Sécurité des implémentations pour la cryptographie

Partie 3 : Résistance aux attaques non-crypto

Benoît Gérard

5 décembre 2017

Plan du cours

Étape 1

Définition du besoin et de l'architecture au niveau système.

Étape 2

Définition de l'interface carte/terminal : API exposée par la carte.

Étape 3

Implémentation d'une version résistante aux attaques non-crypto.

Étape 4

Implémentation d'algo. crypto. résistante aux attaques distantes.

Étape 5

Implémentation d'algo. crypto. résistante aux attaques locales.

Sommaire de la session

Sommaire de la session

Plan

Test des paramètres
Règles

Règle (indiscutable)

Tester les paramètres entrés par l'utilisateur.

Risques :

- Injection SQL
- Attaque sur ECDH (point not on the curve)
- Exploitation d'overflow présents dans le code
- ...

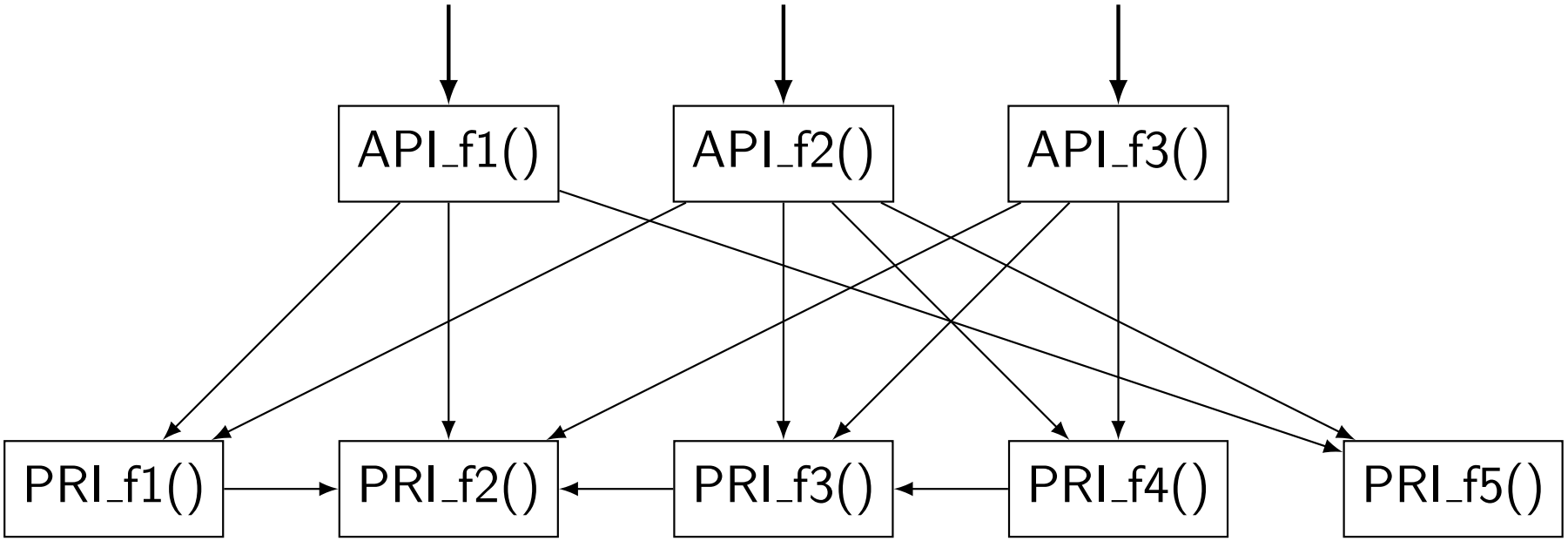
Règle (à adapter)

Tester les paramètres en entrée de fonction.

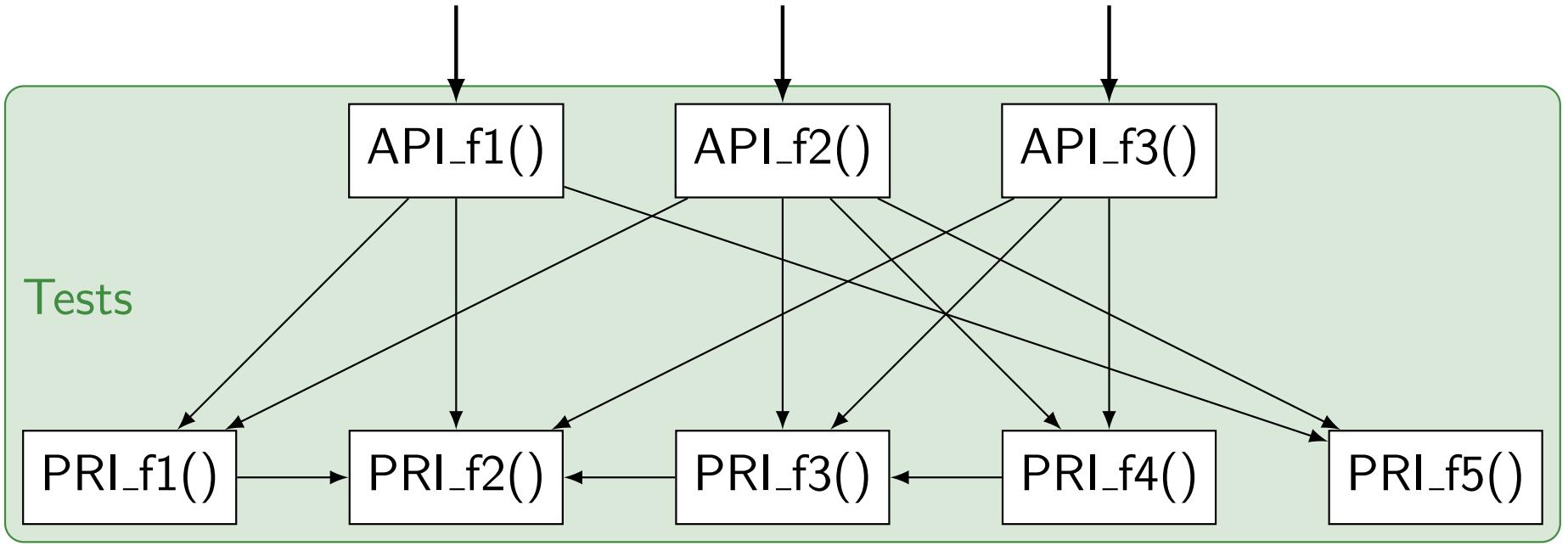
✓ Permet de détecter cetains bugs.

x Tests parfois coûteux en performances.

