Développement logiciel et cybersécurité

Éric Jaeger [1]

Conférence STL, 21 et 28 octobre 2016

1. D'après les supports CyberEdu, avec P. Chifflier, O. Levillain et B. Morin

Plan

Plan

Le besoin de SSI

Quelques grandes tendances des technologies numériques

- Ubiquitaires, toujours plus complexes et critiques
- Convergence technologique (les mêmes normes s'imposent partout)
- Convergence topologique (tout est interconnecté)
- Convergence physique, concentration (*smartphone*, *cloud*)

Smart Grids, domotique, véhicules autonomes, internet des objets, *e*-sport, *e*-santé (jusqu'au équipements médicaux de type *pacemaker*), *Big Data*, administration électronique, *High Frequency Trading*...

La prise en compte de la SSI [2] devient donc de plus en plus vitale, notamment lors du développement logiciel

2. Ou cybersécurité

Prise en compte de la sécurité

La sécurité est un besoin qui a le défaut de souvent rester implicite [3]

- Peu ou pas d'apports « fonctionnels »
- Le risque n'est pas toujours visible
- Les principaux enjeux concernent parfois des tiers

Les exigences de sécurité doivent être identiées et exprimées au plus tôt

Quelques obstacles à la prise en compte de la sécurité

- Accentue parfois la complexité
- Augmente les coûts de développement
- La sécurité est au mieux invisible, au pire perçue comme une gêne
- La sécurité peut être difficile à valoriser

On ne peut cependant pas parler de « surcoût de la sécurité »

3. Hors les obligations réglementaires

Avertissements

La perception de ce qu'est la SSI est très souvent biaisée

- Confondue avec d'autres problématiques (SdF, rens. . .)
- Restreinte à un sous-domaine (cryptologie, virologie, biométrie. . .)
- Limitée à un type d'enjeux (CNIL, classifié de défense. . .)

Malgré les collisions de vocabulaire, bien distinguer sûreté et sécurité

- La sûreté cherche à éviter des événements redoutés conséquences de phénomènes aléatoires (défaillances, probabilités, dépendances. . .)
- La sécurité cherche à résister à des actions malveillantes par des entites intelligentes et adaptatives (attaques, coûts, rebonds. . .)

Exemple typique : intégrité par Crc ou par signature

Un état d'esprit (1/3)

Prendre en compte la sécurité, c'est compléter le raisonnement fonctionnel avec une approche duale

- Fonctionnel : faire ce qui est attendu
- Sécurité : interdire ce qui n'est pas autorisé, prendre en compte le dysfonctionnel, identifier l'imprévu voire l'impossible, *etc.*

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

☐ `ls` ☐ `cd` ☐ `cp` ☐ `cat` ☐ `rm` ☐ `mv`

Un état d'esprit

L'entête d'un paquet *IP* comporte deux champs adresse, un pour la source, l'autre pour la destination ; comment acheminer un tel paquet ?

- Fonctionnel : seule l'adresse destination est utile
- Sécurité : l'adresse source, inutile, n'est sans doute jamais vérifiée ; on peut donc probablement la falsifier (*spoofing*)

Quelles sont les conséquences d'une falsification de l'adresse source ?

Question subsidiaire : qui peut vérifier l'adresse source ?

Un état d'esprit

Comment décrire les fonctions pour la compression (Zip) et la décompression (Unzip) de fichiers ?

- Fonctionnel : Unzip décompresse les fichiers compressés

$$\forall\,(f : \mathsf{File}),\ \mathsf{Unzip}(\mathsf{Zip}\,f) = f$$

- Sécurité : Unzip doit rejeter les fichiers non compressés

$$\forall\,(c : \mathsf{File}),\ (\neg\exists\,(f : \mathsf{File}),\ \mathsf{Zip}\,f = c) \Rightarrow \mathsf{Unzip}\,c = \mathsf{Error}$$

Selon l'état d'esprit adopté, tout change : la rédaction de la spécification, la pratique de développement (style offensif ou défensif, *etc.*), les objectifs sous-jacents (performances, robustesse), la démarche de tests...

Plan

Objectifs de sécurité

Confidentialité

Seules les entités [4] autorisées peuvent accéder à l'information ou au service

Intégrité

L'information ou le service n'est pas modifiable, n'est modifiable que par les entités autorisées ou toute modification non légitime est détectable

Disponibilité

L'information ou le service est disponible lorsque nécessaire ; cela correspond souvent à de la résilience, la capacité à résister à une panne ou une attaque et/ou à revenir à un état normal après un incident

On retrouve souvent d'autres objectifs dérivés tels que authentification, auditabilité, imputabilité, non répudiation...

4. selon les cas, des personnes, des services, des systèmes

Défense en profondeur

Défense en profondeur

Mettre en place des mécanismes de sécurisation au niveau des différentes couches et zones, ne pas faire reposer la sécurité sur un unique mécanisme

On peut notamment raisonner selon 5 grands axes

- « Prévenir », éviter l'apparition de vulnérabilités
- « Bloquer », empêcher une attaque de parvenir jusqu'aux éléments sensibles et vulnérables
- « Limiter », réduire les conséquences d'une attaque
- « Détecter », repérer, pour pouvoir y réagir, une attaque
- « Réparer », disposer de moyens permettant de remettre en fonctionnement le système suite à une attaque

Non modularité

La sécurité n'est pas modulaire

- Ce n'est pas un service
- Elle ne peut pas être obtenue par l'intégration d'un composant
- C'est une propriété « émergente » au niveau d'un système global

Formulation négative : dans un système décomposés en couches et modules, les différentes vulnérabilités ont tendances à s'aligner et se cumuler, plutôt qu'à disparaître

- Les conséquences d'une vulnérabilité logicielle peuvent aller très au-delà du logiciel concerné

À propos de la « sécurité par l'obscurité »

La sécurité par l'obscurité est à proscrire

- Protéger le code source d'un logiciel, la conception d'un système, l'architecture d'un réseau ne doit pas être une condition de sécurité
- À rapprocher du principe de KERCKHOFFS, en cryptographie

La sécurité d'un système doit reposer sur sa robustesse intrinsèque

En développement logiciel, la sécurité par l'obscurité est une tentation

- Le plus souvent pour proteger la propriété intellectuelle
- Pratiques d'offuscation, qui au mieux ralentissent un attaquant, et posent plus de problèmes qu'elles n'apportent de solutions

Réciproquement, la disponibilité du code source n'est pas une garantie !

De la difficulté de la conception sécurisée

Le développement sécurisé est exigeant et moins valorisant que le *pentesting*

L'attaque est toujours plus « facile » ou attirante

- Il suffit de trouver *un* chemin d'attaque [5]
- Activité éclatante (médiatique, potentiellement rémunératrice, etc.)

La défense est fondamentalement plus difficile

- Il faut (tenter de) couvrir *tous* les chemins d'attaque
- Personne ne remarque qu'un logiciel est robuste

5. Même si certains *exploits* relèvent de vrais exploits

Origines des vulnérabilités

Maladresses

- Bugs
- Fragilités, vulnérabilités
- Complexité
- Au niveau de la spécification, de la conception, de l'implémentation, de la configuration, ...

Recherche de fonctionnalités ou de performances, avant la sécurité

Utilisations ou réutilisations hors contexte

Actions volontaires (piégeage)

- Développeur hostile
- Intrusion dans l'environnement de développement
- Utilisation de code tiers (bibliothèque)

Exemple de tentative de piégeage

Une tentative de modification du noyau Linux [6]

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, que le «et» booléen est paresseux, *etc.*

6. *cf.* `lwn.net/Articles/57135/`

Plan

Plan

Sécurité & méthodes de conception

Les exigences de sécurité s'intègrent bien dans les méthodes classiques de génie logiciel

- Typiquement, le cycle en V
- Ceci étant ces méthodes ne mettent pas l'accent sur la sécurité

Qu'en est-il des méthodes plus « modernes » ?

