Langages de développement et sécurité

Mind your language

Éric Jaeger & Olivier Levillain, ANSSI/SDE

JFLA, 5 février 2013

ANSSI

Plan

1. Introduction
2. Illustrations
3. Et OCaml ?
4. Quelques éléments de conclusion

Rapide présentation de l'ANSSI et des labos

- le SGDSN, un service du premier ministre

Rapide présentation de l'ANSSI et des labos

- le SGDSN, un service du premier ministre
- l'ANSSI (agence nationale de la sécurité des systèmes d'information), créée en 2009

Rapide présentation de l'ANSSI et des labos

- le SGDSN, un service du premier ministre
- l'ANSSI (agence nationale de la sécurité des systèmes d'information), créée en 2009
- la sous-direction expertise

Rapide présentation de l'ANSSI et des labos

- le SGDSN, un service du premier ministre
- l'ANSSI (agence nationale de la sécurité des systèmes d'information), créée en 2009
- la sous-direction expertise
- les laboratoires
  - cryptologie
  - composants cryptographiques
  - sans-fil
  - architectures matérielles et logicielles
  - sécurité des réseaux et protocoles

Rappel préalable sur la sécurité

Les objectifs de sécurité s'expriment généralement en termes de confidentialité, intégrité et disponibilité, pour lesquels il ne faut pas avoir une lecture trop restrictive

- Confidentialité : chiffrement mais aussi contrôle d'accès ou encore maîtrise de la rémanence de l'information (*swap*)
- Intégrité : codes correcteur inappropriés, il faut l'immutabilité, le contrôle d'accès ou la signature cryptographique,
- Disponibilité : la redondance suffit rarement, la résilience nécessite surtout de la diversité

Enfin il est nécessaire de pouvoir évaluer ; en l'absence d'une évaluation indépendante on ne peut supposer avoir un niveau de sécurité approprié

Origine des travaux à l'ANSSI

En 2005, un industriel interroge la DCSSI afin de savoir si la langage JAVA peut être utilisé pour le développement de produits de sécurité

La question se révèle intéressante, et à généraliser

- Y a-t-il des langages de développement plus appropriés que d'autres pour un développement de sécurité ?
- Quels sont les critères d'analyse ?

Il semble n'y avoir que peu de travaux existants sur ce thème

Les travaux de l'ANSSI ont notamment menés à deux études

- JAVASEC (2008-2010) sur le langage JAVA
- LAFOSEC (2010-2012) sur les langages fonctionnels

Une citation

Extrait du discours de *C.A.R. Hoare* à l'occasion de la remise de son *Turing Award* en 1980

An unreliable programming language generating unreliable programs constitutes a far greatest risk to our environment and to our society than unsafe cars, toxic pesticides, or accidents at nuclear power stations. Be vigilant to reduce that risk, not to increase it.

Apparemment, il visait... ADA ! Nous retiendrons que le jugement porté sur un langage peut énormément varier selon les attentes

Langages et sécurité

La thématique sécurité est effectivement applicable aux langages

- Certains langages sont-ils plus appropriés que d'autres ?
  - Quelles sont les propriétés attendues ?
  - Veut-on déconseiller, recommander voire imposer un langage ?
- Faut-il plutôt raisonner sur les constructions d'un langage ?
  - Recommandations de codage
  - Robustesse des mécanismes proposés
- Qu'attend-on des outils tels que compilateur ou *runtime* ?
- Quels outils d'analyse sont utiles ou nécessaires à l'évaluation ?
- Quelle est la méthodologie d'évaluation à appliquer ?
- ...

Plan

1 Introduction

2 Illustrations

3 Et OCaml ?

4 Quelques éléments de conclusion

[Java] Charge static 1/2

Le comportement de certains programmes objet, même simples, est parfois difficilement prévisible : que fait le code suivant [1] ?

Source (snippets/java/StaticInit.java)

```
class StaticInit {
  public static void main(String[] args) {
    if (Mathf.pi-3.1415<0.0001)
      System.out.println("Hello world");
    else
      System.out.println("Hello strange universe");
  }
}
```

1. Indice : `Mathf.pi=3.1415`

[Java] Charge static 1/2

Le comportement de certains programmes objet, même simples, est parfois difficilement prévisible : que fait le code suivant [1] ?