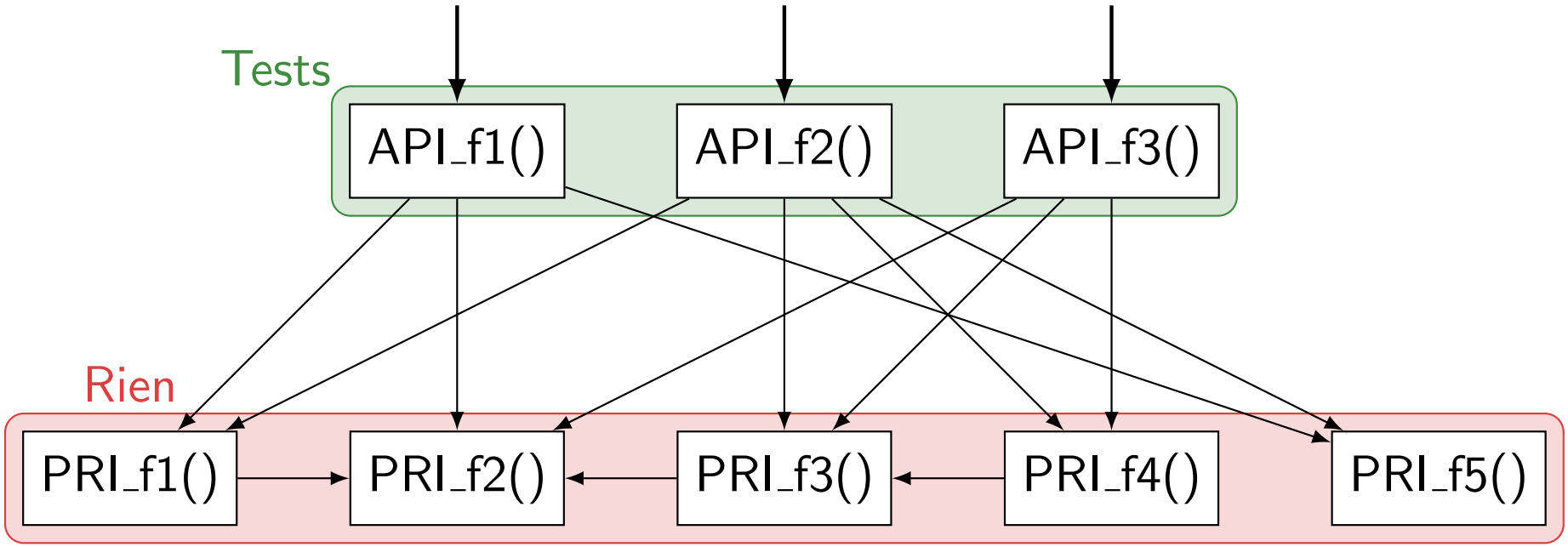
Test des paramètres
Architecture logicielle



Test des paramètres
Architecture logicielle



Test des paramètres
Architecture logicielle



La seconde solution implique une grande confiance dans le code :

- utilisation d'outil d'analyse statique,
- campagnes de tests intensives.

Elliptic-Curve Diffie-Hellman : DH avec $(E, +)$ à la place de $(G, \times)$.

Paramètres :

- une courbe $E(a, b, \mathbb{F}_q)$ avec q puissance d'un premier,
- $P \in E$ avec E d'ordre π (grand premier).

Échange :

1. Alice génère s_A (*secret*) et envoie $Q_A = [s_A]P$ à Bob,
2. Bob génère s_B (*secret*) et envoie $Q_B = [s_B]P$ à Alice,
3. clef partagée $[s_A]Q_B = [s_B]Q_A = [s_A s_B]P$.

Sécurité

Logarithme discret sur un groupe d'ordre π premier $\rightarrow O(\sqrt{\pi})$.

Équation de Weierstrass : courbe définie par (a, b)

$$y^2 = x^3 + ax + b$$

Addition

$$x_{P+Q} = \frac{(y_P - y_Q)^2}{(x_P - x_Q)^2} - x_P - x_Q \quad , \quad y_{P+Q} = \frac{y_P - y_Q}{x_P - x_Q}(x_P - x_r) - y_P.$$

Doublement

$$x_{2P} = \frac{(3x_P^2 + a)^2}{4y_P^2} - 2x_P \quad , \quad y_{2P} = \frac{3x_P^2 + a}{2y_P}(x_P - x_s) - y_P.$$

Oscar :

1. génère une courbe $E'(a, b', \mathbb{F}_q)$,
2. choisit un point $Q_O \in E'$ d'ordre petit r,
3. envoie Q_O à Alice et récupère $[s_A]Q_0$,

Les calculs effectués par Alice restent valide sur E' ! ! !

Ici on se place dans le sous-groupe engendré par Q_0

$$\Longleftrightarrow$$

retrouver $s_A \to O(\sqrt{r})$.

Solution

Comme toute entrée extérieure : il faut tester !
Ici l'appartenance du point reçu à la courbe E.

Plan

Règle (indiscutable)

Indiquer (et vérifier) la taille d'un buffer passé en argument.

Risques :
Écrire/Lire hors de la zone prévue = BOF/BUF (*Buffer Over/Underflow*)

- erreur d'exécution,
- lecture de données sensibles,
- écriture de données corrompues,
- exécution de code malicieux.