- Méthode agile, *extreme programming*, *etc.*
- Les avis sur la question sont partagés
- De manière générale, la rapidité (voire l'urgence) d'implémentation est rarement compatible avec la qualité
- Quoi qu'il en soit, la prise en compte de la sécurité ne doit jamais être reportée à plus tard

Cycle de vie d'une application

La sécurité doit être prise en compte à *toutes* les étapes du cycle de vie

- Spécification
- Conception
- Développement
- Test & validation
- Production & maintenance (ainsi que retrait de service)

Plus la prise en compte de la sécurité est tardive, plus la correction peut être coûteuse

- Problème analogue à celui des *bugs*
- Sans parler du coût de l'éventuelle attaque

Environnements de développement (1/2)

Utiliser des logiciels de suivi de version

- Intégrité et suivi strict des modifications apportées aux sources
- Imputabilité des modification et contrôle d'accès aux sources
- Sauvegarde répartie ≪gratuite≫
- Exemples : git, subversion, mercurial, etc.

Isoler et protéger l'environnement de développement

- De la bureautique
- De l'environnement de production [7]
- Les versions de production doivent être issues du serveur de suivi de versions (et non du poste d'un des développeurs)

Idéalement, les versions des logiciels mises en production sont signées, les signatures vérifiées, les clés de signature correctement protégées. . .

7. C'est un hérésie de trouver un compilateur sur un système en production

Environnements de développement (2/2)

Les ateliers de développement (*Integrated Development Environment* ou *IDE*) participent-ils à la qualité et à la sécurité du code ?

+ Options homogènes
+ Possibilité d'interdire globalement certaines fonctions
– Modes différents (*debug vs release*)
– Nombreuses options « cachées » pour le développeur
– Masque les modes et concepts fondamentaux

Et les cadriciels (*Frameworks*) ?

+ Réutilisation de code
+ Code mis à jour régulièrement
? Activités de la communauté autour du projet
? Qualité du code
– Ajout de nombreuses fonctions inutiles
– Écriture du code *pour* le *framework* au détriment de la conception

Un mot sur les méthodes formelles

Les méthodes formelles donnent des *preuves* de *correction* d'un logiciel

- En comprenant ce que signifie cette notion de « correction »
- En comprenant ce que signifie cette notion de « preuves »

Les méthodes formelles apportent des garanties fortes et exhaustives

- Délicates et coûteuses, elles sont appliquées aux logiciels critiques
- Identifier leurs apports et leur limites, en sûreté comme en sécurité, est un sujet complexe [8]

8. *Cf.* http://www.ssi.gouv.fr/uploads/IMG/pdf/jaeger_thesis.pdf

Plan

Prendre en compte la sécurité

Il est nécessaire d'identifier les besoins de sécurité dès la spécification

- Se protéger
- Protéger ses utilisateurs
- Protéger les tiers
- Ne pas oublier les éventuelles obligations réglementaires

Ces exigences sont souvent négligées car

- Elles ne relèvent que rarement d'exigences fonctionnelles pures
- Elles ne sont pas toujours évidentes à discerner
- Elles peuvent être coûteuses à prendre en compte

Quelques exemples d'exigences (1/2)

Les exigences liées à la sécurité doivent être identifiées et explicitées

- Disponibilité d'un service critique
- Intégrité d'une information ou d'un service
- Confidentialité de données (personnelles ou non)
- Traçabilité à des fins d'imputabilité
- ...

Certaines données peuvent devoir être intègres ou confidentielles

- Pendant leur transport
- Pendant leur stockage
- Pendant leur traitement
- L'externalisation est-elle appropriée, si oui sous quels termes ?

Quelques exemples d'exigences (2/2)

Identifier les différents utilisateurs est souvent important

- Même dans le cas d'un logiciel mono-utilisateur, convient-il de distinguer plusieurs identités logiques selon les usages ?

Identifier les privilèges des différents rôles

- Utilisateur, administrateur, auditeur, *etc.*
- Lecture, écriture, exécution, utilisation, modification, *etc.*
- Authentification souvent nécessaire
- Délégation de droits

Problématiques proches liées à la concurrence (rôles et privilèges des différents process ou fils)

Les plis dans le tapis

Certaines spécifications sont des « pousse au crime » en omettant de recommander d'effectuer des vérifications de validité ou de cohérence

Par exemple pour les formats

- données compressées avec un champ annonçant *a priori* la taille des données décompressées [9]
- TLV avec la longueur en mots sur un octet, et pour certains types une note indiquant que la taille ne dépasse pas $x < 255$ mots
- ambiguïtés pouvant mener à des interprétations incohérentes (voire des chimères, *i.e.* des fichiers compatibles avec 2 formats)
- formats avec des blocs dont les positions sont référencées, fragmentation, formats récursifs, macros, *etc.*

9. Allocation suicidaire si la taille est exagérée, *buffer overflow* si elle est sous-estimée

(Contre-)exemples avec SSL/TLS

CVE-2014-3511 : *Downgrade attack*

Le premier fragment du ClientHello ne permet pas d'identifier la version du protocole ; OpenSSL bascule alors en TLS 1.0

CVE-2014-0224 : *Early ChangeCipherSpec*

Renégociation alors que la négociation initiale n'est pas encore terminée, les clés sont dérivées avec un contexte non initialisé[10]

CVE-2014-6593 : *Early Finished*

La négociation est clôturée avant l'échange des authentifiants ; JSSE considère le serveur authentifié

CVE-2014-6321 : *Winshock*

Un buffer overflow côté serveur lors du parsing de l'authentification du client, même lorsque le serveur ne la demande pas

10. Le bug initial, un *SegFault*, avait été « corrigé » sans analyse

Spécification partielle et/ou non déterministe

Une spécification peut être (explicitement ou implicitement) partielle ou non déterministe

Le développeur sera cependant amené à fournir une implémentation totale et déterministe

- Quel est le comportement hors domaine ?
- Quels sont les cas d'erreur et les comportements associés ?
- La portée du non-déterminisme est-elle bien comprise ?

Ce type de considérations s'applique également lors de l'utilisation de bibliothèques tierces

Plan

Quelques principes de sécurité généraux (1/2)

Minimiser le périmètre

- Développer seulement les fonctions nécessaires et suffisantes
- Les fonctions superflues ne servent qu'aux attaquants
- Factoriser le code autant que possible

Privilégier (autant que possible) la simplicité

Séparer et minimiser les permissions et privilèges

- Interdire par défaut plutôt qu'autoriser
- Utiliser des listes blanches plutôt que noires

Quelques principes de sécurité généraux (2/2)

Utiliser des mécanismes de sécurité existants et robustes

- Éviter de réinventer la roue (*Do not implement your own crypto*)
- Utiliser *correctement* (en sécurité, le tout peut être moins que la somme des parties)

Adopter une posture de méfiance

- Toute donnée externe doit être jugée potentiellement toxique

Journaliser

- Afin d'être en mesure de reconstituer la séquence d'événements conduisant à un problème (et le comprendre)
- Prévoir différents niveaux de détail de journaux

Appliquer le principe de défense en profondeur

Architecture (1/3)

L'architecture d'un programme est la base de sa sécurité

- Découper le logiciel en modules simples
- Définir clairement les interfaces et leur accessibilité
- Identifier les relations de confiance entre modules
- Fixer les protocoles de communication entre modules
- Isoler les opérations sensibles (privilégiées, manipulant des secrets. . .) ou dangereuses (*parsing* par exemple)

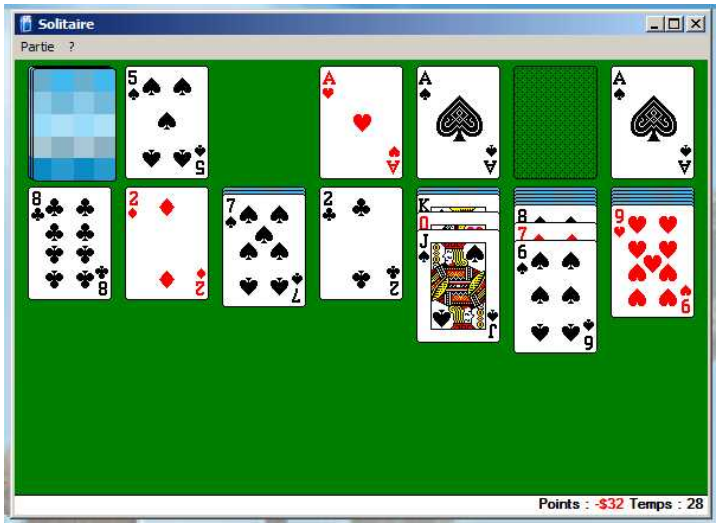
Attention de ne pas aboutir à des architectures trop complexes – et d'éviter que des modules de sécurisation ne deviennent une cible privilégiée

La notion de module utilisée ici doit permettre de garantir une forme de cloisonnement : ADT ou objets (encapsulation), *threads*, *process*. . .

