de type string^{H}, ne sera possible que si $H \leq \mathsf{disk}$ a été spécifié.

Déclassification

- Toutefois, de nombreux programmes devront certainement violer, en quelques points, la politique imposée par le système de types.
- Le langage fournira un moyen, clairement identifiable syntaxiquement, de *déclassifier* une donnée. Les fragments de code qui en feront usage devront être vérifiés manuellement avec attention.
- L'emploi de la déclassification semble rendre impossible la conception d'un système à base de *proof-carrying code*. En d'autres termes, l'outil sera utile au *producteur* du code, mais le *client* devra se reposer, comme actuellement, sur un mécanisme d'authentification.

Quels résultats espérer?

Le respect de la politique de sécurité sera garanti par

- La correction de l'analyse et de l'algorithme d'inférence (prouvées manuellement).
- La correction du compilateur (actuellement non prouvée).
- L'innocuité des déclassifications effectuées par le programme.

Cela peut paraître encore trop, mais on peut néanmoins penser que l'outil sera utile, toute tentative d'analyse manuelle étant irréaliste.

Contacts: `Francois.Pottier@inria.fr`, `Vincent.Simonet@inria.fr`