Technique ROP (*Return Oriented Programming*)

Petit BOF $\Rightarrow$ pas assez de place pour le code malicieux.
ROP = utiliser des bouts de code déjà présents.

Gestion des tailles

Exemple de vulnérabilité non détectée

```
#define MAX_DATA_SIZE 4
struct{
  uint32_t pub[MAX_DATA_SIZE];
  uint32_t priv[MAX_DATA_SIZE];
} data_t;
```

```
void CopyData(uint32_t *src, uint32_t *dest, unsigned size)
{
  for ( unsigned i = 0 ; i < size ; i++ )
    dest[i] = src[i];
}
```

```
void process(data_t *data)
{
  Copy(data->pub,memory,8); // get 64 first bits
  Erase(data); // erase sensitive data
}
```

Nom	Heartbleed	CVE	2014-0160
Logiciel	openSSL		
versions	1.0.1 $\rightarrow$ 1.0.1f	dates	12/11 $\rightarrow$ 04/14

Heartbeat :

- ne pas perdre le contact avec le serveur,
- messages de ping/pong pour garder la session active,
- pratiques réseau : laisser de la flexibilité au message de ping.

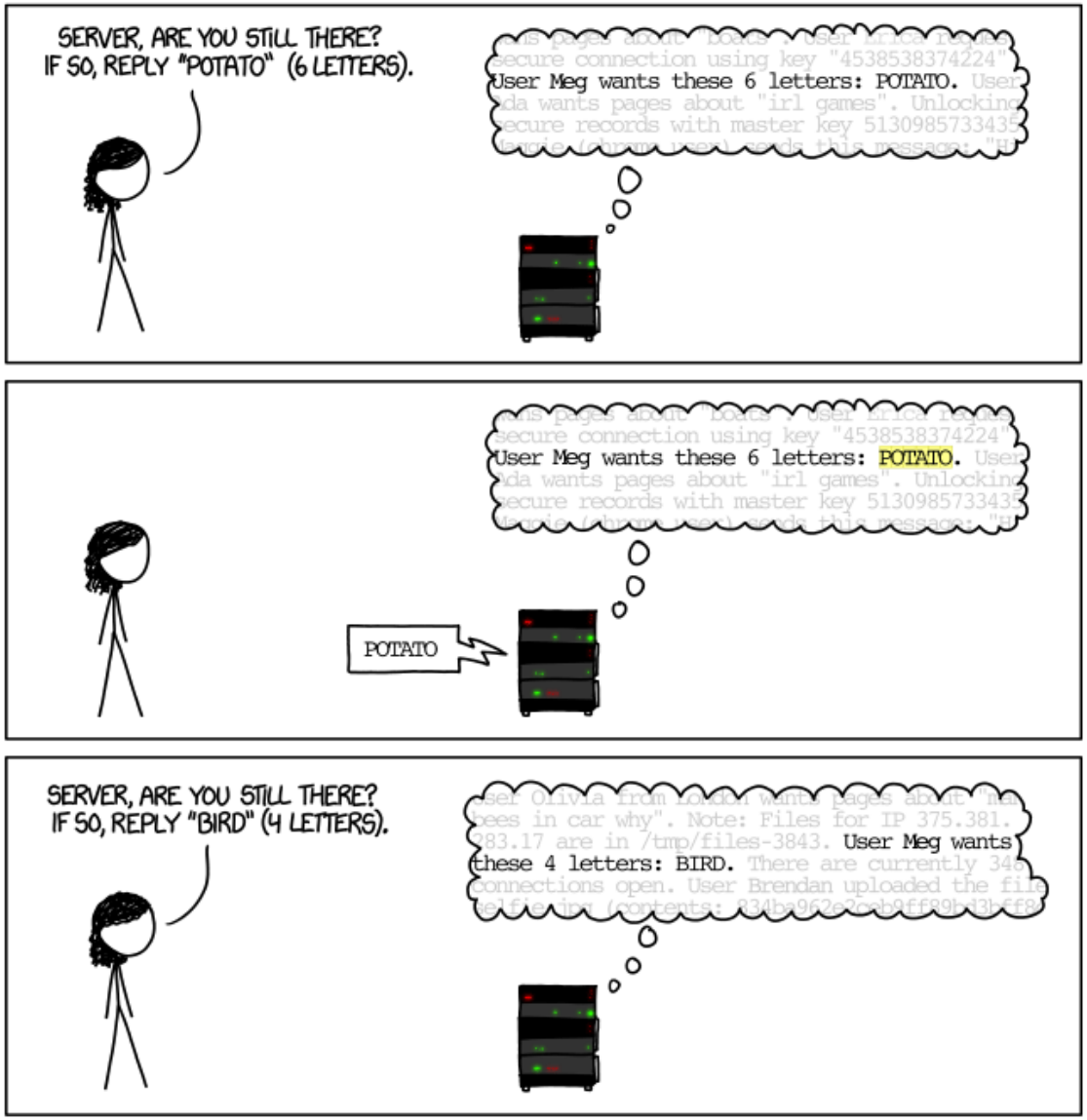
Heartbleed :

- mauvaise gestion des tailles de buffers

$\Rightarrow$ possibilité de *dumper* du contenu mémoire.

