LaFoSec: Étude de la sécurité intrinsèque des langages fonctionnels [1]

Partie I sur IV

Présentation Générale de l'étude LaFoSec

Damien Doligez, Christèle Faure, Thérèse Hardin, Manuel Maarek

JFLA - Février 2013

1. Etude commanditée par l'Agence Nationale de la Sécurité des Systèmes d'Information

Étude LaFoSec

Langages **Fo**nctionnel **Sé**curité (**LaFoSec**)

Est-ce que les langages fonctionnels sont appropriés pour le développement d'application de sécurité ?

- Quels traits de ces langages favorisent/défavorisent la sécurité ?
- Est-ce que les implémentations de ces traits sont robustes à des attaques ? Pendant la compilation, l'exécution ?
- En pratique, ces langages sont-ils adaptés pour la sécurité ?
- Peut-on consolider la sécurité fournie par ces langages ?

Étude LaFoSec

Contexte

- Littérature abondante concernant le langage C
- Bases de données de vulnérabilités logicielles
  - CVE (peu de références aux langages fonctionnels)
  - OWASP (orienté développement Web)
- Résultats de l'étude JavaSec
- Peu de littérature et de vulnérabilités documentées concernant spécifiquement les langages fonctionnels

Déroulement
(tranche ferme)

Étude des langages

- Présentation de l'approche fonctionnelle : Erlang, Scheme, Haskell, Lisp, OCaml, F# et Scala
- Analyse de trois langages : OCaml, F# et Scala
- Analyse avancée de OCaml : modèle d'exécution, outils
- Élaboration de recommandations pour l'utilisation de OCaml
- Proposition de pistes d'évolution de OCaml

Déroulement
(tranche ferme)

Étude des langages

- Présentation de l'approche fonctionnelle : Erlang, Scheme, Haskell, Lisp, OCaml, F# et Scala
- Analyse de trois langages : OCaml, F# et Scala *(cette présentation)*
- Analyse avancée de OCaml : modèle d'exécution, outils *(cette présentation)*
- Élaboration de recommandations pour l'utilisation de OCaml *(voir présentation Partie II)*
- Proposition de pistes d'évolution de OCaml *(voir présentation Partie III)*

Déroulement
(tranches conditionnelles)

Expérimentation

- Développement en OCaml d'une application de sécurité : un prototype de générateur de valideurs XML en suivant les recommandations de sécurité

Évaluation

- Évaluation indépendante par un Centre d'Évaluation de la Sécurité des Technologies de l'Information (CESTI) : recherche de vulnérabilité

- Mise à jour des recommandations suivant les résultats de l'évaluation

Déroulement
(tranches conditionnelles)

Expérimentation

- Développement en OCaml d'une application de sécurité : un prototype de générateur de valideurs XML en suivant les recommandations de sécurité

(voir présentation Partie IV)

Évaluation

- Évaluation indépendante par un Centre d'Évaluation de la Sécurité des Technologies de l'Information (CESTI) : recherche de vulnérabilité

- Mise à jour des recommandations suivant les résultats de l'évaluation

Périmètre de l'étude

Langage et compilation

Tous les traits des langages OCaml, F$^{\#}$ et Scala

Modèles d'exécution

Uniquement ceux de OCaml :
modes natif, bytecode et par boucle interactive
NB L'étude ne porte pas sur les plateformes .NET et JVM

Environnement de développement

Outils associés au développement en OCaml
NB Les bibliothèques standard ou utilisateur sont hors du sujet de l'étude

Langage
OCaml 3.12.0

Impact du langage sur :

- Le contrôle de l'accès aux données
- Le contrôle de l'accès aux traitements

Apports

- Bonne lisibilité du code
- Typage statique
- Encapsulation (module, objet, fermeture)
- Contrôle des bornes

Inconvénients

- Contournement du typage
- Contournement de l'encapsulation
- Mutabilité des chaînes de caractères

Encapsulation

Un type abstrait est utilisé pour encapsuler une donnée confidentielle

```
module M : sig
  type secret
  val v : secret
end = struct
  type secret = ...
  let v = ...
end
module M : sig type secret val v : t end
```

Le type `M.secret` est abstrait, sa structure n'est pas publique
Seules des fonctions de `M` peuvent accéder aux sous-structures de `M.v`