Source (snippets/java/StaticInit.java)

```
class StaticInit {
  public static void main(String[] args) {
    if (Mathf.pi-3.1415<0.0001)
      System.out.println("Hello world");
    else
      System.out.println("Hello strange universe");
  }
}
```

Voici ce que donne l'exécution :

```
demo$ java StaticInit
Bad things happen!
```

1. Indice : `Mathf.pi=3.1415`

[Java] Charge static 2/2

L'explication du comportement de `StaticInit` se trouve dans `Mathf`

Source (snippets/java/Mathf.java)

En Java le chargement d'une classe exécute le code d'initialisation de classe, même en l'absence d'appel à une méthode ou à un constructeur

```
class Mathf {
  static double pi=3.1415;
  static { System.out.println("Bad things happen!");
           System.exit(0); }
}
```

Il semble possible que sur certaines implémentations de la Jvm une simple déclaration sans initialisation (`Mathf dummy;`) puisse avoir des effets similaires

Nous retiendrons surtout que l'exercice de relecture d'un code Java, même élémentaire, peut se révéler assez complexe et contre-intuitif !

[C] Faire mauvaise impression 1/2

Les manipulations de pointeur sont dangereuses et parfois cachées

Source (snippets/c/stringformat.c)

```
#include <stdio.h>

int sfa() {
 char *f="%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x";
 printf(f); printf("\n"); return 0;
}

int main(void) {
  int secret=0x40414243;
  sfa();
  return 0;
}
```

[C] Faire mauvaise impression 1/2

Les manipulations de pointeur sont dangereuses et parfois cachées

Source (snippets/c/stringformat.c)

```
#include <stdio.h>

int sfa() {
 char *f="%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x";
 printf(f); printf("\n"); return 0;
}

int main(void) {
  int secret=0x40414243;
  sfa();
  return 0;
}
```

`...8048469.40414243.8048460...` : le `printf` permet de parcourir à rebours la pile au-delà du contexte lié à l'appel de `sfa`

[C] Faire mauvaise impression 2/2

Avec la balise `%n` il est même possible de corrompre la pile

Source (snippets/c/stringformat2.c)

```
#include <stdio.h>

int main(void) {
  char *f="%x.%x.%x.%x.%x.%x.%n";
  int s=0x40414243;
  int *p=&s;
  printf(f); printf("\n");
  if (s==0x40414243) printf("Hello world\n");
  else printf("Bad things happen! Secret is %08x\n",s);
  return 0;
}
```

[C] Faire mauvaise impression 2/2

Avec la balise `%n` il est même possible de corrompre la pile

Source (snippets/c/stringformat2.c)

```
#include <stdio.h>

int main(void) {
  char *f="%x.%x.%x.%x.%x.%x.%n";
  int s=0x40414243;
  int *p=&s;
  printf(f); printf("\n");
  if (s==0x40414243) printf("Hello world\n");
  else printf("Bad things happen! Secret is %08x\n",s);
  return 0;
}
```

```
51b196.68dff4.51b225.2c1270.40414243.8048580.
Bad things happen! Secret is 0000002d
```

[JavaScript] Reconversion

À choisir, faut-il favoriser *cast* et surcharge, ou la préservation des propriétés usuelles telles que associativité et transitivité ?

Source (snippets/js/cast.js)

```
document.write("'0'",'0'==0?"==":"<>","0 and ");
document.write("0",0=='0.0'?"==":"<>","'0.0' and ");
document.write("'0'",'0'=='0.0'?"==":"<>","'0.0'<br />");

document.write(1+2+'X'); document.write(' and ');
document.write('X'+1+2); document.write(' and ');
document.write('X'+(1+2)); document.write('<br />');
```

[JavaScript] Reconversion

À choisir, faut-il favoriser *cast* et surcharge, ou la préservation des propriétés usuelles telles que associativité et transitivité ?

Source (snippets/js/cast.js)

```
document.write("'0'",'0'==0?"==":"<>","0 and ");
document.write("0",0=='0.0'?"==":"<>","'0.0' and ");
document.write("'0'",'0'=='0.0'?"==":"<>","'0.0'<br />");

document.write(1+2+'X'); document.write(' and ');
document.write('X'+1+2); document.write(' and ');
document.write('X'+(1+2)); document.write('<br />');
```

```
'0'==0 and 0=='0.0' and '0'<>'0.0'
3X and X12 and X3
```

Ici les constantes littérales aident à comprendre ; si à la place on utilise des variables, le comportement sera cryptique. . .

[B] La croisière s'amuse 1/2

Quelles garanties peuvent apporter les méthodes formelles ?

Source (snippets/b/airlock.b)

```
MACHINE Airlock

VARIABLES door1,door2
INVARIANT door1,door2:{open,locked} &
          ~(door1=open & door2=open)

INITIALISATION door1:=locked || door2:=locked
OPERATIONS
  open1 = IF door2=locked THEN door1:=open
  close1 = door1:=locked
  open2 = IF door1=locked THEN door2:=open
  close2 = door2:=locked
```

[B] La croisière s'amuse 1/2

Quelles garanties peuvent apporter les méthodes formelles ?

