



Mind your Language!

Langages de développement et sécurité

Éric JAEGER & Olivier LEVILLAIN

UPMC/M2 STL, vendredi 21 février 2014

An unreliable programming language generating unreliable programs constitutes a far greater 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.

C.A.R. Hoare



En 2005, un industriel interroge la DCSSI pour savoir si le langage JAVA peut être utilisé développer un produit de sécurité

Cette question, généralisée, a mené à différentes études dont

- ▶ JAVASEC : sécurité du langage JAVA
- ▶ LAFOSEC : sécurité des langages fonctionnels (dont OCAML)

L'une des leçons de ces études, c'est que les questions de l'ANSSI à propos des langages ne sont pas toujours partagées ou comprises

Cette présentation s'inscrit dans des travaux visant à illustrer et expliquer *a posteriori* cet intérêt pour la sécurité des langages, et mettre en évidence la question du choix du langage¹

1. Qui pourrait faire partie de la solution et non pas du problème...



Parmi les commandes suivantes, lesquelles sont susceptibles (sans redirection) de provoquer la destruction de données d'un fichier ?

- `ls` `cd` `cp` `cat` `rm` `mv`

Fonctionnel

- ▶ Question ré-interprétée « *Comment détruire les données d'un fichier ?* », seule la commande `rm` est mentionnée

Sécurité

- ▶ Si on cherche à protéger en disponibilité les données, les commandes dangereuses sont `rm` mais aussi `cp` et `mv`



Un paquet *IP* comporte dans son entête deux champs adresse pour sa source et sa destination

Fonctionnel

- ▶ Rendre le service consiste à acheminer l'information jusqu'à l'adresse destination
- ▶ La diffusion convient dans certains contextes

Sécurité

- ▶ L'adresse source n'est probablement ni exploitée ni vérifiée, ce qui permet le *spoofing* d'adresse *IP*
- ▶ En diffusion, un récepteur voit l'ensemble des paquets



Spécification de deux fonctions pour la compression (**Zip**) et la décompression (**Unzip**) de fichiers

Fonctionnel

- ▶ $\forall (f : \text{File}), \text{Unzip}(\text{Zip } f) = f$

Sécurité

- ▶ $\forall (c : \text{File}), (\neg \exists (f : \text{File}), \text{Zip } f = c) \Rightarrow \text{Unzip } c = \text{Error}$
- ▶ En particulier, ne pas avoir confiance en un champ annonçant à l'avance la taille du fichier décompressé



Quelques aspects intéressants d'un langage en termes de sécurité

- ▶ Faux amis ou pièges
- ▶ Constructions non spécifiées ou non définies mais utilisables
- ▶ Possibilités d'offuscation pour le développeur malicieux
- ▶ Bugs dans les outils de développement
- ▶ Comportements inappropriés des outils de développement
- ▶ Limitations des capacités d'analyse
- ▶ Surprises à l'exécution
- ▶ ...

Ce qui est signalé dans la suite n'est pas forcément une erreur, mais « attire l'attention » des experts en sécurité



Plan

- 1 **Toi qui entres ici. . .**
- 2 Au-delà des illustrations
- 3 Quelques éléments de conclusion



- 1 **Toi qui entres ici. . .**
 - Lutte des classes
 - Mise en boîte
 - Effets spéciaux
 - Épreuves d'adresses
 - *Cast-o-rama*
 - Rencontre du 3^e type
 - Où tout est question d'interprétation
 - Les liaisons dangereuses
 - Syntaxe, priez pour nous
 - Comme une odeur de *formal*



Le comportement de programmes objet très simples est parfois difficilement prévisible : que fait le code suivant² ?

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");
    }
}
```

2. Indice : `Mathf.pi=3.1415`



Le comportement de programmes objet très simples est parfois difficilement prévisible : que fait le code suivant ² ?

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");
    }
}
```

demo\$ java StaticInit

Bad things happen!

2. Indice : `Mathf.pi=3.1415`



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

Source (snippets/java/Mathf.java)

```
class Mathf {
    static double pi=3.1415;
    static { // Do whatever you want here
        System.out.println("Bad things happen!");
        // Do not return to calling class
        System.exit(0); }
}
```

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



Évitons donc toute référence à une classe externe non maîtrisée

Source (snippets/java/Deserial.java)

```
import java.io.*;
class Friend { } // Unlikely to be dangerous!
class Deserial {
    public static void main (String[] args)
        throws FileNotFoundException, IOException,
            ClassNotFoundException {
        FileInputStream fis = new FileInputStream("friend");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Friend f=(Friend)ois.readObject();
        System.out.println("Hello world");
    }
}
```



Évitons donc toute référence à une classe externe non maîtrisée

Source (snippets/java/Deserial.java)

```
import java.io.*;
class Friend { } // Unlikely to be dangerous!
class Deserial {
    public static void main (String[] args)
        throws FileNotFoundException, IOException,
            ClassNotFoundException {
        FileInputStream fis = new FileInputStream("friend");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Friend f=(Friend)ois.readObject();
        System.out.println("Hello world");
    }
}
```

Hélas à l'exécution on a **Bad things happen!** Le fichier sérialisé contient la classe de l'objet puis le *cast* échoue... mais trop tard



- 1 **Toi qui entres ici...**
 - Lutte des classes
 - **Mise en boîte**
 - Effets spéciaux
 - Épreuves d'adresses
 - *Cast-o-rama*
 - Rencontre du 3^e type
 - Où tout est question d'interprétation
 - Les liaisons dangereuses
 - Syntaxe, priez pour nous
 - Comme une odeur de *formal*



L'encapsulation objet est-elle un mécanisme de sécurité ?

Source (snippets/java/Introspect.java)

```
import java.lang.reflect.*;
class Secret { private int x = 42; }
public class Introspect {
    public static void main (String[] args) {
        try { Secret o = new Secret();
            Class c = o.getClass();
            Field f = c.getDeclaredField("x");
            f.setAccessible(true);
            System.out.println("x="+f.getInt(o));
        }
        catch (Exception e) { System.out.println(e); }
    }
}
```

L'introspection peut être interdite dans la politique de sécurité JAVA, mais c'est complexe et des effets secondaires sont probables



OCAML offre également différents mécanismes d'encapsulation³

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(); Random.int 8192  
end;;
```

La valeur `id` est visible et celle de `key` masquée

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

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

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



Mais cette encapsulation peut être contournée

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 de `key` est retournée ; impossible d'ouvrir la boîte, mais on peut la comparer à d'autres sur une balance



Pas convaincu ? Remettons en cause le typage, alors...

Source (snippets/ocaml/hsm5.ml)

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

module C : Crypto =
struct
  let id=42
  let secretchar='k'
end;;

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);;
```

L'oracle retourne 107, le code ASCII du caractère 'k'



JAVA permet de définir des classes internes

Source (snippets/java/Innerclass.java)

```
public class Innerclass {
    private static int a=42;

    static public class Innerinner {
        private static int b=54;
        public static void print() {
            System.out.println(Innerclass.a);
        }
    }

    public static void main (String[] args) {
        System.out.println(Innerinner.b);
        Innerinner.print();
    }
}
```

Pourtant les classes internes ne sont pas supportées en *bytecode* !



