Cryptographie : attaques tous azimuts

Jean-Baptiste Bédrune
Sogeti/ESEC - Capgemini

Éric Filiol
ESAT

Frédéric Raynal
Sogeti/ESEC - Capgemini – MISC magazine

Introduction

Attaques opérationnelles sur la cryptographie
Méthodologie d'un attaquant avec peu de moyens :

- Recherche d'un maximum de vecteurs d'attaque
- Obtention d'un maximum d'informations
- Pas de cryptanalyse intensive

Roadmap

Roadmap

Audit d'une application web

Contexte

- Gestion salaires, fiches de paie, etc.
- Client lourd java
- Étude du chiffrement du *token* d'authentification

Architecture

- Deux classes `RSAEncoder` et `RSADecoder`
- Héritent de `RSACoder`

Contenu de `RSACoder`

- Structure du *token* : `login`, `pwd`, `sessionID`, `tstamp`
- Routines chiffrement / déchiffrement RSA
- 4 tableaux d'octets : clés de chiffrement

Vulnérabilité

Analyse des clés

- *Dump* avec `openssl`
- Deux clés publiques (512 et 1024 bits)
- Deux clés privées associées (PKCS#8) $\Rightarrow$ déchiffrement immédiat

Exemple : clé de 512 bits

```
$ openssl pkcs8 -inform DER -in priv512.der -nocrypt -out rsa512.pem
$ openssl rsa -in rsa512.pem -inform PEM -text -noout
Private-Key: (512 bit)
modulus:
    00:9d:f7:70:d4:61:84:ad:8d:fb:64:2f:d5:e9:97:
    ...
privateExponent:
    0e:0c:c5:0f:4e:c1:23:42:9e:9a:71:9a:c0:14:fc:
    ...
```

Roadmap

Des clés dans la pomme

Mac OS X : encore des mots de passe en clair

- 2004 : alerte de sécurité car les mots de passe sont en clair dans le fichier de *swap*
- 2008 : en cherchant un peu, des mots de passe toujours en clair :
  - dans le processus d'authentification `loginwindow.app`
  - dans le fichier d'hibernation `/var/vm/sleepimage`
  - rem : nécessite les droits root

Dumper les mots de passe

Authentification : loginwindow.app

```
>> strings -a 103-malloc-01800000-01826000.raw|grep -A3 longname
longname
Jean-Kevin LeBoulet
password
 jeankevin666
```

Hibernation : /var/vm/sleepimage

```
>> sudo strings -8 /var/vm/sleepimage |grep -A3 longname
longname
JeanKevLeB
password
 jeankevin666
```

Pour résumer

Au-delà des simples attaques

- Trappes : affaiblissement volontaire de l'entropie ou de l'algorithme, . . .
- Génération d'aléa : mauvaise initialisation, générateurs faibles, générateurs forts biaisés . . .

Problèmes

- Nécessité de comprendre les protocoles / formats utilisés
  - Ex. : communications / stockages chiffrés,

$\Rightarrow$ Comment identifier les fonctions cryptographiques utilisées :

- dans des fichiers ?
- dans des flux de communications ?

Roadmap

1. Des failles triviales . . . mais malheureusement réelles
2. Automatisation de l'analyse dans les binaires
   - État de l'art
   - Identification par analyse statique
   - Identification par analyse dynamique
3. Ce à quoi vous avez échappé

Roadmap

1. Des failles triviales . . . mais malheureusement réelles
2. Automatisation de l'analyse dans les binaires
   - État de l'art
   - Identification par analyse statique
   - Identification par analyse dynamique
3. Ce à quoi vous avez échappé

Existant : analyse statique

Motivations

Détection manuelle fastidieuse

Parties crypto = enchaînement de primitives

Patterns et structures caractéristiques pour :

- les fonctions de hachage
- les fonctions de chiffrement par blocs / flot

Outils existants

Recherche de *patterns* uniquement

- `KANAL`, *plugin* pour PEiD
- `findcrypt`, *plugin* IDA
- `searchcrypt`, *plugin* ImmDbg

Identification des fonctions cryptographiques

Utilisation de constantes dans le code :

```
mov     dword ptr [esi], 67452301h
mov     dword ptr [esi+4], 0EFCDAB89h
mov     dword ptr [esi+8], 98BADCFEh
mov     dword ptr [esi+0Ch], 10325476h
mov     dword ptr [esi+10h], 0C3D2E1F0h
```