Source (snippets/b/airlock.b)

```
MACHINE Airlock

VARIABLES door1,door2
INVARIANT door1,door2:{open,locked} &
          ~(door1=open & door2=open)

INITIALISATION door1:=locked || door2:=locked
OPERATIONS
  open1 = IF door2=locked THEN door1:=open
  close1 = door1:=locked
  open2 = IF door1=locked THEN door2:=open
  close2 = door2:=locked
```

N'embarquez pas trop vite : un développeur malicieux peut ouvrir les deux portes tout en prouvant la conformité de son implémentation

[B] La croisière s'amuse 2/2

Ce n'est pas lié à une erreur dans la méthode formelle B mais à la sémantique du langage et de ses preuves ; en effet la preuve ne garantit pas que l'invariant est toujours vrai mais

- Qu'il est vrai à l'initialisation
- Que s'il est vrai avant l'appel d'une opération, alors il reste vrai après son exécution

Rien n'est dit sur les états intermédiaires ; une implémentation ouvrant les deux portes avant de les refermer est donc conforme

Au passage, le test automatisé reposant sur un oracle ne révèlera pas forcément ce type de problème ; par exemple pour un circuit électronique dont la sortie est validée par un signal `ack`, les états transitoires ne sont absolument pas considérés...

[Coq] *Vanitas, vanitas*

Des problèmes peuvent aussi venir d'une incohérence entre la théorie formelle et la concrétisation par traduction dans un langage compilable

Source (snippets/coq/emptyfalse.v)

```
Inductive Empty := onemore:Empty->Empty.

Theorem emptyfalse : forall (e:Empty), False.
Proof.
  intro e; induction e as [_ He]; apply He.
Qed.

Recursive Extraction Empty.
```

[Coq] *Vanitas, vanitas*

Des problèmes peuvent aussi venir d'une incohérence entre la théorie formelle et la concrétisation par traduction dans un langage compilable

Source (snippets/coq/emptyfalse.v)

```
Inductive Empty := onemore:Empty->Empty.

Theorem emptyfalse : forall (e:Empty), False.
Proof.
  intro e; induction e as [_ He]; apply He.
Qed.

Recursive Extraction Empty.
```

Si le type `empty` est vide en Coq, sa traduction en OCaml `type empty = | Onemore` ne l'est pas ; l'incohérence concerne également les valeurs rationnelles d'OCaml qui sont "hors modèle" en Coq

Plan

1 Introduction

2 Illustrations

3 Et OCaml ?

4 Quelques éléments de conclusion

Plaquette “publicitaire” des langages fonctionnels

Sur le papier, les langages fonctionnels semblent avoir des propriétés intéressantes pour les développements de sécurité

- Sémantique plus claire, souvent formalisée
- L'approche fonctionnelle impose d'expliciter les dépendances, réduit les effets de la stratégie d'évaluation
- Déclaration, allocation et initialisation (totale) indissociables
- Pas d'arithmétique des pointeurs
- Gestion automatisée de la mémoire (*Garbage collector*)
- Souvent typage statique fort, encapsulation, *etc.*

Malgré tout, une étude reste utile

- Comment prendre en compte certaines exigences de sécurité ?
- Quelle est la réalité — et la robustesse — des solutions ?

L'exemple du module cryptographique

Idéalement, on veut pouvoir coder une librairie cryptographique offrant des services de chiffrement, signature mais protégeant les clés

- Contrôle d'accès pour la protection en confidentialité et en intégrité – ne pouvant reposer sur la cryptographie
- Minimisation de la durée de présence en mémoire
- Maîtrise de la duplication
- Effacement sécurisé par surcharge

Un langage comme OCaml apporte quelques solutions – par exemple l'encapsulation – mais aussi son lot de problèmes (Gc)

Au passage, notons qu'un mécanisme non fonctionnel [2] tel que la surcharge peut aussi être victime d'optimisations de compilation, de mécanismes de cache, de technologies (mémoires *Flash*), *etc.*

2. C'est à dire sans effet visible sur les résultats de l'exécution

[OCaml] < mais costaud 1/4

Parlons donc encapsulation en OCaml [3]

Source (snippets/ocaml/hsm.ml)

```
module type Crypto = sig val id:int end;;

module C : Crypto  =
struct
  let id=Random.self_init(); Random.int 8192
  let key=(Random.self_init(); let s=Random.int 8192 in
           Printf.printf "< id=%d, key=%d >\n" id s; s)
end;;
```