- 1 **Toi qui entres ici...**
 - Lutte des classes
 - Mise en boîte
 - **Effets spéciaux**
 - Épreuves d'adresses
 - *Cast-o-rama*
 - Rencontre du 3^e type
 - Où tout est question d'interprétation
 - Les liaisons dangereuses
 - Syntaxe, priez pour nous
 - Comme une odeur de *formal*



[C] Questions stratégiques

Si on joue avec les effets de bord, la stratégie d'évaluation devient significative, et des subtilités peuvent apparaître

Source (snippets/c/effect.c)

```
{ int c=0; printf("%d %d\n",c++,c++); }
```

```
{ int c=0; printf("%d %d\n",++c,++c); }
```

```
{ int c=0; printf("%d %d\n",c=1,c=2); }
```



Si on joue avec les effets de bord, la stratégie d'évaluation devient significative, et des subtilités peuvent apparaître

Source (snippets/c/effect.c)

```
{ int c=0; printf("%d %d\n",c++,c++); }  
  
{ int c=0; printf("%d %d\n",++c,++c); }  
  
{ int c=0; printf("%d %d\n",c=1,c=2); }
```

La première ligne affiche 1 0



[C] Questions stratégiques

Si on joue avec les effets de bord, la stratégie d'évaluation devient significative, et des subtilités peuvent apparaître

Source (snippets/c/effect.c)

```
{ int c=0; printf("%d %d\n",c++,c++); }
```

```
{ int c=0; printf("%d %d\n",++c,++c); }
```

```
{ int c=0; printf("%d %d\n",c=1,c=2); }
```

La première ligne affiche **1 0**

La seconde ligne affiche **2 2**



[C] Questions stratégiques

Si on joue avec les effets de bord, la stratégie d'évaluation devient significative, et des subtilités peuvent apparaître

Source (snippets/c/effect.c)

```
{ int c=0; printf("%d %d\n",c++,c++); }
```

```
{ int c=0; printf("%d %d\n",++c,++c); }
```

```
{ int c=0; printf("%d %d\n",c=1,c=2); }
```

La première ligne affiche **1 0**

La seconde ligne affiche **2 2**

La troisième ligne affiche **1 1**



En OCAML le code est statique et les chaînes sont mutables ; mais qu'en est-il des chaînes apparaissant dans le code ?

Source (snippets/ocaml/mutable.ml)

```
let check c =  
  if c then "Tout va bien" else "Tout va mal!";;  
  
let f=check false in  
  f.[8]<- 'b'; f.[9]<- 'i'; f.[10]<- 'e'; f.[11]<- 'n';;  
  
check true;;  
check false;;
```



En OCAML le code est statique et les chaînes sont mutables ; mais qu'en est-il des chaînes apparaissant dans le code ?

Source (snippets/ocaml/mutable.ml)

```
let check c =  
  if c then "Tout va bien" else "Tout va mal!";;  
  
let f=check false in  
  f.[8]<- 'b'; f.[9]<- 'i'; f.[10]<- 'e'; f.[11]<- 'n';;  
  
check true;;  
check false;;
```

Les deux applications de `check` renvoient "Tout va bien"



L'exemple précédent n'est pas une redéfinition de la fonction `alert` mais un simple effet de bord ; pour s'en convaincre, voici ce que cela donne avec une fonction de la bibliothèque standard

Source (snippets/ocaml/mutablebool.ml)

```
let t=string_of_bool true in
  t.[0]<- 'f'; t.[1]<- 'a'; t.[3]<- 'x';;

Printf.printf "1=1 est %b\n" (1=1);;
```

Le code affiche `1=1 est faux` ; d'autres fonctions intéressantes sont concernées, par exemple `Char.escaped`⁴ ainsi que certains *patterns* de développement usuels basés sur les exceptions

4. Rappelons que c'est une fonction de sécurité...



Au préalable, deux petits rappels

Source (snippets/c/strategy1.c)

```
#define abs(X) (X)>=0?(X):(-X)
int abs(int x) { return x>=0?x:-x; }

#define first(x,y) x
int first(int x,int y) { return x; }
```



Au préalable, deux petits rappels

Source (snippets/c/strategy1.c)

```
#define abs(X) (X)>=0?(X):(-X)
int abs(int x) { return x>=0?x:-x; }

#define first(x,y) x
int first(int x,int y) { return x; }
```

Les deux versions de `abs` n'ont pas le même comportement par exemple pour `abs(x++)`



Au préalable, deux petits rappels

Source (snippets/c/strategy1.c)

```
#define abs(X) (X)>=0?(X):(-X)
int abs(int x) { return x>=0?x:-x; }

#define first(x,y) x
int first(int x,int y) { return x; }
```

Les deux versions de `abs` n'ont pas le même comportement par exemple pour `abs(x++)`

Les deux versions de `first` n'ont pas le même comportement par exemple pour `first(x,1/x)` avec `x` valant `0` – les exceptions sont également des effets de bord



Ces subtilités étant clarifiées, que font les deux codes suivants ?

Source (snippets/c/strategy3.c)

```
#include <stdio.h>
int zero(int x) { return 0; }
int main(void) { int x=0; x=zero(1/x); return 0; }
```

Source (snippets/c/strategy3b.c)

```
#include <stdio.h>
int zero(int x) { return 0; }
int main(void) { int x=0; return zero(1/x); }
```



Ces subtilités étant clarifiées, que font les deux codes suivants ?