Utilisation de tables de constantes :

```
mov     dl, ds:rc2_PITABLE[edx]

rc2_PITABLE db 0D9h, 78h, 0F9h, 0C4h, 19h, 0DDh, 0B5h, 0EDh,
            db 28h, 0E9h, 0FDh, 79h, 4Ah, 0A0h, 0D8h, 9Dh,
            db 0C6h, 7Eh, 37h, 83h, 2Bh, 76h, 53h, 8Eh,
            db 62h, 4Ch, 64h, 88h, 44h, 8Bh, 0FBh, 0A2h
            ...
```

État de l'art : pour résumer

Bilan des outils publics existants

- Recherche des *patterns*
- Association avec le nom de l'algorithme
- Énumération des adresses et des références aux *patterns*

KANAL

```
MD4 :: 00008225 :: 77DA8E25
    The reference is above.
MD5 :: 00016CB6 :: 77DB78B6
    The reference is above.
SHA1 [Compress] :: 00013E82 :: 77DB4A82
    The reference is above.
DES [long] :: 0001DF98 :: 77DBEB98
    Referenced at 77DBE39F
    Referenced at 77DBE41A
    Referenced at 77DBE495
...
```

Comment aller plus loin ?

Pourquoi se limiter à l'identification d'un algorithme ?

- Identifier toutes les fonctions à usage cryptographique dans un binaire
- Reconnaître les primitives, leur mode opératoire, etc.

$\Rightarrow$ Écriture d'un *plugin* pour IDA

Étape 1 : analyse statique pour repérer les fonctions cryptographiques
Étape 2 : analyse dynamique pour identifier quelle crypto est mise en œuvre par les fonctions repérées précédemment

Roadmap

1. Des failles triviales . . . mais malheureusement réelles
2. Automatisation de l'analyse dans les binaires
   - État de l'art
   - Identification par analyse statique
   - Identification par analyse dynamique
3. Ce à quoi vous avez échappé

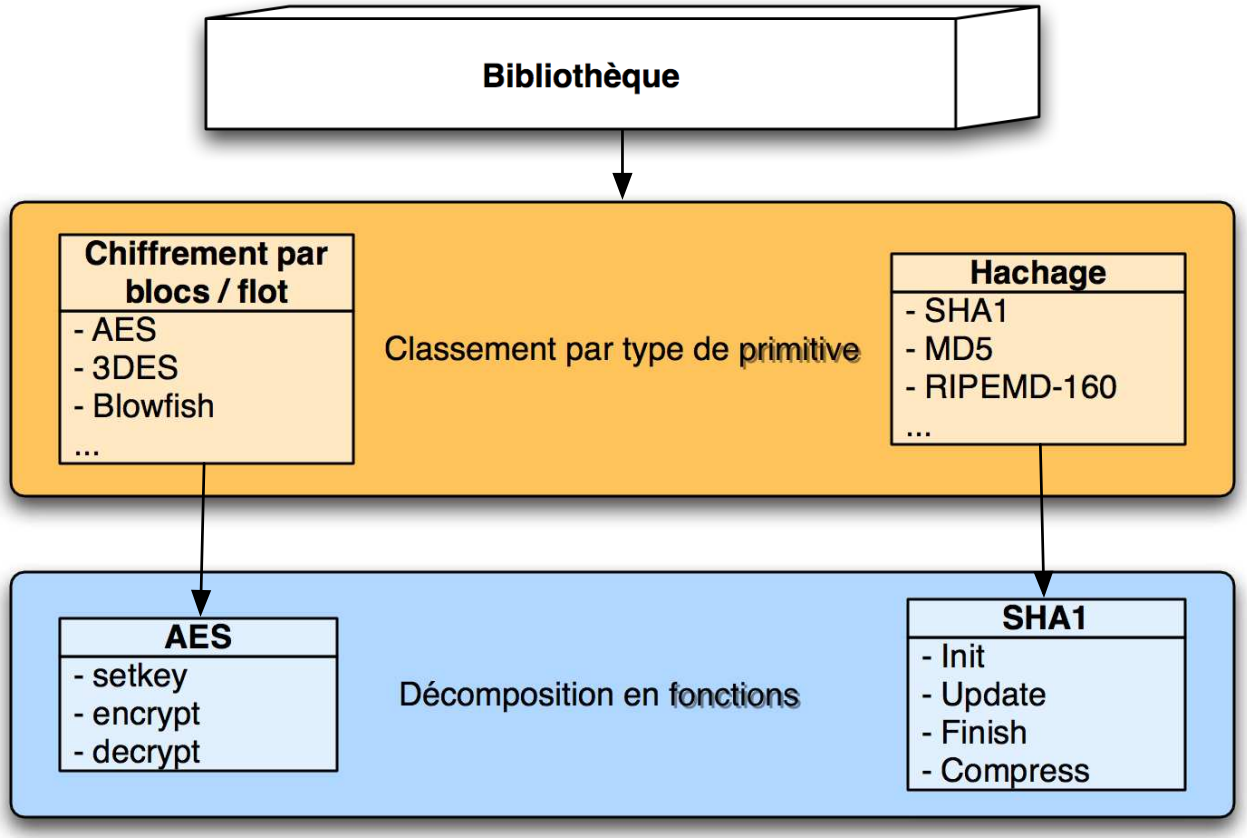
Algorithme d'identification : automatisation