Architecture

Dans le cas des logiciels client-serveur

- Considérer que le client peut être hostile
- Faire les vérifications côté serveur (et, éventuellement, client)
- Implémenter l'intelligence côté serveur

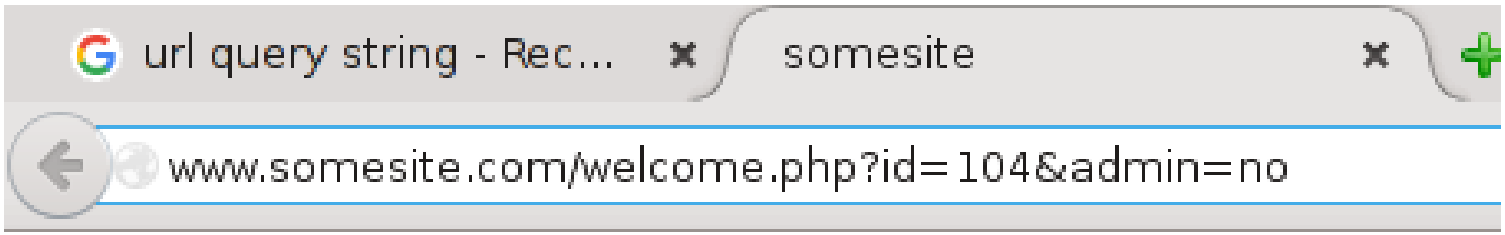


C'est aussi valable pour les interactions entre une application et son interface graphique

Les vérifications ou adaptations de sécurité doivent être faites au plus proche (en temps et en espace) des opérations sensibles

De même on ne déporte pas des paramètres critiques

Que pensez d'une application *web* avec ce genre d'URL ?



Lesquelles de ces propositions permettent de résoudre ce problème ?

- Masquer l'URL
- Renvoyer ces paramètres *via* des champs cachés d'un formulaire
- Renvoyer ces paramètres par du code JAVASCRIPT
- Placer ces paramètres dans des *cookies*

Notion de moniteur

S'assurer du caractère obligatoire de certaines fonctions de sécurité (*i.e.* s'assurer que ces fonctions sont vraiment incontournables), par exemple

- *Proxies* ou *Reverse-proxies*
- Passerelles de chiffrement
- Authentification

Éléments de conception associés

- Automates pour coder l'état d'une fonction
- Moniteurs

Redondance

La robustesse peut parfois être atteinte par la redondance ; en sécurité, cette redondance ne peut généralement pas se limiter à dupliquer un code

- Les deux instances souffriront des mêmes vulnérabilités

Il faut donc envisager un second développement indépendant, ou à défaut considérer la mise en place d'un oracle qui vérifie au lieu de faire

- Trier une liste *vs* vérifier qu'une liste est triée
- Signer un fichier *vs* vérifier la signature d'un fichier
- Normaliser un fichier *vs* valider son format [11]

11. En architecture client-serveur, normaliser côté client, valider côté serveur ?

Interfaces, standards et normes

Les interfaces entre modules présentent des risques

- Toujours tester les données en entrée
- En particulier, les données dont la « nature » change (texte utilisé comme critère dans une requête SQL)
- Identifier les interfaces sensibles et les besoins associés (confidentialité, intégrité, anti-rejeu)

Choisir les bons standards et normes

- Se baser sur des standards pour ne pas réinventer inutilement l'existant (en moins bien)
- Se méfier des standards « à risques », par exemple trop complexes
- Choisir les bons formats et protocoles

Représentation

Choisir une bonne représentation concrète des données abstraites

- Toutes les données sont-elles représentables ?
- Peut-il y avoir plusieurs représentations d'une même donnée ?
- Y a-t-il des représentations ne correspondant à aucune donnée ?
  Notion d'invariant de représentation

Choisir quand possible une représentation bijective ou, à défaut, adopter un style défensif

- Détecter les données non représentables (débordements)
- Normaliser (unicité)
- Vérifier les invariants de représentation (cohérence, validité)
- Que se passe-t-il en cas de contournement des contrôles ?

Plan

Plan

Choix du langage

La question du choix du langage de développement est trop rarement explicitée, alors qu'elle est fondamentale ; certains traits sont intéressants pour le développement sécurisé

- Typage statique fort
- Gestion automatique de la mémoire
- Vérifications dynamiques

Réciproquement, il faut se méfier de certains traits notamment offerts par les langages à objets et/ou interprétés [12]

Il faut aussi savoir exploiter le compilateur

- Activer toutes les options de compilation de « durcissement »
- Ne pas ignorer les avertissements

12. Des illustrations sont fournies dans les planches suivantes

Outils d'analyse

Les outils d'analyse vérifient le respect par le code de certaines propriétés

- Déréférencement de pointeur nul
- Dépassement de tampons
- Opérations de conversion à risque (*cast*)
- *Taint analysis*
- Fuites mémoire
- Situations de compétition (*race conditions*)

Ne pas confondre avec les outils d'audit de code

- Critères lexicaux et syntaxiques
- Taux de commentaires par ligne de code
- Taille des fonctions

Programmation défensive

Minimiser le risque d'introduction d'une vulnérabilité dans un programme en appliquant les bonnes pratiques de développement

- Réutilisation « intelligente » de code
- Vérification systématique des entrées/sorties
- Vérification systématique des codes de retour des fonctions, récupération des exceptions
- Utilisation d'assertions

Références utiles : *CERT Coding Standards* et *OWASP*

Ne pas oublier de vérifier les éventuelles relations entre les entrées – voire entre les entrées et les sorties avant de renvoyer un résultat

Par exemple pour la racine carrée $0 \leq x \wedge \sqrt{x}^2 \leq x \leq (\sqrt{x}+1)^2$

Programmation défensive

Vérifier les données

- Type (notamment données, méta-données, code. . .)
- Valeur
- Traiter les cas de *Confused deputy*, le *Path crawling* en cas d'*upload* de fichiers. . .

Interdire, corriger ou échapper les données non conformes

- Liste blanche plutôt que liste noire
- Est-il pertinent de corriger ?
- Est-il pertinent d'échapper ?

Veiller à la pertinence des vérifications notamment en temps (ToC, ToU), l'idéal étant l'atomicité [13]. . .

13. Voir la vulnérabilité DIRTY COW pour un contre-exemple qui existé pendant 9 ans

Efficacité et dénis de service

De nombreux algorithmes sont retenus pour leur complexité moyenne, mais ont une complexité pire cas catastrophique

- Ce pire cas est statistiquement peu probable

En sécurité, l'attaquant peut favoriser le pire cas

- Tris
- *Hashtable*
- ...