Source (snippets/c/strategy3.c)

```
#include <stdio.h>
int zero(int x) { return 0; }
int main(void) { int x=0; x=zero(1/x); return 0; }
```

Source (snippets/c/strategy3b.c)

```
#include <stdio.h>
int zero(int x) { return 0; }
int main(void) { int x=0; return zero(1/x); }
```

Tout dépend du niveau d'optimisation :

- ▶ -O0 : Floating point exception (core dumped)
- ▶ -O1 : le premier programme termine normalement
- ▶ -O2 : les deux programmes terminent normalement



- 1 **Toi qui entres ici...**
 - Lutte des classes
 - Mise en boîte
 - Effets spéciaux
 - **Épreuves d'adresses**
 - *Cast-o-rama*
 - Rencontre du 3^e type
 - Où tout est question d'interprétation
 - Les liaisons dangereuses
 - Syntaxe, priez pour nous
 - Comme une odeur de *formal*



Petit rappel sur le principe des *Buffer Overflows*⁵

Source (snippets/c/overflow.c)

```
#include <stdio.h>
#include <stdlib.h>

void set(int s,int v) { *(&s-s)=v; }

void bad() { printf("Bad things happen!\n"); exit(0); }

int main(void) {
    set(1,(int)bad); printf("Hello world\n"); return 0;
}
```

5. Ici dans une version balèze, car sans *Buffer*



Petit rappel sur le principe des *Buffer Overflows*⁵

Source (snippets/c/overflow.c)

```
#include <stdio.h>
#include <stdlib.h>

void set(int s,int v) { *(&s-s)=v; }

void bad() { printf("Bad things happen!\n"); exit(0); }

int main(void) {
    set(1,(int)bad); printf("Hello world\n"); return 0;
}
```

Bad things happen! La pile est corrompue, on peut surcharger des variables, modifier une adresse de retour voire injecter du code

5. Ici dans une version balèze, car sans *Buffer*



[C] Faire mauvaise impression

Mais les manipulations de piles sont parfois bien cachées. . .

Source (snippets/c/stringformat3.c)

```
#include <stdio.h>

char *f="%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.\
%08x.%08x.%08x.%08x.%08x.%08x.%n";

void strfmtattack() { printf(f); }

int main(void) {
    int s=0x12345;
    int *p=&s;
    strfmtattack();
    if (s!=0x12345) printf("Bad things happen! s=%08x\n",s);
    return 0;
}
```



[C] Faire mauvaise impression

Mais les manipulations de piles sont parfois bien cachées. . .

Source (snippets/c/stringformat3.c)

```
#include <stdio.h>

char *f="%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.\
%08x.%08x.%08x.%08x.%08x.%08x.%n";

void strfmtattack() { printf(f); }

int main(void) {
    int s=0x12345;
    int *p=&s;
    strfmtattack();
    if (s!=0x12345) printf("Bad things happen! s=%08x\n",s);
    return 0;
}
```

...0036b225.00fdd280.00000000.00012345.Bad things happen! s=0000007e



1 Toi qui entres ici. . .

- Lutte des classes
- Mise en boîte
- Effets spéciaux
- Épreuves d'adresses
- **Cast-o-rama**
- Rencontre du 3^e type
- Où tout est question d'interprétation
- Les liaisons dangereuses
- Syntaxe, priez pour nous
- Comme une odeur de *formal*



ERLANG propose surcharges et conversions automatiques (*cast*) :
`1+1`, `1.0+1.0` et `1+1.0` sont des expressions valides⁶

Source (snippets/erlang/factorial.erl)

```
-module(factorial).  
-compile(export_all).  
  
fact(0) -> 1;  
fact(N) -> N*fact(N-1).
```

6. Et il serait inutile de s'embêter à distinguer entiers et flottants...



ERLANG propose surcharges et conversions automatiques (*cast*) :
`1+1`, `1.0+1.0` et `1+1.0` sont des expressions valides⁶

Source (snippets/erlang/factorial.erl)

```
-module(factorial).  
-compile(export_all).  
  
fact(0) -> 1;  
fact(N) -> N*fact(N-1).
```

```
demo$ erl
```

```
1> factorial:fact(4).
```

```
24
```

```
2> factorial:fact(4.0).
```

```
eheap_alloc: Cannot allocate...
```

6. Et il serait inutile de s'embêter à distinguer entiers et flottants...



JAVASCRIPT offre également tout le confort moderne...

Source (snippets/js/unification2.js)

```
if (0=='0') print("Equal"); else print("Different");

switch (0)
{ case '0':print("Equal");
  default:print("Different");
}
```



JAVASCRIPT offre également tout le confort moderne...

Source (snippets/js/unification2.js)

```
if (0=='0') print("Equal"); else print("Different");

switch (0)
{ case '0':print("Equal");
  default:print("Different");
}
```

L'affichage obtenu est `Equal` puis `Different`



Faut-il préférer *cast* et surcharge, ou associativité et transitivité ?

Source (snippets/js/cast2.js)

```
if ('0'==0) print("'0'==0");
else print("'0'<>0");
if (0=='0.0') print("0=='0.0'");
else print("0<>'0.0'");
if ('0'=='0.0') print("'0'=='0.0'");
else print("'0'<>'0.0'");
```



Faut-il préférer *cast* et surcharge, ou associativité et transitivité ?

Source (snippets/js/cast2.js)

```
if ('0'==0) print("'0'==0");  
else print("'0'<>0");  
if (0=='0.0') print("0=='0.0'");  
else print("0<>'0.0'");  
if ('0'=='0.0') print("'0'=='0.0'");  
else print("'0'<>'0.0'");
```

'0'==0, 0=='0.0' et '0'<>'0.0'



Faut-il préférer *cast* et surcharge, ou associativité et transitivité ?

Source (snippets/js/cast2.js)

```
if ('0'==0) print("'0'==0");  
else print("'0'<>0");  
if (0=='0.0') print("0=='0.0'");  
else print("0<>'0.0'");  
if ('0'=='0.0') print("'0'=='0.0'");  
else print("'0'<>'0.0'");
```

'0'==0, 0=='0.0' et '0'<>'0.0'

Source (snippets/js/cast3.js)

```
a=1; b=2; c='Foo';  
print(a+b+c); print(c+a+b); print(c+(a+b));
```



Faut-il préférer *cast* et surcharge, ou associativité et transitivité ?

Source (snippets/js/cast2.js)

```
if ('0'==0) print("'0'==0");  
else print("'0'<>0");  
if (0=='0.0') print("0=='0.0'");  
else print("0<>'0.0'");  
if ('0'=='0.0') print("'0'=='0.0'");  
else print("'0'<>'0.0'");
```

'0'==0, 0=='0.0' et '0'<>'0.0'

Source (snippets/js/cast3.js)

```
a=1; b=2; c='Foo';  
print(a+b+c); print(c+a+b); print(c+(a+b));
```

3Foo, Foo12 et Foo3



Le compilateur C cherche également à rendre service, que vaut z ?

Source (snippets/c/cast0.c)

```
{ int x=3; int y=4; float z=x/y; }
```



Le compilateur C cherche également à rendre service, que vaut z ?

Source (snippets/c/cast0.c)

```
{ int x=3; int y=4; float z=x/y; }
```

0.0



Le compilateur C cherche également à rendre service, que vaut z ?

Source (snippets/c/cast0.c)

```
{ int x=3; int y=4; float z=x/y; }
```

0.0

Source (snippets/c/cast1.c)

```
{ unsigned char x = 128; unsigned char y = 2;  
  unsigned char z = (x * y) / y; }
```



Le compilateur C cherche également à rendre service, que vaut z ?