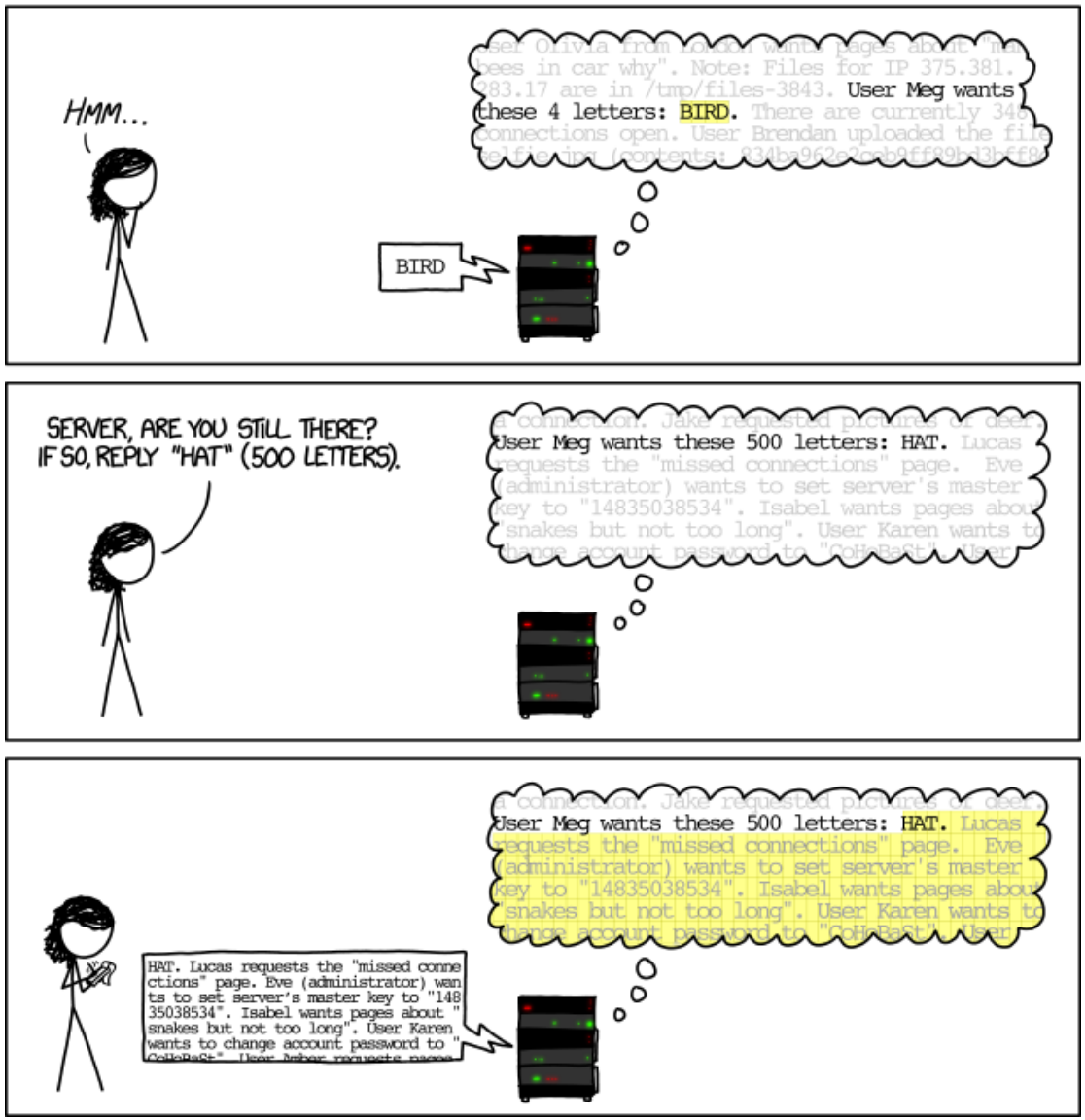
Gestion des tailles

Heartbleed par xkcd 1



Gestion des tailles

Heartbleed par xkcd 2



Attaque indétectable ! ! !

- 500000 certificats potentiellement corrompus,
- tous les mots de passe potentiellement corrompus,
- liste impressionnante d'applications vulnérables.

Dégâts atténués pour les échanges avec PFS.

Rumeur

La NSA aurait rapidement découvert la faille et n'aurait (évidemment) pas averti les développeurs.

Plan

Règle (indiscutable)

Utiliser des codes d'erreur et les tester.

- ✓ facilite le débogage,
- ✓ clarifie le traitement des comportements anormaux,
- x code plus lourd,
- x attention à ne pas donner d'infos à l'attaquant !

1. Toute fonction retourne **uniquement** un code (d'erreur ou OK).
2. Toujours initialiser un code retour à un code d'erreur.
3. Toujours décrémenter un compteur de nombre d'essais *a priori*.

Gestion des erreurs

Ne pas mélanger les choses

Nom	*anonyme*	CVE	2008-5077
Logiciel	OpenSSL	versions	0.9.1c $\rightarrow$ 0.9.8b (7 ans)

Non respect de 1.

Toute fonction retourne uniquement un code (d'erreur ou OK).

```
ok = EVP_VerifyFinal();
if ( !ok )
  ERROR;
else
  GOOD_CERTIFICATE;
```

Or, `ok` $= 0$ si mauvais certificat,
> 0 si bon certificat,
< 0 en cas d'erreur.

La même en 2014 dans la fonction `check_if_ca()` de GnuTLS ...

Nom	goto fail	CVE	2014-1266
Logiciel	iOS	versions	7.0.6

Non respect de 2.

Toujours initialiser un code retour à un code d'erreur.

```
    if ((err = SSLHash.Init(&hashCtx)) != 0)
    goto fail;
    if ((err = SSLHash.update(&hashCtx, &data)) != 0)
    goto fail;
    goto fail;
    if ((err = SSLHash.final(&hashCtx, &hashOut)) != 0)
    goto fail;
    err = sslRawVerify(...);
    ...
    fail:
    return err;
```

Gestion des erreurs

Éviter goto fail en respectant les recommendations

```
unsigned retour = ERROR;
...
err = SSLHash.update(&hashCtx, &data);
if ( err != OK ) {
  retour = err;
  goto fail;
}
goto fail;
err = SSLHash.final(&hashCtx, &hashOut);
if ( err != OK ) {
  retour = err;
  goto fail;
}
err = sslRawVerify(...);
...
fail:
return retour;
```

Non respect de 3.