Encapsulation et exceptions

Contournement de l'encapsulation par les exceptions


```
module M : sig type secret val v : t end
```

Un module malveillant peut utiliser les exceptions de plusieurs manières pour accéder à la donnée confidentielle `M.v`

Encapsulation et exceptions

Contournement de l'encapsulation par les exceptions


```
module M : sig type secret val v : t end

exception Exc of M.secret
raise (Exc M.v)
Fatal error: exception Exc(12345)
```

1. Une exception est définie avec un argument du type abstrait `M.secret`
   La levée de l'exception affiche partiellement ou complètement la valeur

Encapsulation et exceptions

Contournement de l'encapsulation par les exceptions


```
module M : sig type secret val v : t end

exception Exc of M.secret
Printexc.to_string (Exc M.v)
- : string = "Exc(12345)"
```

2. Une exception est définie avec un argument du type abstrait `M.secret`
   L'appel aux fonctions de `Printexc` peut révéler partiellement la valeur selon la structure du type

Encapsulation et exceptions

Contournement de l'encapsulation par les exceptions


```
module M : sig type secret val v : t end

exception Exc of M.secret
let e_secret = Exc M.v
val e_secret : exn = Exc <abstr>
exception Exc of int
```

3. Une première exception est définie avec en argument le type abstrait
   Puis cette exception est masquée par une autre définition d'exception ayant un argument de type public, qui permettra de découvrir la valeur cachée

Encapsulation et exceptions

Contournement de l'encapsulation par les exceptions


```
module M : sig type secret val v : t end

exception Exc of M.secret
let e_secret = Exc M.v
val e_secret : exn = Exc <abstr>
exception Exc of int
let rec recherche min max =
  let j = (max + min) / 2 in
  if e_secret = Exc j then print_int j
  else if e_secret > Exc j
    then recherche (j + 1) max
    else recherche min (j - 1)
val recherche : int -> int -> unit = <fun>
recherche 1 100000
12345- : unit = ()
```

Encapsulation et exceptions

Contournement de l'encapsulation par les exceptions


```
module M : sig type secret val v : t end

exception Exc of M.secret
let e_secret = Exc M.v
val e_secret : exn = Exc <abstr>
exception Exc of int
```

Les exceptions sont des valeurs de type `exn` représentées par un enregistrement contenant une chaîne de caractères pour l'étiquette et un champ par argument
La comparaison entre deux exceptions débute par la comparaison des chaînes des étiquettes, ici identiques

Contournement de l'encapsulation

Égalité, comparaison et hachage cassent l'abstraction


```
module M : sig
  type secret
  val construire : int -> secret
end = struct
  type secret = Secret of int
  let construire i = Secret i
end
module M : sig type secret val construire : int -> secret end
```

Le type `M.secret` est abstrait, sa structure n'est pas publique
Seules des fonctions de `M` peuvent accéder aux sous-structure des valeurs construites par `M.construire`

Contournement de l'encapsulation

Égalité, comparaison et hachage cassent l'abstraction


```
module M : sig
  type secret
  val construire : int -> secret
end = struct
  type secret = Secret of int
  let construire i = Secret i
end
let x = M.construire 2          let y = M.construire 25
val x : M.secret = <abstr>      val y : M.secret = <abstr>
x = y
- : bool = false
x < y
- : bool = true
```

Mais l'égalité et la comparaison sont possibles car polymorphes elles ne connaissent pas les barrières de l'abstraction

Contournement de l'encapsulation

Égalité, comparaison et hachage cassent l'abstraction


```
module M : sig
  type secret
  val construire : int -> secret
end = struct
  type secret = unit -> int
  let construire i = fun () ->  i
end
let x = M.construire 2          let y = M.construire 25
val x : M.secret = <abstr>      val y : M.secret = <abstr>
```

Encapsuler la valeur dans une fonction permet d'empêcher l'égalité et la comparaison

Contournement de l'encapsulation

Égalité, comparaison et hachage cassent l'abstraction


```
module M : sig
  type secret
  val construire : int -> secret
end = struct
  type secret = unit -> int
  let construire i = fun () ->  i
end
let x = M.construire 2          let y = M.construire 25
val x : M.secret = <abstr>      val y : M.secret = <abstr>
x = y
Exception: Invalid_argument "equal: functional value".
x < y
Exception: Invalid_argument "equal: functional value".
```