Source (snippets/c/cast0.c)

```
{ int x=3; int y=4; float z=x/y; }
```

0.0

Source (snippets/c/cast1.c)

```
{ unsigned char x = 128; unsigned char y = 2;  
  unsigned char z = (x * y) / y; }
```

128 (ce n'est pas une optimisation du compilateur mais un *cast*)



Pour les curieux, voici le code assembleur associé à `cast1.c`

Source (snippets/c/castregister.asm)

```
unsigned char x = 128;
    movb    $-128, -19(%rbp)
unsigned char y = 2;
    movb    $2, -18(%rbp)
unsigned char z = (x * y) / y;
    movzbl  -19(%rbp), %edx
    movzbl  -18(%rbp), %eax
    imull  %edx, %eax
    movzbl  -18(%rbp), %edx
```



Le développeur peut parfois mettre son propre grain de sel

Source (snippets/java/Confuser.java)

```
class Confuser {  
  
    static void A(short i) { System.out.println("Foo"); }  
    static void A(int i) { System.out.println("Bar"); }  
  
    public static void main (String[] args) {  
        short i=0;  
        A(i);  
        A(i+i);  
        A(i+=i);  
    }  
}
```



Le développeur peut parfois mettre son propre grain de sel

Source (snippets/java/Confuser.java)

```
class Confuser {  
  
    static void A(short i) { System.out.println("Foo"); }  
    static void A(int i) { System.out.println("Bar"); }  
  
    public static void main (String[] args) {  
        short i=0;  
        A(i);  
        A(i+i);  
        A(i+=i);  
    }  
}
```

Le programme affiche **Foo**, **Bar**, **Foo** ; dans la « vraie » vie ajoutez de l'héritage et des `Integer`



Avant de présenter l'exemple suivant, un petit rappel : combien y a-t-il de solutions pour l'équation $x = -x$ en \mathbb{C} ?

Source (snippets/c/cast2.c)

```
#include <stdio.h>

int main(void) {
    int z=0;
    if (z== -z) printf ("%d==-(%d)\n", z, z);

    int r=1<<((sizeof(int)*8)-1);
    if (r== -r) printf ("%d==-(%d)\n", r, r);

    return 0;
}
```

$0 == -(0)$ et $-2147483648 == -(-2147483648)$, c'est ce qui fait tout le charme du complément à 2 pour représenter les entiers signés



[C] C'est mauvais signe...

Même quand on pense avoir bien compris, il reste des surprises...

Source (snippets/c/castsigned2.c)

```
{ unsigned char a = 1; signed char b = -1;
  if (a<b) printf("%d<%d\n",a,b);
  else printf("%d>=%d\n",a,b); }

{ unsigned int a = 1; signed int b = -1;
  if (a<b) printf("%d<%d\n",a,b);
  else printf("%d>=%d\n",a,b); }
```



[C] C'est mauvais signe...

Même quand on pense avoir bien compris, il reste des surprises...

Source (snippets/c/castsigned2.c)

```
{ unsigned char a = 1; signed char b = -1;
  if (a<b) printf("%d<%d\n",a,b);
  else printf("%d>=%d\n",a,b); }

{ unsigned int a = 1; signed int b = -1;
  if (a<b) printf("%d<%d\n",a,b);
  else printf("%d>=%d\n",a,b); }
```

Avec les types `char`, on obtient `1>=-1`, alors qu'avec les types `int` on obtient `1<-1` – encore des subtilités sur les promotions



Ces mécanismes peuvent rendre certains contrôles caduques

Source (snippets/c/castbound.c)

```
#include <stdio.h>

void write(int tab[],int size,signed char ind,int val) {
    if (ind<size) tab[ind]=val;
    else printf("Fail\n");
}

int main(void) {
    size_t size=120; int tab[size];
    write(tab,size,127,42);
    write(tab,size,128,42);
    return 0;
}
```



Ces mécanismes peuvent rendre certains contrôles caduques

Source (snippets/c/castbound.c)

```
#include <stdio.h>

void write(int tab[],int size,signed char ind,int val) {
    if (ind<size) tab[ind]=val;
    else printf("Fail\n");
}

int main(void) {
    size_t size=120; int tab[size];
    write(tab,size,127,42);
    write(tab,size,128,42);
    return 0;
}
```

Première écriture refusée (**Fail**) mais seconde acceptée – le où est laissé à la sagacité de l'auditoire... Et si `size=150` ?



[C] Une vraie vulnérabilité

Si vous le lui demandez, le compilateur peut parfois vous aider à repérer ces problèmes (`-Wconversion`⁷ pour GCC)

Les plus curieux d'entre vous pourront étudier la faille CVE-2010-0740 (*Record of death vulnerability*) sur OpenSSL

Source (snippets/c/patch-CVE-2010-0740.diff)

```
- /* Send back error using their
-  * version number :-) */
- s->version=version;
+ if ((s->version & 0xFF00) == (version & 0xFF00))
+ /* Send back error using their minor version number :-)
+  */
+ s->version = (unsigned short)version;
```

Il semble qu'une faille d'authentification sur MySQL puisse également être liée à ce type de difficultés

7. À moins que ce ne soit `-Wextra`? Ce n'est pas très clair...



Un autre détail : qu'affiche le code suivant ?

Source (snippets/c/3264.c)

```
unsigned long setBit1(int n)
{ return (1 << n); }

unsigned long setBit2(int n)
{ return ((unsigned long) 1 << n); }

void main () {
    printf ("%lu\n", setBit1 (33));
    printf ("%lu\n", setBit2 (33));
}
```



Un autre détail : qu'affiche le code suivant ?

Source (snippets/c/3264.c)

```
unsigned long setBit1(int n)
{ return (1 << n); }

unsigned long setBit2(int n)
{ return ((unsigned long) 1 << n); }

void main () {
    printf ("%lu\n", setBit1 (33));
    printf ("%lu\n", setBit2 (33));
}
```

Réponse : ça dépend de l'architecture retenue pour la compilation

- ▶ En 32 bits, les deux lignes affichent **2**
- ▶ En 64 bits, la première affiche **2** et la seconde **8589934592** (2^{33})



Celle-là est quand même gratinée⁸

Source (snippets/php/castincr.php)

```
$x="2d8";  
print("\$x+1 = "); $x=$x+1; print("$x.\n");  
  
$x="2d8";  
print("\$x++ = ".$x++."\n"); print("\$x++ = ".$x++."\n");  
print("\$x++ = ".$x++."\n"); print("\$x++ = ".$x++."\n");
```

8. Même JAVASCRIPT n'ose pas aller jusque là, c'est dire



Celle-là est quand même gratinée⁸

Source (snippets/php/castincr.php)

```
$x="2d8";  
print("\$x+1 = "); $x=$x+1; print("$x.\n");  
  
$x="2d8";  
print("\$x++ = ".$x++."\n"); print("\$x++ = ".$x++."\n");  
print("\$x++ = ".$x++."\n"); print("\$x++ = ".$x++."\n");
```