L'approche défensive consiste ici à choisir le bon algorithme ou à détecter les tentatives d'abus

Conventions de développement

Le *coding style* participe à la lisibilité et la qualité d'un programme

Le respect de conventions de développement est donc recommandé pour la sécurité

Exemples de styles

- Règles du noyau Linux
- *Google* Coding Style

Plan

Bogues et vulnérabilités

Une erreur menant à un comportement inapproprié d'un programme sera souvent une source de vulnérabilités – l'attaquant se contentant de placer le programme dans la situation permettant de la déclencher

Dans la suite, nous nous concentrons sur certaines catégories d'erreurs dont les conséquences peuvent être beaucoup plus importantes qu'initialement envisagé

Attaque de la pile système

Rappel sur la corruption de la pile, par exemple *via* un *buffer overflow*

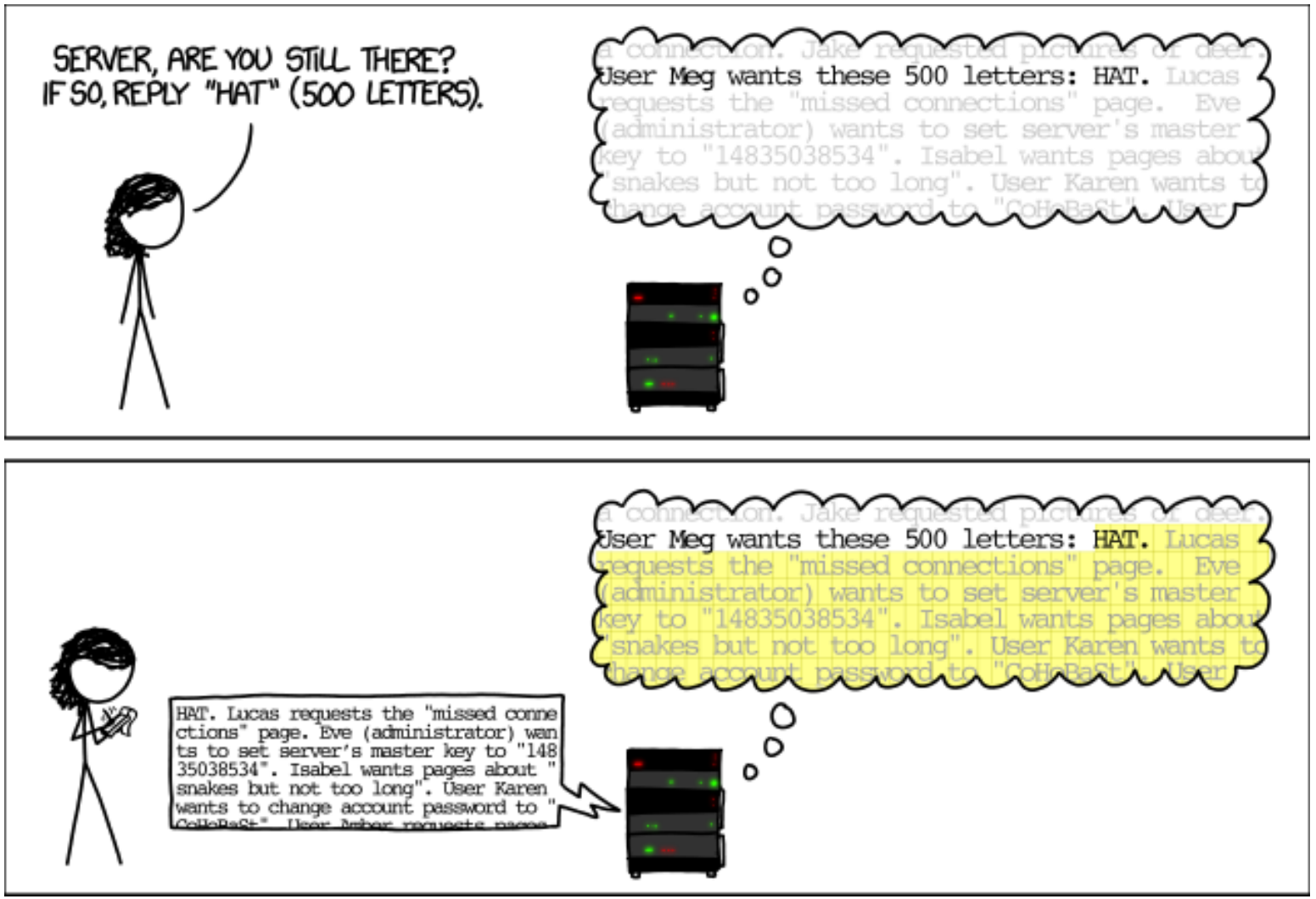
Source (snippets/c/overflow.c)

```
#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;
}
```

On peut changer la valeur de variables ou le flot d'exécution : ici l'adresse de retour est modifiée, après `set` le programme ne revient pas à `main` mais exécute `bad` ; on pourrait même injecter puis exécuter du code !

La faille Heartbleed

Une des meilleures explications vient du site `xkcd.com`



La faille Heartbleed (2/2)

La faille *Heartbleed* (CVE-2014-160) a été révélée en mars 2014

Concrètement, un serveur HTTPS sur deux de la planète était concerné [14] avec une compromission possible

- Des clés privées
- Des mots de passe
- De toute autre information présente en mémoire du process. . .

L'intégration de ce service de sécurité cryptographique a induit une vulnérabilité dont l'impact va très au-delà du périmètre de ce service

C'est un simple oubli de vérification de bornes dans le code d'une fonction non critique du protocole SSL/TLS

14. Sans même parler des autres catégories d'équipements

Coercitions dangereuses (1/2)

On peut vouloir implémenter des fonctions pour manipuler des tableaux en toute sécurité, mais il faut éviter quelques maladresses sur les types

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... Et si `size=150` ?

Coercitions dangereuses (2/2)

Si vous le lui demandez, le compilateur peut parfois vous aider à repérer ces problèmes (`-Wconversion`[15] pour GCC)

C'est ce type de problème qui a mené à 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

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

Format String Attack (1/2)

Une *format string attack* manipule aussi la pile (mais discrètement)

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
```

Format String Attack (2/2)

À noter dans l'exemple précédent que la fonction `strfmtattack` échappe à son contexte, et parcourt celui de sa fonction appelante

C'est indéniablement une vulnérabilité, mais elle n'est pas facile à repérer si on conserve une vision fonctionnelle : difficile d'imaginer forger plus ou moins au hasard une chaîne de caractères qui s'affichera bizarrement

Moralement, on est plus proche d'une attaque par injection : la donnée, une chaîne de caractères, devient quelque chose de différent – une méta-donnée – une fois interprétée par la fonction `printf`

Plan

Maîtrise du code

Identifier le code qui va s'exécuter et le moment de son exécution n'est pas toujours immédiat, en particulier avec les langages à objets

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");
  }
}
```

Ici la référence à `Mathf.pi` signifie qu'à l'exécution la JVM va charger la classe `Mathf` présente sur le système et exécuter son code d'initialisation (code hors méthode marqué `static`)

Maîtrise du code

En JAVA la référence a une classe n'est pas toujours explicite dans le code, et par exemple apparaît dans une sérialisation

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");
  }
}
```

Maîtrise du code (3/3)

Plus généralement

- En objet, de manière évidente la création d'une instance appelle le constructeur ; il ne faut cependant pas oublier d'autres exécutions liées à la finalisation, au chargement de classes, à la désérialisation, ou encore les possibilités offertes par la surcharge. . .
- *Via* l'édition de liens en C comme en OCaml il est possible de faire exécuter du code de bibliothèque préalablement au démarrage de l'application
- . . .

Stratégie et macros

Comprendre quel est le code qui va s'exécuter peut passer par une extension de la notion de stratégie : que se passe-t-il à la compilation, à l'édition de liens, au lancement, pendant l'exécution ou à la fin ?

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++)` ; de même pour `first(x,1/x)` avec `x` valant `0` – les exceptions sont des effets de bord

Plan

Valeurs spéciales

Les comportements à la marge pour certains 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`, *etc.*

Booléens complexes (1/3)

Dans de nombreux langages, les booléens n'existent pas en tant que tels, mais sont obtenus par projection ; ici encore il peut y avoir quelques subtilités

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 "Authentification 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

Booléens complexes (2/3)

Focus sur la vulnérabilité *Goto Fail* de GnuTLS en mars 2014 (`lwn.net`)

> *But this bug is arguably much worse than* Apple*'s, as it has allowed crafted certificates to evade validation check for all versions of* GnuTLS *ever released since that project got started in late 2000.[...]*
> *The* `check_if_ca` *function is supposed to return true (any non-zero value in C) or false (zero) depending on whether the issuer of the certificate is a certificate authority (CA). A true return should mean that the certificate passed muster and can be used further, but the bug meant that error returns were misinterpreted as certificate validations.*

Un problème de langage (typage) et d'Api

Booléens complexes (3/3)

Petit rappel historique sur une vulnérabilité OpenSSL remontant à 2008 (CVE-2008-5077)

Le patch correctif remplace `if (!i)` par `if (i<=0)`, la valeur `i` en question étant le retour d'une fonction vérifiant un certificat, qui était interprétée un peu vite comme un booléen pur sans prendre en compte les autres valeurs correspondant à un code d'erreur

C'est *exactement* le même problème que le *Goto Fail* de GnuTLS en 2014, sur exactement le même type de fonction sur exactement le même type de logiciel critique...

Plan

Maîtrise minimale requise

En guide d'introduction, un petit rappel...

Source (snippets/java/PrivArrayAccess.java)

```
class PrivArray {
  static private int[] tab={1,2};
  static public int[] get() { return tab; } // No set!
  static public void print()
    { System.out.println(tab[0]+","+tab[1]); }
}

class PrivArrayAccess {
  public static void main (String[] args) {
    PrivArray.print();
    PrivArray.get()[1]=3; // Do you set?
    PrivArray.print();
  }
}
```

Le programme affiche `1,2` puis `1,3`...

Protections fragiles

Certains langages proposent des mécanismes *a priori* intéressants en sécurité, par exemple pour cloisonner ; mais ces mécanismes ne sont parfois que des outils d'ingénierie logicielle et se révèlent fragiles

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); }
  }
}
```

Sécurité embarquée

RUBY intègre des mécanismes de sécurité...