Toujours décrémenter un compteur de nombre d'essais a priori.

```
unsigned nb_tests = 10;
...
error_t check(unsigned a, unsigned ref, bool *ok) {
  ...
  *ok = false;
  if ( nb_tests == 0 )
    return SECURITY_ERROR;
  if ( a == ref )
  {
    nb_tests = 10;
    *ok = true;
  }
  else
    nb_tests --;
  return NO_ERROR;
}
```

Plan

Règle (indiscutable)

Toujours effacer une *donnée sensible* après utilisation.

Données sensibles :

- nonce,
- donnée intermédiaire de calcul,
- clef
- ...

Risques :

- attaque “cold boot”,
- espionnage mémoire,
- sortie accidentelle d’un secret.

Autres bonnes pratiques

Limiter les `return`

Règle (quasi-indiscutable)

Une fonction possède un seul point de sortie.

Permet :

- de gérer proprement les effacements/libérations de mémoire,
- de limiter les risques de ROP.

```
error_t foo(unsigned a) {
  unsigned tab = malloc(1000);
  tab[0] = 1;
  if ( a >= 1000 )
    return ERROR_PARAM;
  else
    tab[a] = 1;
  free(tab);
  return OK;
}
```

Autres bonnes pratiques

Faire attention aux over/underflows arithémtiques

Soit

- tester les valeurs en entrée d'une opération arithmétique,
- prouver qu'un over/underflow est impossible + commentaire.

Exemple 1

Compteur 32-bits remis à 0 par incrémentation en 15 minutes :

- obtention de droits uniquement accessibles à l'initialisation.

Exemple 2

Code pour paiement électronique :

- achat de \$370,000.00
- conversion en yens $\rightarrow 43,113,493.36¥$
- or, 2^{32} sen $= 42,949,672.96¥ \rightarrow 163,820,40¥ = \$1,405.91$

Autres bonnes pratiques
Tester le code 1/2

Règle (indiscutable)

Effectuer des tests du code :

- tester le fonctionnement normal,
- tester les cas d'erreur,
- envoyer des paramètres *hors calibre*,
- faire tourner des tests *d'endurance*,
- et en cours de développement, tests de non-regression.

Anaylse des jeux de tests :

- couverture de code,
- couverture d'états,
- couverture des transitions.

Autres bonnes pratiques
Tester le code 2/2

Différents niveaux de tests :

- tests pour chaque fonction (tests unitaires),
- uniquement les fonctions de l'API,
- uniquement les appels de l'application utilisatrice.

On peut prendre le risque de faire moins de tests aux niveaux bas pour gagner du temps . . . mais c'est souvent un mauvais calcul.

En testant

On aurait évité

- goto fail,
- faille des certificats GnuTLS/OpenSSL,
- et beaucoup d'autres non présentées ici.

Autres bonnes pratiques

Faire simple et maintenable

1. Documenter
   - ✓ permet de se poser les bonnes questions,
   - ✓ aide la relecture/maintenance.

2. Utiliser une algorithmie simple
   - ✓ plus facile à relire par quelqu'un de moins "brillant",
   - ✓ plus facile à comprendre pour le compilateur,

     code final plus performant ?
   - ✓ facilite l'application d'outil de détection de bugs,
   - x dans certains cas on est obligé de complexifier.

3. Tester
   - x chronophage,
   - ✓ des tests précoces permettent la détection de bugs au plus tôt.

Exemple 1.

```
a += (b > 3);
```

En C :

- false $= 0$,
- true $\neq 0$.

Le compilateur “a le droit” de traduire par

```
a += (b - 3);
```

Exemple 2.

```
/* todo: check operator priorities */
a += (3&b>>2);
```

Autres bonnes pratiques
Outils utilisés (utilisables) en pratique

- Règles de codage : MISRA
- Base de données de vulnérabilités : CVE/CWE
- Tests intensifs :
  - tests unitaires,
  - fuzzing,
  - couverture de code
  - génération automatique de tests (par modélisation)
- Analyse statique : Frama-C, Klockwork, Polyspace, . . .
- Détection d'erreurs à l'exécution.

Remarque sur "goto fail"

Aurait été évité par de l'analyse statique car du code mort aurait été détecté.

Plan

Besoin : diminuer le traffic TLS
Solution : compresser les données

1. Compresser les données chiffrées.
2. Compresser les données claires.

CRIME (BREACH est du même genre)

Donnée claire cible : cookie de session (contenant une donnée sensible).
Corrompre le navigateur permet de :

- de modifier une partie du cookie,
- effectuer des requêtes de chiffrement à l'insu de l'utilisateur.

La taille des messages compressés permet d'identifier les octets présents dans la donnée.

Version rollback :

Alice
(supporte TLS $\rightarrow$ 1.3)

Bob
(supporte TLS $\rightarrow$ 1.2)

Version rollback :

Alice
(supporte TLS $\rightarrow$ 1.3)

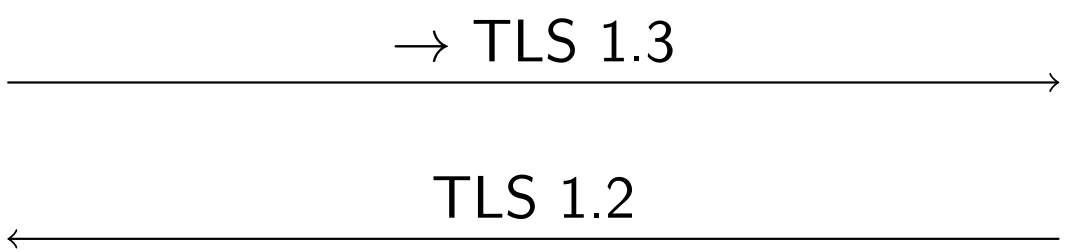
Bob
(supporte TLS $\rightarrow$ 1.2)

$\rightarrow$ TLS 1.3 $\longrightarrow$

Version rollback :