La première ligne affiche `$x+1 = 3` – *a priori*, il prend le préfixe entier de la chaîne et ajoute 1

8. Même JAVASCRIPT n'ose pas aller jusque là, c'est dire



Celle-là est quand même gratinée⁸

Source (snippets/php/castincr.php)

```
$x="2d8";  
print("\$x+1 = "); $x=$x+1; print("$x.\n");  
  
$x="2d8";  
print("\$x++ = ".$x++."\n"); print("\$x++ = ".$x++."\n");  
print("\$x++ = ".$x++."\n"); print("\$x++ = ".$x++."\n");
```

La première ligne affiche `$x+1 = 3` – *a priori*, il prend le préfixe entier de la chaîne et ajoute 1

Mais le `++` a une sémantique différente, puisqu'on obtient ensuite `$x++ = 2d8`, puis `$x++ = 2d9`, `$x++ = 2e0`... et enfin `$x++ = 3`

8. Même JAVASCRIPT n'ose pas aller jusque là, c'est dire



Surcharges et *casts* permettent au compilateur de tripatouiller le code jusqu'à lui trouver *un*⁹ sens

Source (snippets/js/weirdeval.js)

```
{ } + { }  
[ ] + { }  
{ } + [ ]  
( { } + { } )
```

Toutes ces expressions ont un sens différent de celui des autres (même la première et la quatrième ligne)

jscert.org donne d'autres éléments surprenants sur JAVASCRIPT

9. Article volontairement indéfini



1 Toi qui entres ici. . .

- Lutte des classes
- Mise en boîte
- Effets spéciaux
- Épreuves d'adresses
- *Cast-o-rama*
- **Rencontre du 3^e type**
- Où tout est question d'interprétation
- Les liaisons dangereuses
- Syntaxe, priez pour nous
- Comme une odeur de *formal*



[OCAML] Certains sont plus égaux que d'autres

En OCAML, x et y étant des valeurs, combien de valeurs peut prendre l'expression booléenne $x=y$?



En OCAML, `x` et `y` étant des valeurs, combien de valeurs peut prendre l'expression booléenne `x=y` ?

Source (snippets/ocaml/bool4.ml)

```
type tree= Leaf | Node of tree*tree;;

let zero = Leaf;;
let one = Node(Leaf,Leaf);;
let rec many = Node(Leaf,many);;
let rec more = Node(more,Leaf);;
```

`zero=zero` donne `true`, `zero=one` donne `false`, `many=many` boucle et `more=more` provoque une erreur – donc 4 comportements



Pas convaincu par la remarque précédente ? Et pourtant. . .

Source (snippets/shell/login.sh)

```
#!/bin/bash
PIN=1234
echo -n "Veuillez saisir le code PIN (4 chiffres): "
read -s PIN_SAISI; echo

if [ "$PIN" -ne "$PIN_SAISI" ]; then
    echo "Code PIN invalide."; exit 1
else
    echo "Authentication OK"; exit 0
fi
```



[Shell] Interdiction d'interdire

Pas convaincu par la remarque précédente ? Et pourtant. . .

Source (snippets/shell/login.sh)

```
#!/bin/bash
PIN=1234
echo -n "Veuillez saisir le code PIN (4 chiffres): "
read -s PIN_SAISI; echo

if [ "$PIN" -ne "$PIN_SAISI" ]; then
    echo "Code PIN invalide."; exit 1
else
    echo "Authentication OK"; exit 0
fi
```

Un mauvais code PIN sera rejeté ; par contre, si l'utilisateur saisit des caractères non numériques, l'accès lui sera accordé !



Les comportements à la marge des types sont souvent intéressants

Source (snippets/sql/partial.sql)

```
SELECT CONCAT(IF(@X<=@Y, 'X<=Y', 'X>Y'),  
              ' and ',  
              IF(@X>=@Y, 'X>=Y', 'X<Y')) AS Test;
```

Avec `SET @X=1; SET @Y=2;` on obtient `X<=Y and X<Y`



Les comportements à la marge des types sont souvent intéressants

Source (snippets/sql/partial.sql)

```
SELECT CONCAT(IF(@X<=@Y, 'X<=Y', 'X>Y'),  
              ' and ',  
              IF(@X>=@Y, 'X>=Y', 'X<Y')) AS Test;
```

Avec `SET @X=1; SET @Y=2;` on obtient `X<=Y and X<Y`

Avec `SET @X=NULL` par contre on obtient `X>Y and X<Y` (le même type d'observations peut être fait avec les flottants et `NaN`)



Types à l'exécution

Des types statiques constituent une information purement logique qui n'a aucune existence concrète dans l'implémentation

Il faut donc éviter des raisonnements étrequés excluant des situations « mal typées » mais néanmoins possibles

- ▶ Désérialisation de valeurs incohérentes en OCAML
- ▶ Attaque *Wrap & Decipher* sur l'API PKCS#11
- ▶ ...



- 1 **Toi qui entres ici...**
 - Lutte des classes
 - Mise en boîte
 - Effets spéciaux
 - Épreuves d'adresses
 - *Cast-o-rama*
 - Rencontre du 3^e type
 - **Où tout est question d'interprétation**
 - Les liaisons dangereuses
 - Syntaxe, priez pour nous
 - Comme une odeur de *formal*



On peut en PHP faire appel à un interpréteur SQL

Source (snippets/php/injectionsql.php)

```
$dbc=mysqli_connect(HST,LOG,PWD,"School");  
$cmd="SELECT * FROM Students WHERE id='".$val."'";  
$dbr=mysqli_query($dbc,$cmd);
```



On peut en PHP faire appel à un interpréteur SQL

Source (snippets/php/injectionsql.php)

```
$dbc=mysqli_connect(HST,LOG,PWD,"School");  
$cmd="SELECT * FROM Students WHERE id='".$val."'";  
$dbr=mysqli_query($dbc,$cmd);
```

Bien entendu, si `$val="Bobby'; DROP TABLE Students; //"`...



L'injection résulte donc de l'utilisation d'un interpréteur, en général celui d'un autre langage

Source (snippets/ocaml/syscommand.ml)

```
let printfile filename =  
  Sys.command("cat "^filename);;
```

10. L'ANSSI décline toute responsabilité pouvant découler d'essais



L'injection résulte donc de l'utilisation d'un interpréteur, en général celui d'un autre langage

Source (snippets/ocaml/syscommand.ml)

```
let printfile filename =  
  Sys.command("cat "^filename);;
```

`printfile "texput.log"` aura le résultat escompté
`printfile "--version ; cd / ; rm -ri ."` sans doute pas¹⁰

10. L'ANSSI décline toute responsabilité pouvant découler d'essais



Certains langages interprétés proposent un évaluateur interne

Source (snippets/php/injectioneval.php)

```
$cmd1="echo 'Hello ".$val."\n';";  
  
eval($cmd1);
```

Avec `$val` valant `''; die('eval killed me'); //` par exemple on dévie du comportement imaginé

C'est bien sur dangereux, mais surtout impossible à analyser



[PHP] C'est de la dynamique !