> *The first [part] is a mechanism for distinguishing safe data from untrusted, or tainted, data. The second [part] is a technique for restricted execution, which allows you to "lock down" the Ruby environment and prevents the Ruby interpreter from performing potentially dangerous operations on tainted data.*

Lesquels exactement et avec quelle robustesse reste à déterminer...

> *Furthermore, keep in mind that Ruby's security model has not received the kind of careful and prolonged scrutiny that Java's security architecture has.*

Contournements

Dans le cas d'une représentation avec invariant, les valeurs doivent respecter des contraintes, idéalement vérifiées (programmation défensive)

- Dans les constructeurs (programmation objet)
- Lors des transformations

La désérialisation permet en général de contourner toute forme de vérification, que ce soit en JAVA, en OCAML ou en PYTHON – même des hypothèses fortes liées au typage (confusion entre données et code) peuvent ainsi être remise en cause

Les problèmes seront les mêmes avec du *parsing* n'effectuant pas de vérification de cohérence

Mécanismes à géométrie variable

Utiliser en C le modificateur `const` lorsque c'est possible est une excellente pratique

Il faut cependant bien distinguer les conséquences selon les cas

- Chargement en page *read only* pour les constantes globales sur le tas ; la propriété est garantie par le système
- Sur un paramètre passé par la pile, ce n'est qu'une indication

Attention aussi dans le cas des pointeurs de distinguer les pointeurs constants et les valeurs pointées constantes

Plan

Effets de bord

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 `2 2` et la troisième `1 1` : ces opérateurs sont délicats à maîtriser, il est par contre facile de s'en passer ! Ces lignes ne devraient pas pouvoir être compilées sans avertissement. . . et tout développeur devrait s'abstenir d'écrire quelque chose de ce genre

Plan

Partage maximal

Le partage maximal (*hash consing*) est une approche intéressante, mise en œuvre par exemple dans les bibliothèques standards de JAVA

Source (snippets/java/IntegerBoxing.java)

```
Integer a1=42;
Integer a2=42;
if (a1==a2) System.out.println("a1 == a2");

Integer b1=1000;
Integer b2=1000;
if (b1==b2) System.out.println("b1 == b2");
```

Ici `a1==a2` mais pas `b1==b2`, ce qui met en évidence un mécanisme de cache de taille limitée ; mais que se passerait-il si par introspection on accédait à ce cache pour le corrompre ?

Modification d'une valeur partagée (1/2)

En OCaml le code est statique et les chaînes sont mutables ; 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"`

Modification d'une valeur partagée (2/2)

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` [16] ainsi que certains *patterns* de développement usuels basés sur les exceptions

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

Spécialisation par surcharge

La surcharge permet de spécialiser pour adopter le comportement qui semble le plus « naturel », même si ce n'est pas toujours cohérent...

Source (snippets/ruby/intervals.rb)

```
> "a"<"ab"
=> true

> "ab"<"b"
=> true

> ("a".."b")==="ab"
=> true

> ("a".."b").each { |x| print (x.to_s+".") }
a.b. => "a".."b"
```

Localement, tout semble avoir du sens, mais globalement, la sémantique des intervalles de chaînes de caractères n'est pas du tout claire

Empoisonnement par surcharge

La surcharge permet d'envisager des empoisonnements à l'ordre supérieur

Source (snippets/ruby/poison.rb)

```
> 3==4
=> false
> 3.eql?4
=> false
> 3.equal?4
=> false

> class Fixnum; def ==(y); true; end; end
=> nil
> 3==4
=> true
> 3.eql?4
=> true
> 3.equal?4
=> false
```

C'est aussi possible en JavaScript par exemple

Plan

Principe

Les vulnérabilités liées à des possibilités d'injection sont très nombreuses, mais cette notion est souvent présentée dans un contexte particulier – typiquement l'injection SQL ciblant les serveurs *web*

Pourtant, le principe de l'injection est très générique, et c'est le concept même qui doit être compris par les développeurs afin qu'ils identifient la vulnérabilité quelque soit le contexte dans lequel ils la rencontrent

- envoi des données malléables à un interpréteur
- confusion entre données, méta-données et code

Exemples standards des applications web (1/2)

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; //"...`

Exemples standards des applications web (2/2)

Les attaques de type *Cross Site Scripting* (XSS) correspondent également à une injection : cette fois, il s'agit d'insérer dans une chaîne de caractères du code JAVASCRIPT qui sera interprété par le navigateur

De manière assez intéressante, c'est une attaque par rebond dans laquelle le serveur *web* est complice sans être cible ; identifier ce type de vulnérabilité nécessite d'être sensibilisé à la notion d'injection, mais également d'être capable de prendre du recul sur les objectifs de sécurité

D'autres exemples

On retrouve fréquemment dans les langages des fonctions permettant de faire appel à un *Shell*

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 pas !

D'autres exemples

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 aussi impossible à analyser

D'autres exemples (3/3)

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
```

Il serait souhaitable de ne pas écrire de code de ce type – mais en pratique il y en a dans certains *frameworks* pour le développement *web*

Injections cachées

En RUBY, `Kernel.open` et `File.open` permettent d'ouvrir un fichier et ont presque le même comportement... Le premier (celui invoqué par `open` sans plus de précisions) permet en fait d'exécuter une commande *Shell* et de traiter sa sortie comme un fichier

Source (snippets/ruby/injectionshell.rb)

```
> open ("|ls").each { |x| p x }
"beginend.rb\n"
"beginend.rb~\n"
...
```

Sur quel critère ? Le fait que le nom de fichier commence par le caractère `|`

Injections cachées

Le site `rubygems.org` a présenté une vulnérabilité intrigante...

> *RubyGems.org contained a bug that could allow an attacker to replace some .gem files on our servers with a different file that they supplied. We deployed a partial fix on April 2nd and a complete fix on April 4th, 2016. [...] Gems whose name contains a dash (e.g. 'blank-blank') uploaded before that date should be verified by their authors.*

Deux problèmes distincts ont été identifiés, l'un introduit juin 2014, l'autre remontant à la création du site en 2004

Injections cachées (3/3)

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` ?

[Web] Question de philosophie...

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

Plan

Promotion

Dans le code ci-dessous `z` vaut `128` en raison des promotions

Source (snippets/c/cast1.c)

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

Attention donc aux fausses bonnes idées sur les mécanismes anti-débordement pour les tableaux

Source (snippets/c/cast5.c)

```
int main(void) {
  unsigned char shift[256];
  unsigned char base; unsigned char offset;
  ... shift[base+offset]=base; ...
}
```

Coercitions signées

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

Plan

Politique de surcharge et de coercition

En ERLANG `1+1`, `1.0+1.0` et `1+1.0` sont des expressions valides, il ne semble donc pas utile de distinguer entiers et flottants, et pourtant...

Source (snippets/erlang/factorial.erl)

```
-module(factorial).
-compile(export_all).