Alice
(supporte TLS $\rightarrow$ 1.3)

Bob
(supporte TLS $\rightarrow$ 1.2)



Version rollback :

Alice	Oscar	Bob
(supporte TLS $\rightarrow$ 1.3)	*(attaquant)*	*(supporte TLS $\rightarrow$ 1.2)*

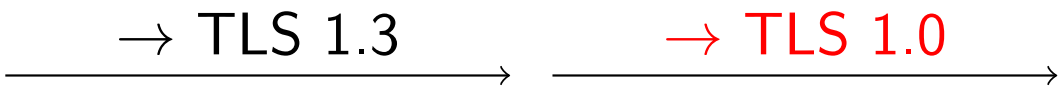
Version rollback :

Alice	Oscar	Bob
(supporte TLS $\rightarrow$ 1.3)	*(attaquant)*	*(supporte TLS $\rightarrow$ 1.2)*

$\xrightarrow{\rightarrow \text{ TLS 1.3}}$

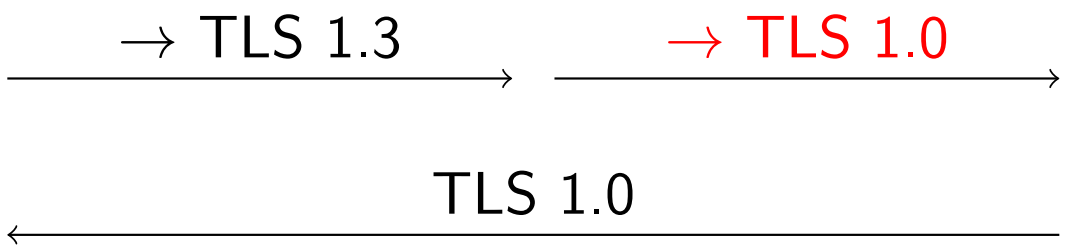
Version rollback :

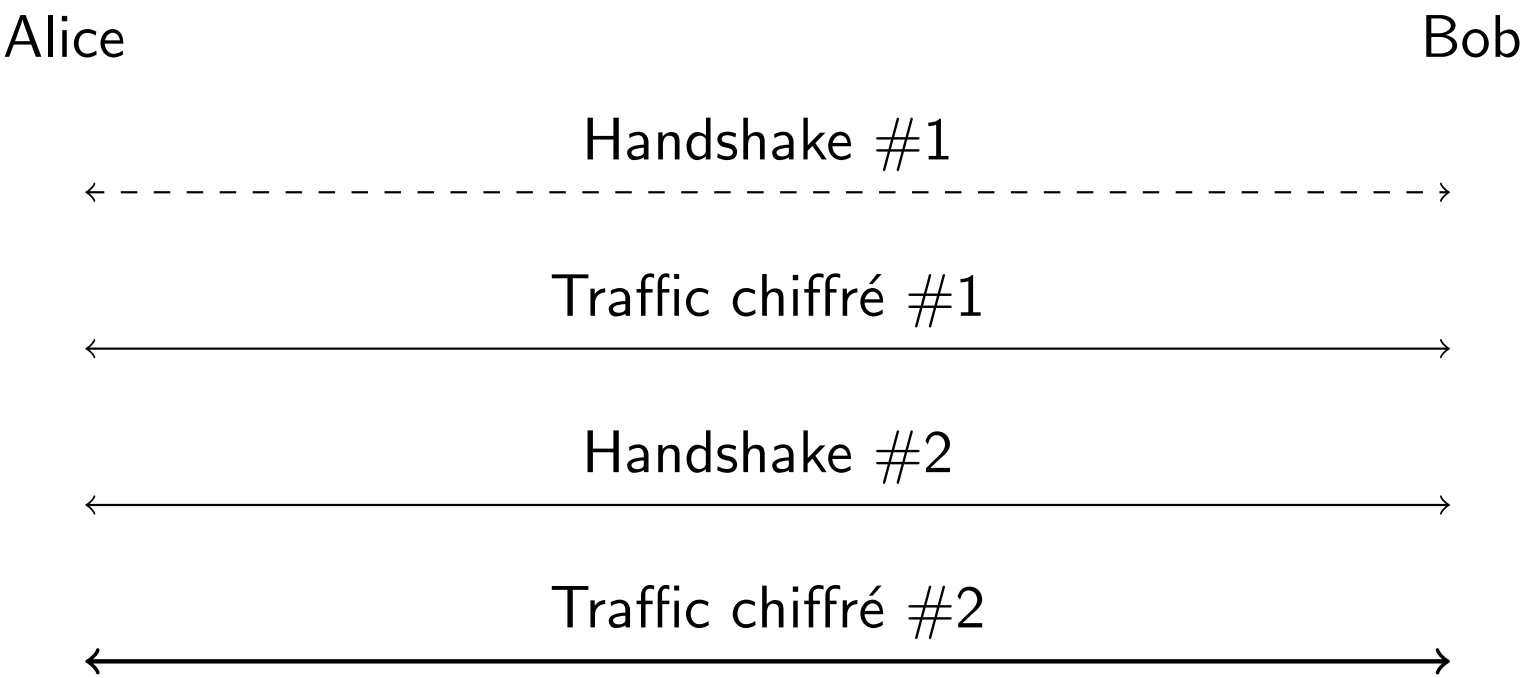
Alice	Oscar	Bob
(supporte TLS → 1.3)	*(attaquant)*	*(supporte TLS → 1.2)*