D'autres constructions, sans être des évaluateurs, sont également inquiétantes (ne parlons même pas de lisibilité ou de traçabilité)

Source (snippets/php/injectionvar.php)

```
function hello() { echo "Hello world<br />"; }  
function goodbye() { echo "Good bye<br />"; }  
  
$x="hello"; $x();  
  
$y="x"; $$y="goodbye"; $x();
```



[PHP] C'est de la dynamique !

D'autres constructions, sans être des évaluateurs, sont également inquiétantes (ne parlons même pas de lisibilité ou de traçabilité)

Source (snippets/php/injectionvar.php)

```
function hello() { echo "Hello world<br />"; }  
function goodbye() { echo "Good bye<br />"; }  
  
$x="hello"; $x();  
  
$y="x"; $$y="goodbye"; $x();
```

Hello world

Good bye



Une fois le concept d'injection bien compris, on en arrive à se poser des questions curieuses par exemple sur le *Shell*...

On peut utiliser `*` en paramètre dans une ligne de commande ; *a priori* c'est simple, pourtant il peut y avoir des questions assez subtiles, par exemple que se passe-t-il s'il existe un fichier nommé `"*`, `"-o"`, `"> rights.acl"` ou encore `"; rm *` ?



Une fois le concept d'injection bien compris, on en arrive à se poser des questions curieuses par exemple sur le *Shell*...

On peut utiliser `*` en paramètre dans une ligne de commande ; *a priori* c'est simple, pourtant il peut y avoir des questions assez subtiles, par exemple que se passe-t-il s'il existe un fichier nommé `"*`, `"-o"`, `"> rights.acl"` ou encore `"; rm *` ?

Rien... si ce n'est qu'un fichier dont le nom commence par `"-` sera généralement interprété *par la commande appelée* comme une option



Une fois le concept d'injection bien compris, on en arrive à se poser des questions curieuses par exemple sur le *Shell*...

On peut utiliser `*` en paramètre dans une ligne de commande ; *a priori* c'est simple, pourtant il peut y avoir des questions assez subtiles, par exemple que se passe-t-il s'il existe un fichier nommé `"*", "-o", "> rights.acl"` ou encore `"; rm *"` ?

Rien... si ce n'est qu'un fichier dont le nom commence par `"-` sera généralement interprété *par la commande appelée* comme une option

Un autre détail : si vous exécutez `cat foo*` dans un répertoire ne contenant aucun fichier de la forme `foo*`, vous savez ce que le *Shell* passe en paramètre à la commande `cat` ?



- 1 **Toi qui entres ici. . .**
 - Lutte des classes
 - Mise en boîte
 - Effets spéciaux
 - Épreuves d'adresses
 - *Cast-o-rama*
 - Rencontre du 3^e type
 - Où tout est question d'interprétation
 - **Les liaisons dangereuses**
 - Syntaxe, priez pour nous
 - Comme une odeur de *formal*



PYTHON offre une construction syntaxique proche du `map` classique sur les listes, la définition de liste en compréhension

Source (snippets/python/listcomp.py)

```
>>> l = [s+1 for s in [1,2,3]]
>>> l
[2, 3, 4]
```

Que se passe-t-il si ensuite on tape `s` à l'invite ?



PYTHON offre une construction syntaxique proche du `map` classique sur les listes, la définition de liste en compréhension

Source (snippets/python/listcomp.py)

```
>>> l = [s+1 for s in [1,2,3]]
>>> l
[2, 3, 4]
```

Que se passe-t-il si ensuite on tape `s` à l'invite ?

À moins d'utiliser la dernière version de Python 3, `s` vaut `3`, alors que la variable `s` devrait être locale (liée).



Un autre comportement intrigant

Source (snippets/python/localvar3.py)

```
source = [1,2,3,4]
dest = [0] * 4

def copy1 ():
    dest = source

def copy2 ():
    for i in range(4):
        dest[i]=source[i]
```

Initialement `dest` vaut `[0,0,0,0]`, après l'appel de `copy1()` il vaut toujours `[0,0,0,0]`, et après l'appel de `copy2()` il vaut `[1,2,3,4]`



En fait scripts et localité ne semblent pas faire bon ménage

Source (snippets/php/link.php)

```
$var = "var";  
$tab = array("foo ", "bar ", "blah ");  
  
echo "Tab="; echo $tab; echo " ; ";  
echo "Loop="; { foreach ($tab as $var) { echo $var; } }  
echo " ; ";  
  
echo "Var="; echo $var;
```



En fait scripts et localité ne semblent pas faire bon ménage

Source (snippets/php/link.php)

```
$var = "var";  
$tab = array("foo ", "bar ", "blah ");  
  
echo "Tab="; echo $tab; echo " ; ";  
echo "Loop="; { foreach ($tab as $var) { echo $var; } }  
echo " ; ";  
  
echo "Var="; echo $var;
```

On obtient `Tab=Array ; Loop=foo bar blah ; Var=blah`, la variable `var` est écrasée et survit à la boucle ! Les plus audacieux pourront tester `foreach ($tab as $tab)...`



- 1 **Toi qui entres ici. . .**
 - Lutte des classes
 - Mise en boîte
 - Effets spéciaux
 - Épreuves d'adresses
 - *Cast-o-rama*
 - Rencontre du 3^e type
 - Où tout est question d'interprétation
 - Les liaisons dangereuses
 - **Syntaxe, priez pour nous**
 - Comme une odeur de *formal*



Une modification du noyau LINUX¹¹

Source (snippets/c/kernel.diff)

```
+ if ((options==( __WCLONE|__WALL)) && (current->uid=0))  
+  retval = -EINVAL;
```

11. Cf. lwn.net/Articles/57135/



Une modification du noyau LINUX¹¹

Source (snippets/c/kernel.diff)

```
+ if ((options==( __WCLONE|__WALL)) && (current->uid=0))  
+  retval = -EINVAL;
```

Piégeage pur et simple : lorsque la condition sur `options` est vraie, `current->uid` devient 0 (*i.e.* le process passe `root`)

L'attaquant joue sur la confusion entre `=` et `==`, mais aussi le fait que l'affectation renvoie une valeur, que le typage ne distingue pas un booléen d'un entier, *etc.*

11. Cf. lwn.net/Articles/57135/



Petite caractéristique intrigante des commentaires OCAML

Source (snippets/ocaml/comments.ml)

```
(* blah blah " blah blah *)  
let x=true;;  
  
(* PREVIOUS VERSION -----  
(* blah blah " blah blah *)  
let x=false;;  
(* blah blah " blah blah *)  
-----*)  
  
(* blah blah " blah blah *)
```



Petite caractéristique intrigante des commentaires OCAML

Source (snippets/ocaml/comments.ml)

```
(* blah blah " blah blah *)
let x=true;;

(* PREVIOUS VERSION -----
(* blah blah " blah blah *)
let x=false;;
(* blah blah " blah blah *)
-----*)

(* blah blah " blah blah *)
```

Il est possible d'ouvrir une chaîne de caractères dans un commentaire OCAML, ce qui peut induire en erreur un relecteur, surtout si la coloration syntaxique n'est pas conforme



[C] Long discours ou petit dessin ?