Une exception est levée à l'exécution

Contournement de l'encapsulation

Égalité, comparaison et hachage cassent l'abstraction


```
module M : sig
  type secret
  val construire : int -> secret
end = struct
  type secret = unit -> int
  let construire i = fun () ->  i
end
let x = M.construire 2          let y = M.construire 25
val x : M.secret = <abstr>      val y : M.secret = <abstr>
let z = M.construire 2
val z : M.secret = <abstr>
Hashtbl.hash x = Hashtbl.hash y
- : bool = false
Hashtbl.hash x = Hashtbl.hash z
- : bool = true
```

Le hachage permet encore d'identifier les valeurs identiques

Contournement de l'encapsulation
Module `Obj`

Les fonctions du module `Obj` permettent de contourner le typage

```
module M : sig
  type secret
  val v : secret
end = struct
  type secret = Secret of string * int
  let v = Secret ("secret", 12345)
end
Obj.magic M.v
- : 'a = <poly>
(Obj.magic M.v : string * int)
- : string * int = ("secret", 12345)
(Obj.magic ("intrusion",67890) : M.secret)
- : M.secret = <abstr>
```

L'attribution de types concrets à une valeur polymorphe peut aboutir à des valeurs concrètes et révéler la valeur initialement encapsulée si la structure du type attribué équivaut à celle du type abstrait

Contournement de l'encapsulation

Module `Obj`

Les fonctions du module `Obj` permettent de contourner le typage

```
let incr, reset =
  let init = 100 in
  let c = ref init in
  (fun () -> c:=!c + 1; !c),
  (fun () -> c:= init; !c)
incr (); incr (); incr()

(Obj.obj (Obj.field (Obj.field (Obj.repr incr) 1) 0) : int)
- : int = 103
```

Contournement de l'encapsulation

Module Obj

L'appel à la fonction `Obj.magic` n'est pas visible dans le code compilé

L'identification des utilisations du module `Obj` peut être faite par relecture de code ou à l'aide de Camlp4/Camlp5 par exemple ou encore par des outils comme Mascot

```
EXTEND Gram

  GLOBAL: a_UIDENT;

  a_UIDENT:
      [ [ `UIDENT "Obj" -> raise Use_of_Obj
        | `UIDENT s ->  s
      ] ];

END;
```

Contournement de l'encapsulation

Sérialisation et désérialisation


```
module M : sig
  type secret
  val v : secret
end = struct
  type secret = Secret of string * int
  let v = Secret ("secret", 12345)
end
module M : sig type secret val v : t end
```

Contournement de l'encapsulation

Sérialisation et désérialisation


```
module M : sig
  type secret
  val v : secret
end = struct
  type secret = Secret of string * int
  let v = Secret ("secret", 12345)
end
let v_out = Marshal.to_string M.v []
val v_out : string = "\132\149..."
let v_in = Marshal.from_string v_out 0
val v_in : 'a = <poly>
```

Une valeur sérialisée ne porte pas son type

Contournement de l'encapsulation

Sérialisation et désérialisation


```
module M : sig
  type secret
  val v : secret
end = struct
  type secret = Secret of string * int
  let v = Secret ("secret", 12345)
end
let v_out = Marshal.to_string M.v []
val v_out : string = "\132\149..."
let v_in = Marshal.from_string v_out 0
val v_in : 'a = <poly>
type structure = { s : string; i : int }
let v_in_structure : structure = v_in
val v_in_structure : structure = {s = "secret"; i = 12345}
```

Il est donc possible de lui attribuer un autre type pour révéler sa valeur concrète

Contournement de l'encapsulation

Sérialisation et désérialisation


```
module M : sig
  type secret
  val v : secret
end = struct
  type secret = Secret of string * int
  let v = Secret ("secret", 12345)
end
type structure = { s : string; i : int }
let intrus : M.secret =
  Marshal.from_string
    (Marshal.to_string { s = "intrusion"; i = 67890 } []) 0
val intrus : M.secret = <abstr>
```