À l'exécution, on obtient `< id=2570, secret=6151 >`

La valeur `id` est visible, alors que `secret` est masquée

`C.id;;` donne `- : int = 2570`

`C.key;;` donne `Error: Unbound value H.secret`

3. Ici les modules, sachant que les objets d'OCaml sont "fragiles"

[OCaml] < mais costaud 2/4

Cette encapsulation peut être contournée (sans même utiliser `Obj`)

Source (snippets/ocaml/hsmoracle.ml)

```
let rec oracle o1 o2 =
  let o = (o1 + o2)/2 in
  let module O = struct let id=C.id let key=o end in
  if (module O:Crypto)>(module C:Crypto)
  then oracle o1 o
  else (if (module O:Crypto)<(module C:Crypto)
        then oracle o o2
        else o);;

oracle 0 8192;;
```

À l'exécution, la valeur retournée est bien `- : int = 6151` ; les modules étant des valeurs en OCaml, on peut les comparer avec l'opérateur `<` qui bien que polymorphe fait une analyse structurelle !

[OCaml] < mais costaud 3/4

Détail amusant, en inversant les champs on obtient toujours 4096 [4]

Source (snippets/ocaml/hsm2.ml)

```
module type Crypto = sig val id:int end;;

module C : Crypto  =
struct
  let key=Random.self_init(); Random.int 8192
  let id=Random.self_init(); Random.int 8192
end;;

let rec oracle o1 o2 =
  let o = (o1 + o2)/2 in
  let module O = struct let key=o let id=C.id end in
  if (module O:Crypto)>(module C:Crypto)
  then oracle o1 o
  else (if (module O:Crypto)<(module C:Crypto)
        then oracle o o2 else o);;
```

4. L'ordre des champs dans un module a donc une sémantique observable

[OCaml] < mais costaud 4/4

Mais en fait à quoi bon s'embêter avec les champs ou les types ?

Source (snippets/ocaml/hsm5.ml)

```
module type Crypto = sig val id:int end;;

module C : Crypto  =
struct
  let id=Random.self_init(); Random.int 8192
  let key='k'; (* ASCII code 107 *)
end;;

let rec oracle o1 o2 =
  let o = (o1 + o2)/2 in
  let module O = struct let id=C.id let notachar=o end in
  if (module O:Crypto)>(module C:Crypto)
  then oracle o1 o
  else (if (module O:Crypto)<(module C:Crypto)
        then oracle o o2 else o);;
```

Ici `oracle 0 8192` renvoie `107` !

[OCaml] *Mutatis mutandis* 1/3

En OCaml le code est figé et les chaînes sont mutables ; qu'en est-il des chaînes apparaissant dans le code ?

Source (snippets/ocaml/mutable.ml)

```
let alert test =
  if test then "Tout va bien" else "Tout va mal!";;

alert true;;
alert false;;

(alert false).[8]<-'b'; (alert false).[9]<-'i';
(alert false).[10]<-'e'; (alert false).[11]<-'n';;

alert false;;
```

[OCaml] *Mutatis mutandis* 1/3

En OCaml le code est figé et les chaînes sont mutables ; qu'en est-il des chaînes apparaissant dans le code ?

Source (snippets/ocaml/mutable.ml)

```
let alert test =
  if test then "Tout va bien" else "Tout va mal!";;

alert true;;
alert false;;

(alert false).[8]<-'b'; (alert false).[9]<-'i';
(alert false).[10]<-'e'; (alert false).[11]<-'n';;

alert false;;
```

Voici ce qui s'affiche à l'exécution :

```
- : string = "Tout va bien"
- : string = "Tout va mal!"
- : string = "Tout va bien"
```

[OCaml] *Mutatis mutandis* 2/3

Il est important de comprendre dans l'exemple précédent que ce n'est pas une redéfinition de la fonction `alert` mais un simple effet de bord ; pour en être convaincu, voici que ce cela donne avec une fonction de la bibliothèque standard

Source (snippets/ocaml/mutablebool.ml)

```
(string_of_bool true).[0]<-'f';
(string_of_bool true).[1]<-'a';
(string_of_bool true).[3]<-'x';
Printf.printf "1=1 s'evalue a %b\n" (1=1);;
```

[OCaml] *Mutatis mutandis* 2/3

Il est important de comprendre dans l'exemple précédent que ce n'est pas une redéfinition de la fonction `alert` mais un simple effet de bord ; pour en être convaincu, voici que ce cela donne avec une fonction de la bibliothèque standard