Les commentaires visuels ne sont pas toujours les plus utiles

Source (snippets/c/comments2.c)

```
#include <stdio.h>

int main(void) {

// /\ DO NOT REMOVE COMMENTS IN NEXT BLOCK /\
/*****
    const char status []="Safe";
// /\ SET TO SAFE ONLY FOR TESTS /\
*****/

// /\ NEXT LINE REALLY IMPORTANT /\
    const char status []="Unsafe";
    printf("Status: %s\n",status);
}
```

Status: Safe (à moins qu'il n'y ait des espaces ici ou là...)



Les commentaires sont parfois vraiment délicats à traiter

Source (snippets/c/comments.c)

```
#include <stdio.h>

int foo() {
    int a=4; int b=2;
    return a /**
                /**/ b
;
}

int main(void) {
    printf("%d\n",foo()); return 0;
}
```

Ce code, compilé et exécuté, affiche **4** ; mais si on compile en mode C89 (option `-std=c89` de GCC) on obtient **2**



Certains compilateurs sont compatibles avec l'encodage UTF-8

Source (snippets/java/Preprocess.java)

```
public class Preprocess {
    public static void main (String[] args) {
        if (false==true)
        { //\u000a\u007d\u007b
            System.out.println("Bad things happen!");
        }
    }
}
```

Bad thing happens : le code source est donc *a priori* préprocessé préalablement à la compilation



La norme ANSI pour le C permet l'utilisation de claviers auxquels il manque des caractères utiles

La norme définit donc des *trigraphes*

- ▶ `??=` devient `#`
- ▶ `??/` devient `\`
- ▶ `??'` devient `^`
- ▶ `??(` devient `[`
- ▶ `??<` devient `{`
- ▶ `??!` devient `|`
- ▶ `??)` devient `]`
- ▶ `??>` devient `}`
- ▶ `??-` devient `~`

Donc si vous voyez un commentaire de la forme `// Blah ??/` vous saurez à quoi (ne pas) vous en tenir...



- 1 **Toi qui entres ici. . .**
 - Lutte des classes
 - Mise en boîte
 - Effets spéciaux
 - Épreuves d'adresses
 - *Cast-o-rama*
 - Rencontre du 3^e type
 - Où tout est question d'interprétation
 - Les liaisons dangereuses
 - Syntaxe, priez pour nous
 - Comme une odeur de *formal*



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
```



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 : l'invariant doit être vrai avant et après une opération, mais pas forcément **pendant**



Source (snippets/coq/inconsistent.v)

```
Inductive Z := Pos : nat->Z | Neg : nat->Z.  
Axiom zero : Pos 0=Neg 0.  
  
Theorem absurd : False.  
Proof.  
  generalize zero; intros H; inversion H.  
Qed.
```

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.
```



Pré-conditions, gardes, tout ça n'est que logique virtuelle

Source (snippets/coq/head.v)

```
Require Import List.
Variable secret:nat.

Definition head1(l:list nat)(p:l<>nil):
  {h:nat|exists l':list nat, l=h::l'}.
  intros [ | h l] p.
  exists secret; destruct p; apply refl_equal.
  exists h; exists l; apply refl_equal.
Defined.

Definition head2(l:list nat):
  {h:nat|l<>nil->exists l':list nat, l=h::l'}.
  intros [ | h l].
  exists secret; intros p; destruct p; apply refl_equal.
  exists h; intros _; exists l; apply refl_equal.
Defined.
```

Dans les deux cas le code extrait exporte `secret`



Plan

- 1 Toi qui entres ici . . .
- 2 Au-delà des illustrations
- 3 Quelques éléments de conclusion



Extrait de la spécification officielle du langage JAVA relative à la méthode `clone` de la classe `Object`

*The **general intent** is that, for any object `x`, the expression : `x.clone() != x` will be true, and that the expression : `x.clone().getClass() == x.getClass()` will be true, but these are **not** absolute requirements. While it is **typically** the case that : `x.clone().equals(x)` will be true, this is **not** an absolute requirement.*

La spécification des opérations de sérialisation (`writeObject` et `readObject`) est aussi assez intrigante



[C] Pointless pointer operations

Encore un extrait de “*The C programming language (Second edition)*”

The meaning of “adding 1 to a pointer,” and by extension, all pointer arithmetic, is that $pa+1$ points to the next object, and $pa+i$ points to the i -th object beyond pa .

En d'autres termes, l'effet de `p++` dépend du type pointé, et plus exactement de la taille de sa représentation mémoire – qui peut être obtenue notamment par l'utilisation de l'opérateur `sizeof` ; c'est relativement intuitif



[C] Pointless pointer operations

Encore un extrait de “*The C programming language (Second edition)*”

The meaning of “adding 1 to a pointer,” and by extension, all pointer arithmetic, is that $pa+1$ points to the next object, and $pa+i$ points to the i -th object beyond pa .

En d'autres termes, l'effet de `p++` dépend du type pointé, et plus exactement de la taille de sa représentation mémoire – qui peut être obtenue notamment par l'utilisation de l'opérateur `sizeof` ; c'est relativement intuitif

Ceci dit que se passe-t-il exactement si `p` est un pointeur de fonction en architecture CISC ? Cette question est dénuée de sens. . . mais le code correspondant est compilable et exécutable



[C] L'auberge espagnole

Un extrait de “*The C programming language (Second edition)*” de *B. W. Kernighan & D. M. Ritchie*

The direction of truncation for / and the sign of the result for % are machine-dependent for negative operands, as is the action taken on overflow or underflow.

La question est de savoir comment **tester** la conformité d'un compilateur à cette spécification non-déterministe. . . Pensez-y



[C] L'auberge espagnole

Un extrait de “*The C programming language (Second edition)*” de *B. W. Kernighan & D. M. Ritchie*

The direction of truncation for / and the sign of the result for % are machine-dependent for negative operands, as is the action taken on overflow or underflow.