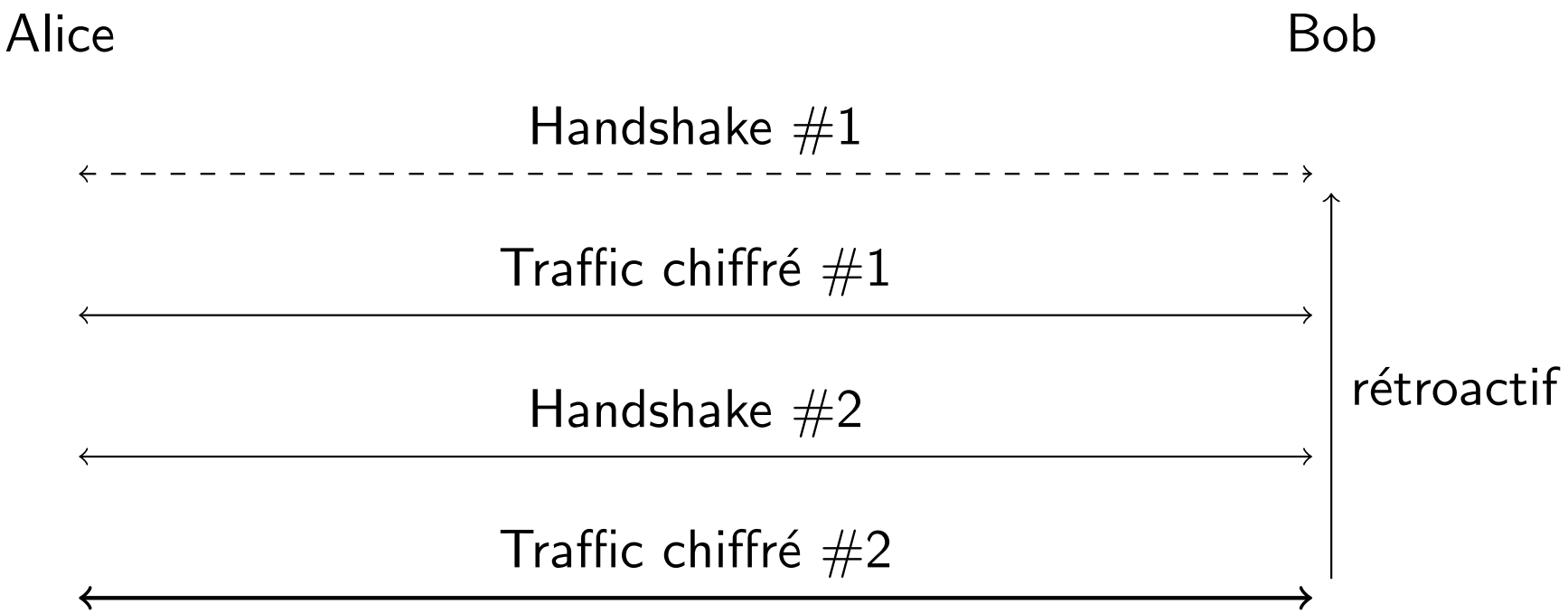


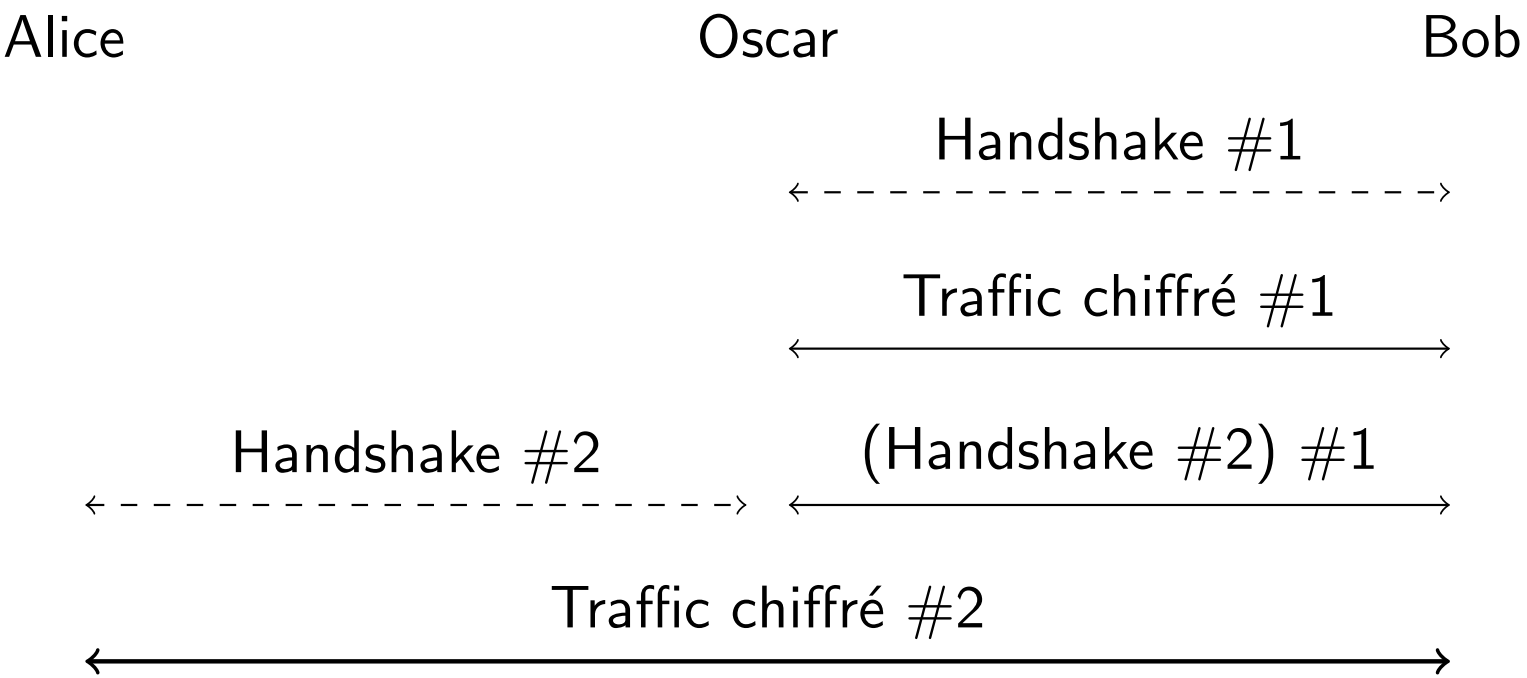
Version rollback :