fact(0) -> 1;
fact(N) -> N*fact(N-1).
```

`factorial:fact(4)` renvoie `24`, mais `factorial:fact(4.0)` provoque un débordement de pile car `0` et `0.0` ne sont pas unifiables

Politique de surcharge et de coercition

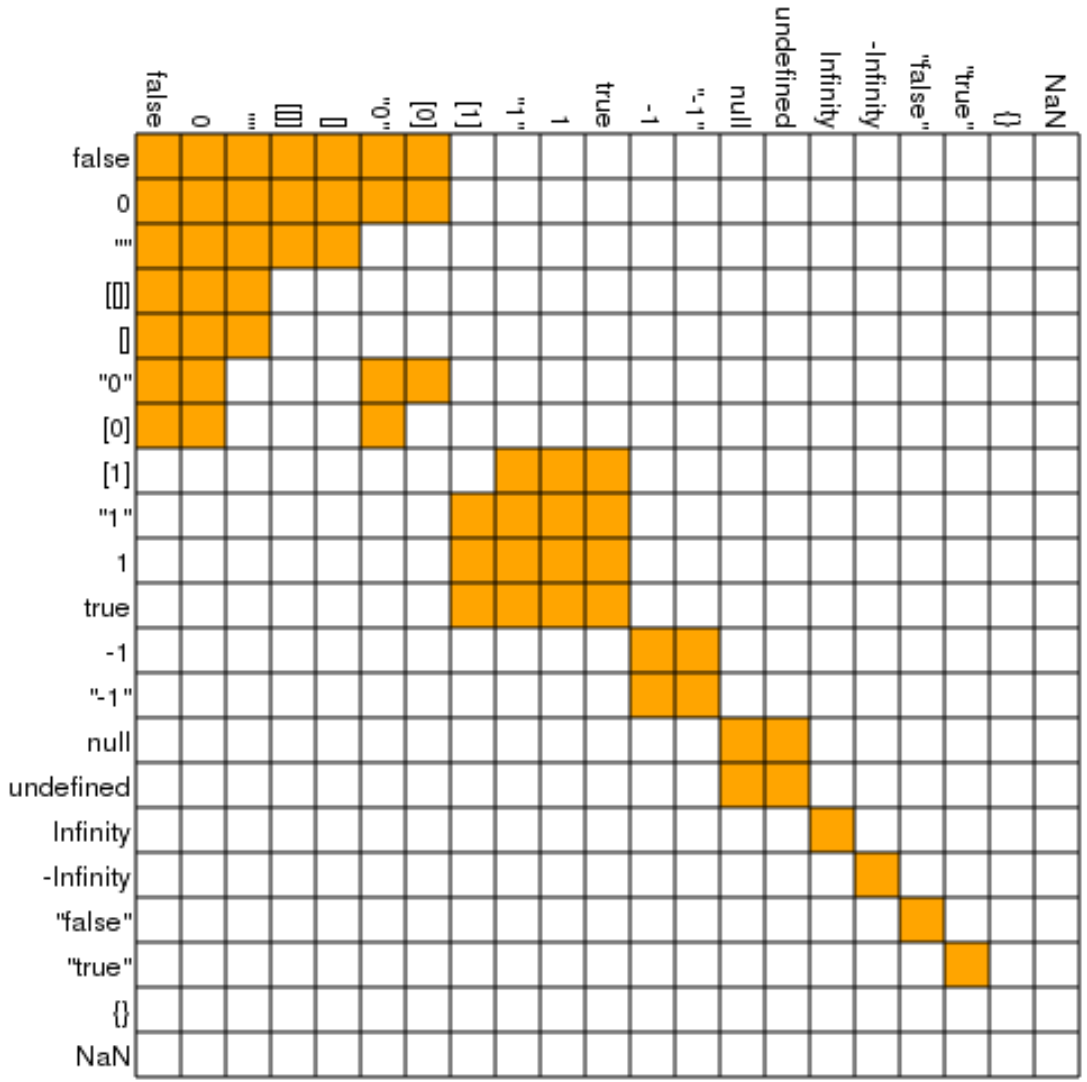
De même en JavaScript, `0=='0'` est vrai mais cette égalité ne vaut pas pour l'unification dans le cas d'un `switch` ; avec les coercitions `'0'==0` et `0=='0.0'` sont vrais mais pas `'0'=='0.0'` : l'égalité n'est plus transitive

La perte de propriétés intuitives peut se produire avec d'autres opérateurs, par exemple l'associativité de `+` : le code suivant affiche `3Foo`, `Foo12` et `Foo3`

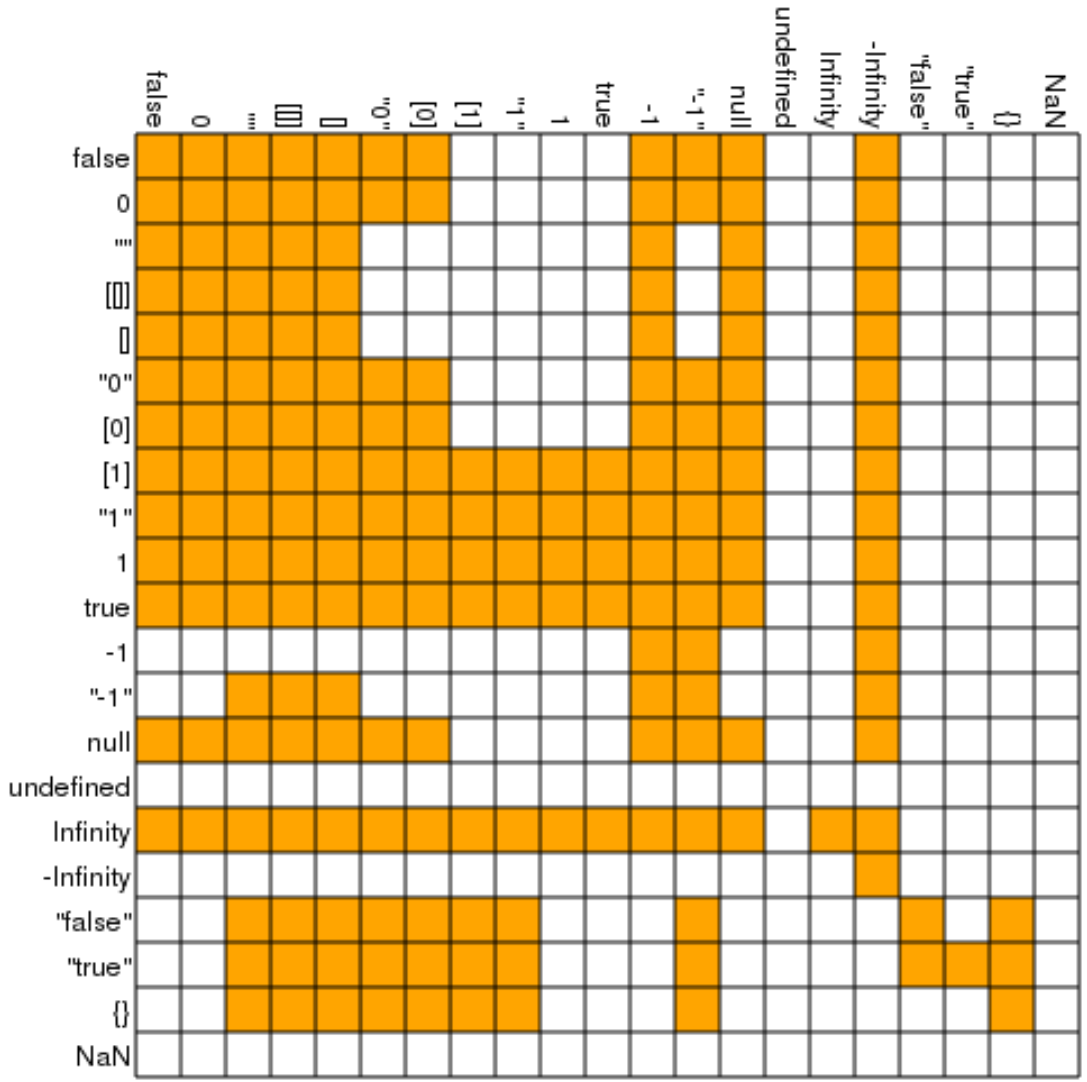
Source (snippets/js/cast3.js)

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

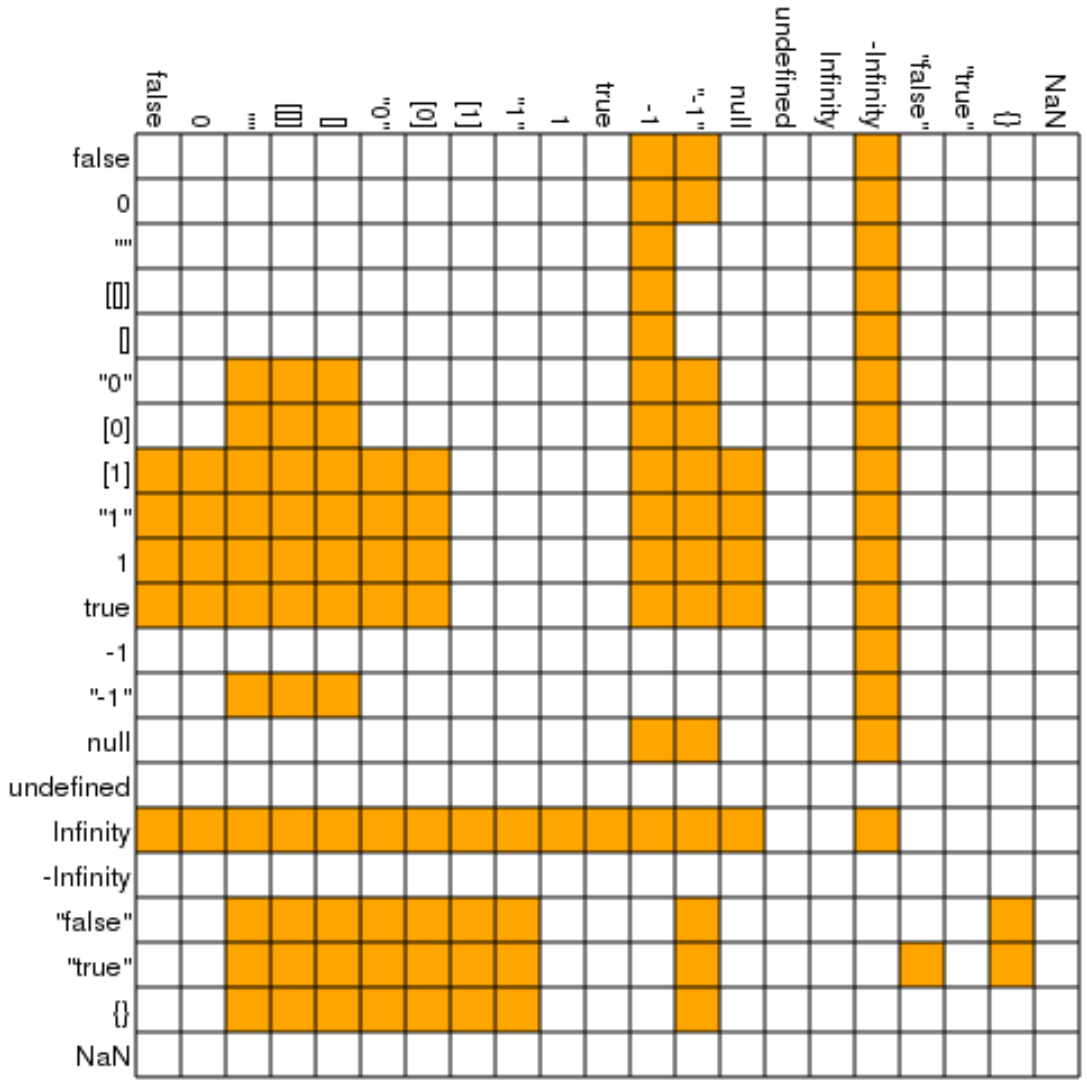
Les 3 tables sur les comparaisons JavaScript montrent d'autres bizarreries, notamment `null<=false` est vrai alors que `null==false` et `null<false` sont faux !


false
0
""
[[]]
[]
"0"
[0]
[1]
"1"
1
true
-1
"-1"
null
undefined
Infinity
-Infinity
"false"
"true"
{}
NaN

Égal ==



Plus petit ou égal <=


false
0
""
[[]]
[]
"0"
[0]
[1]
"1"
1
true
-1
"-1"
null
undefined
Infinity
-Infinity
"false"
"true"
{}
NaN

Plus petit <

Mauvaise gestion de la portée (1/2)

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

Source (snippets/python/listcomp.py)

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

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

Mauvaise gestion de la portée (2/2)

On peut produire des exemples similaires en PHP

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` donc la variable `var` est écrasée et survit à la boucle ! Les plus audacieux pourront aussi tester `foreach ($tab as $tab)`

Plan

Optimisations non conservatives

Que font les deux codes suivants ?

Source (snippets/c/strategy3.c)