Comment ça marche ?

- Reconnaissance des constantes et des tables
- Reconstruction des dépendances (fonctions appelantes)
- Reconstruction des structures génériques (API haut niveau)
- Redescente à partir des structures identifiées

$\Rightarrow$ Identification des fonctions cryptographiques et de leur rôle

Structure d'une bibliothèque



Algorithme d'identification : les constantes

Travail préparatoire

- Décomposer les protocoles en primitives
  - Échange de clés, partage de secrets, etc.
- Décomposer les primitives en blocs élémentaires
  - Hachage : `init`, `update`, `finish`, `compress`
  - Chiffrement : `setkey`, `encrypt`, `decrypt`

Dans le *plugin*

- Recherche des constantes / *patterns* classiques
- Identification des fonctions et de leur rôle par rapport à ces valeurs
- Problème : une même constante peut servir dans différents blocs élémentaires

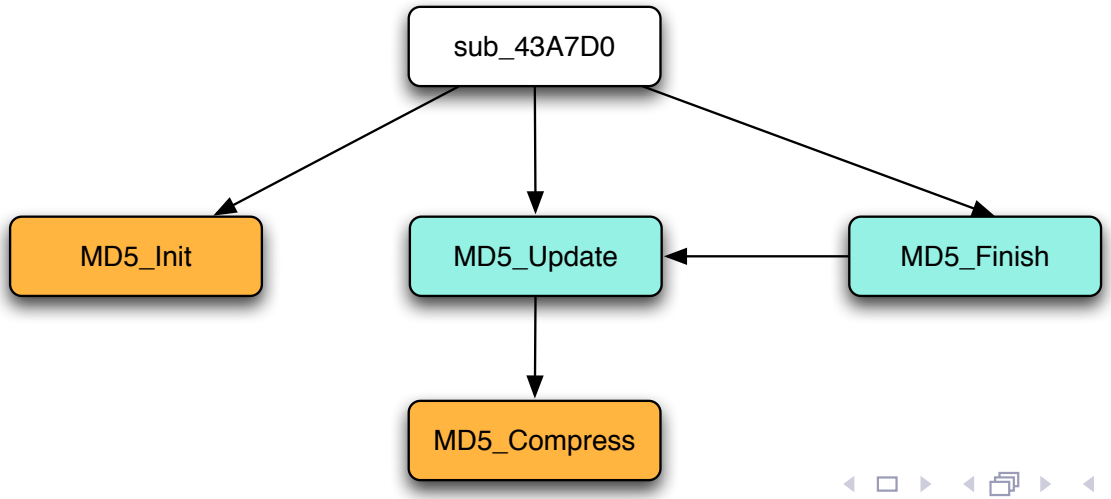
$\Rightarrow$ Nécessité de déterminer le rôle de la fonction

Algorithme d'identification : les dépendances

Isomorphisme de graphes

- Identifier les blocs élémentaires grâce aux constantes
  - Indique le rôle de certaines fonctions
- Recherche des dépendances entre ces blocs élémentaires
  - Reconnaissance de fonctions appelantes au rôle inconnu
- Isomorphisme entre le graphe théorique et celui reconstruit