Alice	Oscar	Bob
(supporte TLS → 1.3)	*(attaquant)*	*(supporte TLS → 1.2)*



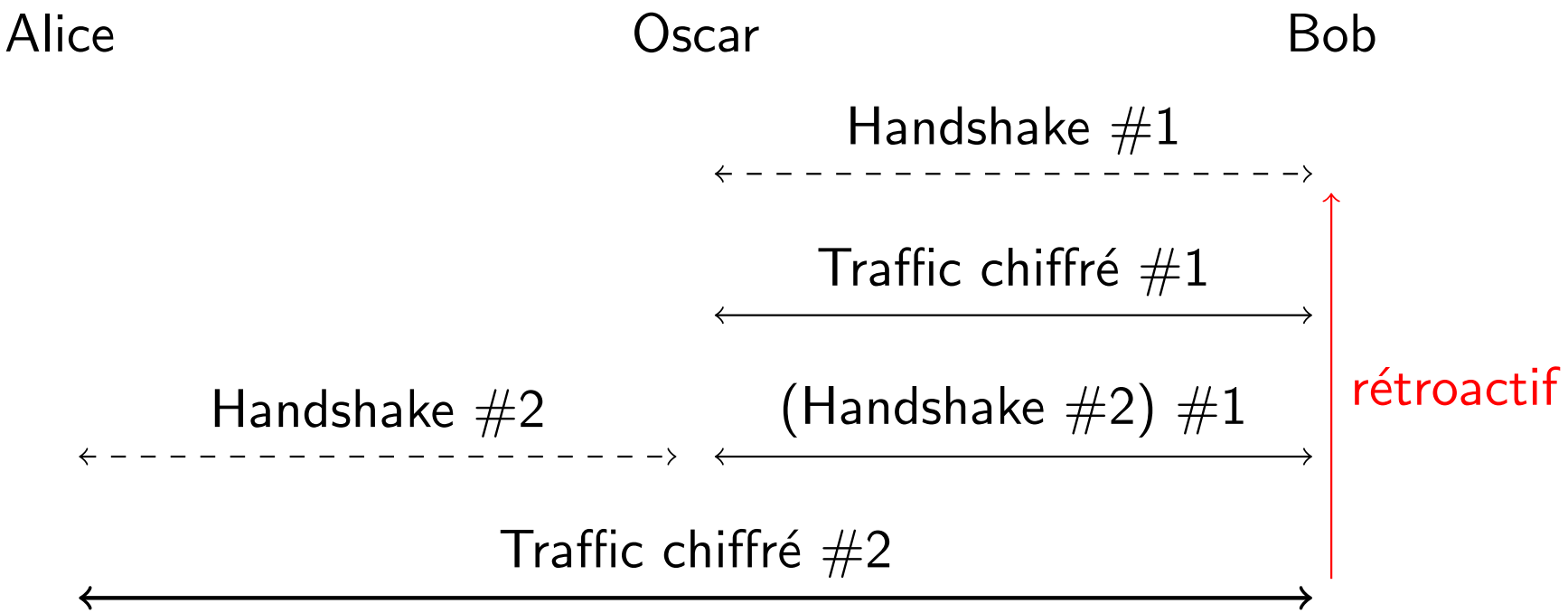

Alice
Bob
Handshake #1
Traffic chiffré #1
Handshake #2
Traffic chiffré #2


Alice
Bob
Handshake #1
Traffic chiffré #1
Handshake #2
Traffic chiffré #2
rétroactif


Alice
Oscar
Bob
Handshake #1
Traffic chiffré #1
Handshake #2
(Handshake #2) #1
Traffic chiffré #2

Impact des fonctionnalités non cryptographiques

Renégociation TLS : attaque



Plan

Trop de choix

Exemple fonction `doRSA(mode,padding,...)`

Mode	Padding	Sécurité
Chiffrement	rien	chiffrement malléable
Chiffrement	pkcs1v15	Attaque de Bleichenbacher
Chiffrement	oaep	ok
Chiffrement	pss	PSS prévu pour la signature
Signature	rien	signature malléable
Signature	pkcs1v15	dépend des applications
Signature	oaep	dépend des applications
Signature	pss	ok

Efficacité contre sécurité

- Cryptocat : réduction de la taille de la clef.
- HIVE : utilisation de RC4 pour générer de l'aléa.

Favoriser l'utilisation des fonctions les plus sécurisées.

Toujours prendre la version sécurisée sauf si

- besoin avéré d'efficacité,
- analyse de sécurité ok.

Exemple

```
void dolt();
void dolt_secure();
```

devient

```
void dolt_unsecure();
void dolt();
```

- Suppression d'effacement

```
char passwd[20];
...
memset(passwd,0,20);
return OK;
```

- Suppression de sécurité anti-déroutage

```
void function(int secure, ...) {
  if ( secure != 0x1234 )
    return ERROR;
  ...
  if ( secure != 0x1234 )
    return ERROR;
}
```

Mauvaise utilisation des outils

Compilateur et attaques physiques

- Mutualisation de calculs
  - attaques en fautes.
- Accélérations algorithmiques
  - attaques en fautes (RSA-CRT),
  - timings attacks.
- Accélération matérielle
  - timing attacks (utilisation de cache).

Ces derniers points posent plus de problèmes techniques.

À retenir

Messages

- Faire les choses le plus simplement et clairement.
- Prendre le temps de réfléchir et de préciser les détails.
- Ne pas supposer de compétence particulière chez l'utilisateur.

Bonnes pratiques

- Documenter (valeurs attendues, justification des choix).
- Tester (les paramètres, les retours de fonction, le code).
- Tester encore (les cas limites, les cas d'erreur . . .).
- Ne pas privilégier la performance à la sécurité par défaut !