```
int zero(int x) { return 0; }
int main(void) { int x=0; x=zero(1/x); return 0; }
```

Source (snippets/c/strategy3b.c)

```
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

Traits du source non préservés

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(); }
}
```

Mais elles n'existent pas en *bytecode* ! Le compilateur doit remettre tout à plat et retirer les `private` pour préserver l'accessibilité

Problématique de l'effacement

Les mécanismes d'optimisation d'un compilateur cherchent à préserver le comportement observable d'un programme

Source (snippets/c/memset.c)

```
void encrypt_data(char *buf)
{
  char secret_key[256] = "...";
  /* ... */
  /* suppression de la cle en memoire */
  memset(secret_key, '\0', sizeof secret_key);
}
```

L'appel à `memset` est supprimé par le compilateur à partir de `-O1`

Dans certains cas, préserver l'observable ne suffit donc pas

Coercitions abusives

L'égalité == de PHP propose des mécanismes de coercition proches de ceux de JAVASCRIPT, tout en préservant la transitivité ; mais le résultat est encore pire

Source (snippets/php/castincr.php)

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

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

if ("0xF9"=="249") { print("Equal\n"); }
else { print("Different\n"); }
```

La première ligne affiche `3` (entier)

La seconde ligne affiche `2d9` (chaîne), `2e0` (chaîne) puis `3` (flottant)

La troisième ligne affiche `Equal` : il y a coercition même pour comparer deux valeurs du même type

Coercitions abusives

Quel rapport avec la sécurité ? Voici un exemple

Source (snippets/php/hash.php)

```
$s1='QNKCDZO'; $h1=md5($s1);
$s2='240610708'; $h2=md5($s2);
$s3='A169818202'; $h3=md5($s3);
$s4='aaaaaaaaaaaumdozb'; $h4=md5($s4);
$s5='badthingsrealmlavznik'; $h5=sha1($s5);

if ($h1==$h2) print("Collision\n");
if ($h2==$h3) print("Collision\n");
if ($h3==$h4) print("Collision\n");
if ($h4==$h5) print("Collision\n");
```

`Collision` est affiché 4 fois, mais ne concluez pas trop vite que MD5 et SHA1 sont mis en cause ici

Plan

Apple's Goto Fail

Encore un bug dans une bibliothèque cryptographique révélé en 2014

Source (snippets/c/gotofail.c)

```
/* Extract from Apple's sslKeyExchange.c */
if ((err=SSLHashSHA1.update(&hashCtx,&serverRandom))!=0)
    goto fail;
if ((err=SSLHashSHA1.update(&hashCtx,&signedParams))!=0)
    goto fail;
    goto fail;
if ((err=SSLHashSHA1.final(&hashCtx,&hashOut))!=0)
    goto fail;
```

La syntaxe n'aide pas et le compilateur ne semble pas se préoccuper de signaler du code manifestement mort

À propos des commentaires (1/3)

Petite caractéristique intriguante 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 lorsque la coloration syntaxique n'est pas conforme)

À propos des commentaires (2/3)

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)

À propos des commentaires (3/3)

On peut également jouer sur une ambiguité liées aux différentes normalisations de C et C++

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`

Encodages

Un encodage tel que UTF-8 est très délicat à traiter, et justifierait de nombreuses remarques de sécurité ; nous nous limitons ici à une illustration dans laquelle le compilateur JAVA accepte cet encodage pour le source

Source (snippets/java/Preprocess.java)

```
public class Preprocess {
  public static void ma\u0069n (String[] args) {
    if (false==true)
    { //\u000a\u007d\u007b
      System.out.println("Bad things happen!");
    }
  }
}
```

`Bad thing happens` : le code source est donc préprocessé

Encodages

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

Gestion des séparateurs (1/2)