La question est de savoir comment **tester** la conformité d'un compilateur à cette spécification non-déterministe. . . Pensez-y

C'est fait ?



[C] L'auberge espagnole

Un extrait de “*The C programming language (Second edition)*” de B. W. Kernighan & D. M. Ritchie

The direction of truncation for / and the sign of the result for % are machine-dependent for negative operands, as is the action taken on overflow or underflow.

La question est de savoir comment **tester** la conformité d'un compilateur à cette spécification non-déterministe. . . Pensez-y

C'est fait ?

Votre test rejeterait-il un compilateur changeant l'arrondi à **chaque appel**, ce qui permettrait d'avoir $1/-2==1/-2$ évalué à faux ? C'est une instantiation de ce qu'on appelle le *Paradoxe du raffinement*



JAVA est un langage qui se compile en *bytecode* vérifié et interprété par une JVM ; cela peut susciter quelques questions :

- ▶ Peut-on écrire en *bytecode* plus de choses qu'en JAVA ? Les vérifications sont-elles bien pensées au niveau *bytecode* ?
- ▶ Peut-on empêcher l'exécution d'un *bytecode* en restreignant les droits au niveau du système de fichiers (`chmod a-x`) ?
- ▶ Peut-on empêcher l'exécution d'un *bytecode* présent en mémoire en marquant sa page comme non exécutable ?
- ▶ La JVM est-elle compatible avec les mécanismes de prévention d'exécution de certaines pages mémoire ?
- ▶ Quelle relation entre privilèges du *bytecode* et de la JVM ?

Bref quelle sont les conséquences sur l'efficacité ou la possibilité de mettre en œuvre des mécanismes de sécurité système ?



OCAML met en œuvre un GC qui gère la mémoire ; supposons que nous voulions implémenter très proprement une bibliothèque cryptographique manipulant des clés secrètes et/ou privées

- ▶ Comment interdire les copies (compactage ou *swap*) ?
- ▶ Comment minimiser la durée de présence d'une clé en mémoire ?
- ▶ Comment effacer une clé par surcharge ?

Au passage, notons qu'un mécanisme non fonctionnel¹² tel que la surcharge peut aussi être victime d'optimisations de compilation, de mécanismes de cache, de technologies telles que celle des mémoires *flash*, etc.

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



[JAVA] Le facteur humain (sonne toujours 3 fois)

Cf. thedailywtf.com/Articles/Java-Destruction.aspx

Source (snippets/java/Destruction.java)

```
public class Destruction {
    public static void delete (Object object) {
        object = null;
    }
}
```

Plus de 160 commentaires en réponse, dont

- ▶ *Obviously the problem is that he forgot to call `System.gc()`*
- ▶ *It allows the object to be garbage collected... it is still a [problem] because a one liner should be done by the caller*
- ▶ *You know nulling objects to "help" the garbage collector is usually considered bad practice, right?*
- ▶ *Maybe the coder was used to VB, where parameters are sent ByRef by default?*



Sur stackoverflow.com/questions/4456438/how_...

How can I pass the string "Null" through WSDL (SOAP) from ActionScript 3 to a ColdFusion web service without receiving a "missing parameter error" ?

We have an employee whose last name is Null. He kills our employees lookup application when his last name is used as the search term. . .

C'est sans doute une plaisanterie. . . Mais elle fait réfléchir



Source (snippets/c/rosefault.c)

```
int main (void) {
    char* s = "Hello";
    printf ("%s\n",s);
    s[0] = 'h';
    printf ("%s\n",s);
    return 0;
}
```

Source (snippets/c/badoptim.c)

```
struct tun_struct *tun = __tun_get(tfile);
struct sock *sk = tun->sk;
if (!tun)
    return POLLERR;
/* use *sk for write operations */
```



Plan

- 1 Toi qui entres ici. . .
- 2 Au-delà des illustrations
- 3 Quelques éléments de conclusion



A propos de l'enseignement

Comment former un développeur ou un évaluateur ?

- ▶ La sécurité n'est pas un module qui s'intègre parmi d'autres
 - ▶ Elle ne peut pas être totalement déléguée aux "experts"
 - ▶ Que devrait savoir tout développeur à propos de la sécurité ?
- ▶ 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 identifier et écarter 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
 - ▶ ...



A 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 ambiguë¹³
- ▶ *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 voire impossible

13. Voire formalisée. . .



A 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 compilation¹⁴
 - ▶ Modèle mémoire reflétant l'encapsulation
 - ▶ Surveiller le flot d'exécution même en présence de fautes
- ▶ Disposer d'outils maîtrisés voire de confiance

14. Cela peut aussi concerner les architectures. . .



Remerciements

Les exemples de cette présentation sont fournis ou inspirés par :

- ▶ Les laboratoires de l'ANSSI
- ▶ Les participants à l'étude JAVASEC
- ▶ Les participants à l'étude LAFOSSEC
- ▶ 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*
 - ▶ *Rob Kendrick (Some dark corners of C)*

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



Outils et langages utilisés

- ▶ **C** : gcc (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3
- ▶ **ERLANG** : Erlang R14B04 (erts-5.8.5)
- ▶ **JAVA** : Eclipse Java Compiler 0.972_R35x, 3.5.1 release
- ▶ **JAVASCRIPT** : Mozilla Firefox 16.0.2 for Ubuntu canonical - 1.0
- ▶ **SQL** : mysql Ver 14.14 Distrib 5.5.28, for debian-linux-gnu (i686)
- ▶ **OCAML** : The Objective Caml compiler, version 3.12.1
- ▶ **PHP** : Server version: Apache/2.2.22 (Ubuntu)
- ▶ **PYTHON 3**
- ▶ *Shell* : GNU bash, version 4.2.24(1)-release (i686-pc-linux-gnu)

Come, let us go down and confuse their language so they will not understand each other.

Babel, Genesis 11 :7

All animals are equal, but some animals are more equal than others.

Animal Farm, George Orwell

If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization

Gerald M. Weinberg

Everyone knows that debugging is twice as hard as writing programs in the first place. So if you're as clever as you can be when you write it, how will you ever debug it.

Brian Kernighan

The tools we are trying to use and the language or notations we are using to express or record our thoughts, are the major factor determining what we can think or express at all !

Edsger W. Dijkstra

Don't you see that the whole aim of Newspeak is to narrow the range of thought? In the end we shall make thought-crime literally impossible, because there will be no words in which to express it.

1984, George Orwell

Don't get suckered in by comments, they can be terribly misleading.

Dave Storer

Beware of bugs in the above code ; I have only proved it correct, not tried it.

Donald Knuth

**Software and cathedrals are very much the same –
first we build them, then we pray**

Sam Redwine

There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.

C.A.R Hoare