Source (snippets/ocaml/mutablebool.ml)

```
(string_of_bool true).[0]<-'f';
(string_of_bool true).[1]<-'a';
(string_of_bool true).[3]<-'x';
Printf.printf "1=1 s'evalue a %b\n" (1=1);;
```

Bien entendu, à l'exécution nous obtenons `1=1 s'evalue a faux`
Cela ne marche pas avec `string_of_int`, mais d'autres fonctions semblent concernées, par exemple `Char.escaped`

[OCaml] *Mutatis mutandis* 3/3

Cela s'applique également aux exceptions. . . avec des *patterns* de développement qui deviennent dès lors dangereux alors qu'on les trouve dans la bibliothèque standard (*cf.* `char_of_int`)

Source (snippets/ocaml/mutableexc.ml)

```
let alert test =
  if test then failwith "minor" else failwith "major";;

let reaction test =
  try ignore (alert test)
  with Failure "minor" -> ()
     | Failure "major" -> failwith "major";;

try alert false with Failure x -> (x.[1]<-'i'; x.[2]<-'n');;

reaction false;;
```

[OCaml] *Mutatis mutandis* 3/3

Cela s'applique également aux exceptions. . . avec des *patterns* de développement qui deviennent dès lors dangereux alors qu'on les trouve dans la bibliothèque standard (*cf.* `char_of_int`)

Source (snippets/ocaml/mutableexc.ml)

```
let alert test =
  if test then failwith "minor" else failwith "major";;

let reaction test =
  try ignore (alert test)
  with Failure "minor" -> ()
     | Failure "major" -> failwith "major";;

try alert false with Failure x -> (x.[1]<-'i'; x.[2]<-'n');;

reaction false;;
```

À l'exécution, le `reaction false` ne génère plus d'exception ; le flot d'exécution a été modifié

Plan

1 Introduction

2 Illustrations

3 Et OCaml ?

4 Quelques éléments de conclusion

À propos de l'enseignement

Comment former un développeur ou un évaluateur de sécurité ?

- La sécurité n'est pas un module qui s'intègre parmi d'autres
- Savoir aller au-delà du fonctionnel (un plus court chemin)
  - L'attaquant cherche les erreurs, préconditions et valeurs observables mais pourtant hors modèle, *etc.*
  - Le développeur de sécurité doit imaginer tout ce qui peut mal tourner (conserver un seul chemin acceptable)
- Maîtriser les fondamentaux
  - Sémantique des langages
  - Théorie de la compilation
  - Principes des systèmes d'exploitation
  - Architecture des ordinateurs
  - ...

À propos des langages

Comment aider à l'amélioration de la sécurité ou l'assurance ?

- La spécification d'un langage est idéalement complète, déterministe et non ambiguë [5]
- *Simple is beautiful*
  - Ne conserver que ce qui est nécessaire
  - Éviter ce qui est complexe ou dénué de sens
  - Ne pas contrarier l'intuition ou la logique élémentaire
- Sans maîtrise, la puissance n'est rien
  - Faciliter la lisibilité et la traçabilité : un mot clé pour un concept, des notations cohérentes, *etc.*
  - Ne pas confondre aide au développeur avec laxisme ou devinettes
  - Introspection, évaluation, traits dynamiques rendent toute forme d'analyse "délicate"

5. Voire formalisée...

À propos des outils

Quels outils (ou options) pour la sécurité et l'assurance ?

- Ce qui n'est pas spécifié pour le langage devrait être interdit par les outils – ou au moins signalé
- Implémenter les vérifications possibles, et les faire au plus tôt
- Minimiser les manipulations silencieuses
- Savoir aller au-delà du fonctionnel
  - Le raisonnement de sécurité nécessite de penser au-delà des interfaces d'une boîte noire
  - Certaines optimisations sont inappropriées en sécurité
- Étendre le domaine des invariants de compilation [6]
  - Modèle mémoire reflétant l'encapsulation
  - Surveiller le flot d'exécution même en présence de fautes
- Disposer d'outils maîtrisés voire de confiance

6. Cela peut aussi concerner les architectures...

Remerciements

Les exemples de cette présentation ont étés fournis ou inspirés par

- les laboratoires de l'ANSSI
- les participants à l'étude JAVASEC
- les participants à l'étude LAFOSEC
- différents sites et blogs, notamment :
  - www.thedailywtf.com, www.xkcd.com
  - le site de la société MLSTATE
  - Sami Koivu (Slightly Random Broken Thoughts)
  - Jeff Atwood (Coding Horror)
  - Software Engineering Not At School
  - Functional Orbitz

Sans oublier, bien entendu, l'aimable collaboration des concepteurs des langages et des outils ;-)

Merci de votre attention

Avez-vous des questions ?