Certains choix de syntaxe peuvent mener à des erreurs lors du développement

Source (snippets/ruby/break.rb)

```
def test1 (x,y)
  x +
  y
end

def test2 (x,y)
 x
 + y
end
```

`test1(1,2)` retourne `3`, `test2(1,2)` retourne `2`

Gestion des séparateurs (2/2)

Une autre bizarrerie de la syntaxe de RUBY

Source (snippets/ruby/parentheses.rb)

```
> def test(x); x*2; end
= > nil

> test 1
=> 2
> test(1)
=> 2
> test(1)+1
=> 3
> test (1)+1
=> 4
```

Plan

Développements à haut niveau de sécurité

Ce qui précède fournit des avertissements et des recommandations destinées aux développeurs d'applications standards

Dans cette partie, pour la culture, quelques aspects plus avancés sont mentionnés, utiles pour des développements à haut niveau de sécurité

- Criticité
- Exposition et niveau de menace

Quelques menaces avancées

Piégeage en développement

- Bogues
- Fonctionnalités cachées
- Canaux cachés (*covert channels*)

Accès physique

- Surveillance d'une interface, d'un bus
- Modifications de l'environnement matériel ou logiciel
- Perturbations ou injection de fautes matérielles [17]

Canaux auxiliaires (*side channel attacks*)

- Énergie
- Temps
- Usage de la mémoire, des caches

17. Parfois possibles par attaque logique, *cf.* par exemple RowHammer

Quelques mécanismes avancés

Pour des développements à haut niveau de sécurité, des propriétés spécifiques peuvent donc être recherchées aux travers de mécanismes particuliers

- Cloisonnement fort
- Zéroisation
- Perturbation ou suppression des canaux auxiliaires
- Mesures d'intégrité de l'environnement ou du programme
- Architectures avancées et recours au matériel (cartes à puce)
- ...

Fonctionnalités difficiles

La génération d'aléa, lorsqu'elle a des objectifs de sécurité, est

- complexe
- non spécifiable
- non testable

Exemples : bug OpenSSL sur *Debian*, *Mining your P's and Q's*

Au passage, les propriétés attendues d'un aléa sont extrêmement dépendantes du contexte

Gestion de la mémoire

Dans un langage à Gc la mémoire est gérée automatiquement, ce qui permet de se prémunir *a priori* contre un certain nombre d'erreurs

Ceci étant, pour une implémentation à haut niveau de sécurité gérant des clés cryptographique, cela peut poser quelques problèmes [18]

- 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 fonctionnel tel que la surcharge peut aussi être victime d'optimisations de compilation, de mécanismes de cache, de technologies telles que celle des mémoires *flash* ou de tout autre mécanisme d'optimisation du même type

18. Ce n'est pas forcément simple non plus dans un langage tel que le C en raison des optimisations faites par le compilateur

Cloisonnement fort mais contournable (1/3)

OCaml offre un mécanisme d'encapsulation fort basé sur les modules

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;;
```

C'est un boîte fermée ; la valeur `id` est visible et celle de `key` masquée

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

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

Cloisonnement fort mais contournable (2/3)

OCaml permet de définir des fonctions polymorphes, *i.e.* qui n'analysent pas toute la structure de leurs paramètres, mais propose aussi un opérateur de comparaison structurel et polymorphe, ce qui est contradictoire ; il est dès lors possible de contouner le cloisonnement des modules

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

Cloisonnement fort mais contournable (3/3)

Pas convaincu ? Remettons en cause le typage d'OCaml

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'

Plan

Le logiciel dans son environnement

Exploiter au mieux les fonctions de sécurité qu'offre l'environnement d'exécution du logiciel, notamment le système d'exploitation

- Mécanismes de contrôle d'accès
- Protections génériques contre l'exécution de code arbitraire
- Mécanismes d'isolation, de *sandboxing*, *etc.*

Ne reposer *exclusivement* sur ces protections génériques

- Les protections W⊕X, ASLR et canaries sont efficaces mais un logiciel doit en premier lieu être exempt de failles de type *buffer overflow*
- Impacts des machines virtuelles, mécanismes JIT, *etc.*
- Proscrire l'utilisation abusive de comptes utilisateurs privilégiés [19]

19. Minimiser ce qui est donné, mais aussi ce qui est exigé

Modèles d'exécution

JAVA est un langage qui se compile en *bytecode* vérifié et interprété par une JVM, dès lors

- les vérifications devraient être pensées au niveau *bytecode*, surtout si l'expressivité des deux langages diffère
- l'exécution d'un *bytecode* ne peut être empêchée par le mécanisme des droits sur le fichier (`chmod a-x`) ou le marquage de la page mémoire où il est stocké
- utiliser une JVM avec compilation JIT est incompatible avec les mécanismes de prévention d'exécution de pages mémoire
- les privilèges accordés au *bytecode* par le système sont *a priori* ceux accordés à la JVM

Plan

Tests

L'approche classique du test ne permet pas d'apporter de garanties fortes sur la robustesse ou la sécurité d'un logiciel

- Le test est une approche fonctionnelle (conformité)
- On teste par rapport à ce que l'on attend
- La quantité de code requise pour des tests « suffisants » dépasse (de loin) celle du code testé !

Program testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence.

E.W. Dijkstra

Test et paradoxe du raffinement

Pour illustrer la difficulté du test, considérons une spécification non déterministe : 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

Un test standard rejetterait-il un compilateur changeant l'arrondi à chaque appel, ce qui permettrait d'avoir `1/-2==1/-2` faux ?

Un mot sur le *fuzzing*

Le *fuzzing* est l'arme de prédilection des attaquants pour identifier les failles potentielles d'un logiciel ; cela consiste à envoyer de la « boue » à une application et voir quand elle plante

Cette méthode relève du niveau zéro de l'évaluation. . . Il est regrettable qu'elle fonctionne aussi bien en pratique

Vérifications à l'exécution

Outils de vérification à l'exécution

- Debugger (gdb)
- Outil de vérification (*Valgrind*)
- ...

Ces outils

- Sont utiles pour comprendre un comportement douteux ou un effet de bord
- Permettent d'explorer l'environnement d'exécution
- Demandent une compréhension plus fine
- Ne peuvent pas être exhaustifs

Configuration

Les erreurs lors de la configuration ou l'intégration du logiciel peuvent ruiner les étapes précédentes

Tout logiciel possède une configuration par défaut

- C'est généralement la plus utilisée
- D'où la nécessité de « sécuriser par défaut »

Exemples de mauvaises pratiques

- Ne pas imposer de changer le mot de passe par défaut [20]
- Autoriser une suite cryptographique faible, voire nulle
- Fournir des exemples de mot de passe dans la configuration ou la documentation

20. *A fortiori*, pas de mot de passe en dur non plus, ni de *backdoor*

Maintenance et évolutions

Les évolutions d'un logiciel sont inévitables

- Une correction peut-elle créer un nouveau problème ?
- Les effets de bords des changements sont difficiles à maîtriser, comment les changements sont-ils testés ?
- Les conditions de sécurité des fonctions doivent être documentées

Un projet de développement logiciel devrait prévoir le « maintien en condition de sécurité » au même titre que le « maintien en condition opérationnelle »

Ne pas oublier aussi la gestion des dépendances

- Exemple : multiples vulnérabilités trouvées dans la bibliothèque OpenSSL sur laquelle s'appuient nombre de logiciels pour la sécurisation du transport des données.

Prendre en compte une vulnérabilité

Un éditeur devrait se préparer à réagir en cas de découverte d'une faille sur ses logiciels

- Au minimum, mettre en place un canal de remontée d'alertes
- *Responsible vs Full-disclosure* pour l'inventeur, par contre *Bug secrecy* à proscrire pour l'éditeur

Mesurer l'impact et l'exploitabilité d'une vulnérabilité

Développer, tester et déployer le correctif

- Plus ou moins en urgence en fonction de la gravité du problème
- Anticiper la survenue de vulnérabilités analogues
- Ne pas se contenter du *one-liner* qui corrige (mal) le `strcpy` malencontreux

À votre disposition pour toute question

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

Gerald M. Weinberg

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

Sam Redwine

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

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

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.

C.A.R. Hoare

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

Babel, Genesis 11 :7