Il est aussi possible de construire une valeur d'un type abstrait

Mutabilité des chaînes de caractères

Les chaînes de caractères sont mutables en OCaml

```
let s = "abcde"
val s : string = "abcde"
s.[3] <- 'X'
- : unit = ()
s
val s : string = "abcXe"
```

Mutabilité des chaînes de caractères

Une chaîne de caractères littérale définie dans le corps d'une fonction est donc modifiable si elle est retournée

```
let f () = "abcde"
val f : unit -> string = <fun>
let s = f ()
val s : string = "abcde"
s.[3] <- 'X'
- : unit = ()
let t = f ()
val t : string = "abcXe"
```

Mutabilité des chaînes de caractères

Une chaîne de caractères littérale définie dans le corps d'une fonction est donc modifiable si elle est retournée

```
let f () = String.copy "abcde"
val f : unit -> string = <fun>
let s = f ()
val s : string = "abcde"
s.[3] <- 'X'
- : unit = ()
let t = f ()
val t : string = "abcde"
```

Il est possible d'éviter ce type d'altération en retournant systématiquement des copies de littéraux

Mutabilité des chaînes de caractères

De la même manière, certaines chaînes de caractères littérales de la bibliothèque standard peuvent être modifiées à l'exécution

```
let s = string_of_bool false
val s : string = "false"
s.[0]<-'t'; s.[1]<-'r'; s.[2]<-'u'; s.[3]<-'e'; s.[4]<-' '
- : unit = ()
string_of_bool false
- : string = "true "
```

Exemples d'utilisation de chaînes de caractères littérales :

- `Sys.argv.(1)`
- `Char.escaped '\''`
- `Filename.parent_dir_name`
- `Filename.temp_dir_name`
- `Printexc.to_string Stack_overflow`
- `Sys.ocaml_version`

Désactivation du contrôle des bornes

Fonctions `unsafe_*`

Les fonctions `unsafe_*` et l'option `-unsafe` désactivent le contrôle des bornes des chaînes de caractères et des tableaux

```
print_string "Password? "; let p = read_line ()
Password? abcdef
val p : string = "abcdef"
let secret = Printf.sprintf "%s %8s" "password is" p
val secret : string = "password is abcdef"
Printf.printf "Secret: %s\n" secret
Secret: password is   abcdef
```

Le programme effectue une manipulation de données confidentielles

Désactivation du contrôle des bornes

Fonctions `unsafe_*`

Les fonctions `unsafe_*` et l'option `-unsafe` désactivent le contrôle des bornes des chaînes de caractères et des tableaux

```
let action () =
  let base = String.make 1 'c' in
  let buf = String.make 20 'x' in
  let ofs = ref 0 in
  begin try while true do
    String.unsafe_blit base !ofs buf 0 20;
    if buf.[0] = 'p' && String.sub buf 1 10 = "assword:"
    then raise Exit;
    incr ofs;
  done with Exit -> () end;
  Printf.eprintf "Secret vole: %s\n" (String.sub buf 12 8)
val action : unit -> unit
action ()
Secret vole:   abcdef
```

Un code malveillant peut récupérer ces données par accès non contrôlé à la mémoire

Langage
Conclusion

Contrôle de l'accès aux données et aux traitements

- ✚ Encapsulation vérifiée statiquement par typage
- ▬ Contournements du typage et de l'encapsulation :
  - ▬ Redéfinition de constructeur d'exceptions
  - ▬ Polymorphisme ad hoc de l'égalité, de la comparaison et du hachage
  - ▬ Désérialisation non typée
  - ▬ Module `Obj`
- ▬ Mutabilité des chaînes de caractères

Langage
Comparaison

Contrôle de l'accès aux données et aux traitements

- Encapsulation et typage

OCaml

- \+ Encapsulation par module, fermeture, objet
- − Pas de typage dynamique
- − Possibilités de contournement du typage et de l'encapsulation

F#

- \+ Encapsulation par module, fermeture, objet
- \+ Typage dynamique du bytecode .NET
- − Possibilité de contournement du typage par introspection

Scala

- \+ Encapsulation par objet
- − Pas de typage du bytecode Java
- − Possibilité de contournement du typage par introspection

