Università degli Studi di Pisa

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Cybersecurity

Language-Based Technology for Security: Homework Assignment 1

Alessandro Niccolini | Giacomo Vitangeli | Piero Turri

Anno Accademico 2021-2022

Capitolo 1

Introduzione

L'obiettivo del progetto è quello di implementare un interprete che fa uso di astrazioni primitive per rendere sicura l'esecuzione del mobile code, utilizzando un linguaggio di programmazione funzionale (OCaml).

Il mobile code è un approccio di programmazione in cui i programmi sono considerati come dati, e quindi sono accessibili, trasmessi e valutati. Possiamo pensare al mobile code come ad un Applet Java ovvero codice che viene trasmesso attraverso una rete ed eseguito su una macchina remota.
Poiché gli sviluppatori di mobile code hanno poco o nessun controllo dell'ambiente in cui il loro codice verrà eseguito, speciali preoccupazioni di sicurezza diventano rilevanti. Infatti, l'adozione del codice mobile introduce uno scenario in cui le informazioni private e sensibili vengono spinte fuori dal perimetro di un ambiente di esecuzione affidabile, dove i benefici della libera condivisione del codice mobile hanno un costo: cercare di rendere il codice combinabile da più fonti può andare ad intaccare la sicurezza dell'applicazione.

In questo elaborato, ci riferiamo al codice mobile come ad un software che viene trasmesso su una rete eterogenea, attraversando domini di protezione, e viene eseguito automaticamente all'arrivo a destinazione.

Capitolo 2

Implementazione

L'interprete utilizza le funzionalità dello stack inspection tramite la primitiva execute, la quale va a valutare l'espressione passata come mobile code da eseguire. Per questo motivo l'interprete dispone dell'insieme di permessi ({}, {read, write}, {send}), e dei metodi per controllarli. Ciò permette l'esecuzione del mobile code solamente se i permessi sono abilitati, altrimenti l'esecuzione fallisce.
Nelle righe succesive sono descritti i comandi per compilare ed eseguire il programma.

```
$ ocamlc source.ml sandbox.ml main.ml
$ ./a.out
```

2.1 Permessi

I data types vengono ridefiniti, per essere gestiti dall'interprete del progetto piuttosto che dall'interprete di default di OCaml. Tra i vari data types si distinguono quelli relativi alla sicurezza, in particolare legati alla gestione dei permessi. Ad esempio *permissiont* può essere uno dei permessi tra lettura, scrittura o invio; il data type *permissionList* è una lista di permessi, utilizzata dai metidi di verifica dei permessi per controllare che il chiamante di una determinata funzione abbia i permessi richiesti.
Nella seguente figura è possibile vedere i permessi che troviamo all'interno dell'interprete, utilizzati per il meccanismo di controllo degli accessi.

```
(* tipo di permessi che possiamo trovare*)
type permissiont =
  | Pread
  | Pwrite
  | Psend;;
```

Figura 2.1: Permessi

2.2 Estensione del linguaggio

Il linguaggio è stato esteso per lavorare con: interi, boleani, stringhe, operazioni binarie let, if ed altre operazioni. Oltre ai soliti parametri la funzione *eval*, ovvero l'interprete, prende in input anche una lista di permessi che consente di specificare nella fase di dichiarazione della funzione, quali permessi sono necessari all'esecuzione.
La funzione execute non ha nelle proprie espressioni i permessi perché questi sono controllati durante la valutazione. Infatti, durante il *match* l'espressione *e* viene valutata. Se l'espressione corrisponde ad esempio al costrutto *If(e1,e2,e3)*, come vediamo nella seguente figura:

```
|If(e1,e2,e3) ->
    (let g = (eval e1 env secure) in
     match typecheck "bool" g , g with
     | true, Bool(true) -> eval e2 env secure
     | false, Bool(false) -> eval e3 env secure
     | _, _ -> raise (DynamicTypeError "La guardia non è di tipo booleano")
    )
```

Figura 2.2: Espressione IfThenElse

accade che la guardia e1 viene valutata tramite la funzione *eval*, nell'ambiente env, fornita della lista di permessi *secure*. Dopodiché, facciamo il typecheck della guardia per vedere se questa corrisponde ad un tipo corretto. Se il typecheck ha esito positivo, il *match* di *g* ci dice in quale ramo dobbiamo andare (then o else). Di conseguenza viene valutata l'espressione, *e2 o e3* a seconda del ramo preso; anche in questo caso viene fatta la valutazione nell'ambiente *env* con la lista dei permessi *secure*.