Implémentation

Compilation – Bibliothèque standard – Exécution

OCaml 3.12.0

Assurer l'intégrité des données manipulées et du programme

- Protection des données gérées par le programme
- Protection du code du programme

Apports

- Séparation programme – données
- GC
- Bibliothèques de grands entiers

Inconvénients

- Exécution bytecode
- Absence d'effacement
- Désérialisation non typée
- Chargement dynamique non vérifié

Garbage Collector

Le GC est en charge de l'allocation et de la libération de mémoire

- Une portée trop grande d'un identifiant engendre une présence trop longue en mémoire
- Le moment de la libération est déterminé par le GC, elle peut être forcée par appel au GC
- Il n'existe pas d'effacement à la libération

Le GC recopie les valeurs en mémoire

- La recopie par le GC ne peut être contrôlée
- La recopie par les mécanismes système ne peut être contrôlée

Observation de l'exécution bytecode

Activation du mode débogue d'exécution

Le mode débogue comporte deux volets :

- Un mode de compilation bytecode (option `-g`) qui annote le bytecode avec des informations de localisation
- Un mode d'exécution avec `ocamlrun` dirigé par `ocamldebug`

Activation du mode débogue d'exécution

L'exécution en mode débogue est possible en bytecode quelles que soient les options de compilation (avec ou sans `-g`) en définissant la variable d'environnement `CAML_DEBUG_SOCKET`

Implémentation

Compilation – Bibliothèque standard – Exécution

Intégrité du programme et de ses données à l'exécution

- ✚ Typage et gestion automatique de la mémoire (pas d'arithmétique de pointeurs)
- ✚ Typage des fonctions de *string format* (Module `Printf`)
- ✚ Séparation programme – données
- ✚ Vérification de non–débordement
- ▬ Absence d'effacement par le GC
- ▬ Recopies par le GC
- ▬ Exécution de bytecode injecté
- ▬ Désérialisation non typée
- ▬ Chargement dynamique non vérifié
- ▬ Désactivation du contrôle des bornes
- ▬ Non contrôle du dépassement d'entiers

Implémentation
Compilation – Bibliothèque standard – Exécution

Protection des données et du code

OCaml

- \+ Gestion automatique de la mémoire
- \+ Mode natif (possibilité de profiter des protections mémoire OS NX)
- \+ Bibliothèque de grands entiers
- − Ni effacement ni verrouillage par le GC (possible avec du code C)
- − Débordement d'entiers standard

F#

- \+ Gestion automatique de la mémoire
- − Pas de mode natif
- − Ni effacement ni verrouillage
- − Débordement d'entiers

Scala

- \+ Gestion automatique de la mémoire
- − Pas de mode natif
- − Ni effacement ni verrouillage
- − Débordement d'entiers

Sécurité et langage de programmation

Apport du langage pour la sécurité

Niveau source

- Modules et interfaces
- Types concrets (construction de valeurs)
- Typage fort statique
- Encapsulation (module, objet, fermeture)
- Vérification du filtrage
- Vérifications étendues (option `-w`)

Niveau exécutable

- Contrôle des bornes
- Ramasse-miettes (GC)
- Séparation programme – données

Sécurité et langage de programmation

Apport du langage pour la sécurité

Vulnérabilités classiques évitées

- Dépassement de bornes
- *format string attack*
- Injection de code
- Déréférencement de pointeur nul
- Variables non initialisées
- Double libération mémoire
- Randomisation des tables de hachage (depuis OCaml 4.00)

Conclusion et perspectives

- De vraies propriétés de sécurité offertes par OCaml
- Devant être complétées et garanties pour rendre plus robuste l'implémentation
- Apports pour la traçabilité
- Apports des analyses de code
- Importance de considérer le langage et son modèle d'exécution
- Prendre en compte l'évaluation de sécurité dès le début et fournir une documentation appropriée

Liste des participants

Tranche ferme du projet LaFoSec

- François Armand
- Guillaume Burel
- Véronique Delebarre
- Damien Doligez
- Catherine Dubois
- Christèle Faure
- Thérèse Hardin
- Manuel Maarek
- Renaud Rioboo

Contacts

- Véronique Delebarre : veronique.delebarre@safe-river.com
- Christèle Faure : christele.faure@safe-river.com