$\Rightarrow$ Identification de nouvelles fonctions et de leur rôle



Algorithme d'identification : le haut niveau

API utilisateur

- Les bibliothèques utilisent une API générique pour les primitives
  - Ex. : structure commune pour tous les chiffrements symétriques
- Présence d'une structure de description pour chaque primitive

$\Rightarrow$ Identification des fonctions restantes et des modes opératoires

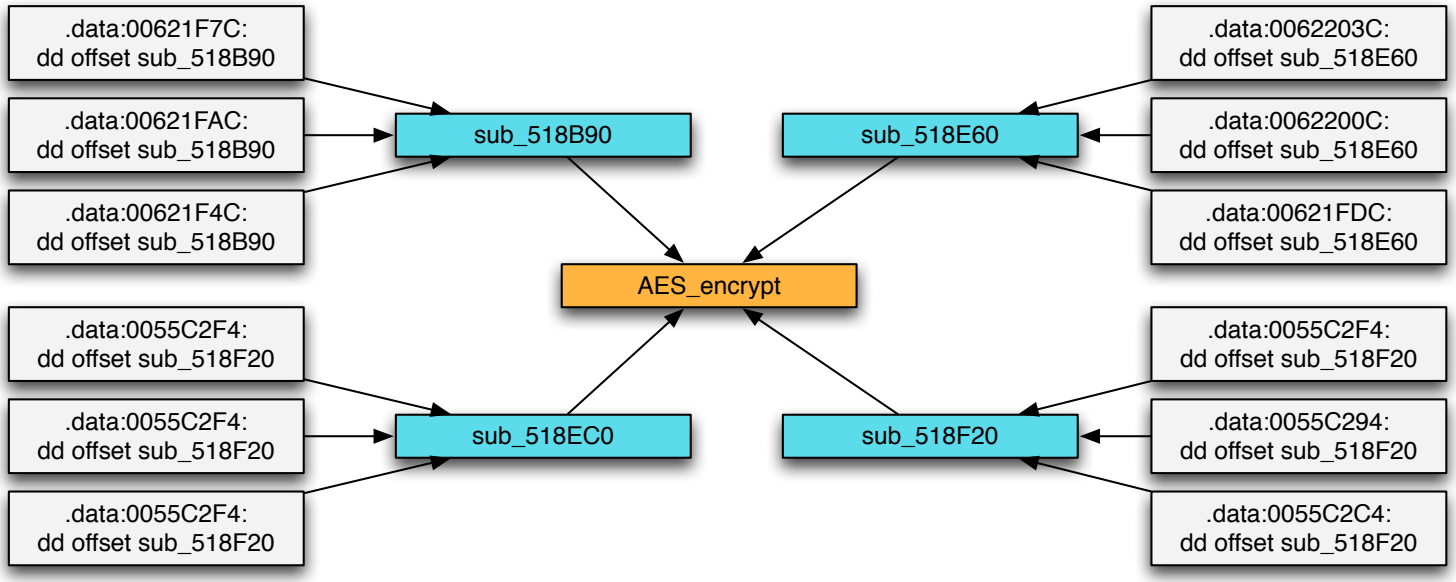
```
struct evp_cipher_st {
  int nid;
  int block_size;
  int key_len;
  int iv_len;
  int (*init)();
  int (*do_cipher)();
  ...}
```

```
dword_55C2E0
  dd 96h ; nid
  dd 10h ; block_size
  dd 20h ; key_len
  dd 10h ; iv_len
  dd offset AES128_setkey
  dd offset AES_CBC_encrypt
  ...
```

Exemple complet sur AES

Reconstruire AES

- Reconnaissance par constante de `AES_encrypt`
- Parcours en profondeur sur les parents de la fonction de chiffrement
  - 3 longueurs de clé possibles
  - 4 modes opératoires associés



Roadmap

Monitoring des appels de fonction

Mise en œuvre

- Exécution du programme
- *Hook* de toutes les fonctions détectées précédemment
- *Dump* et interprétation de chacun des arguments
  - en entrée : données à chiffrer, clés de chiffrement, vecteur d'initialisation, etc.
  - en sortie : données chiffrées, condensat, etc.

Possibilités

Difficulté

- Déterminer le rôle de chaque argument à l'exécution
- Identifier précisément chaque fonction, pour connaître la taille des arguments à *dumper*
- Ne rater aucune fonction importante, sinon la sortie est incompréhensible

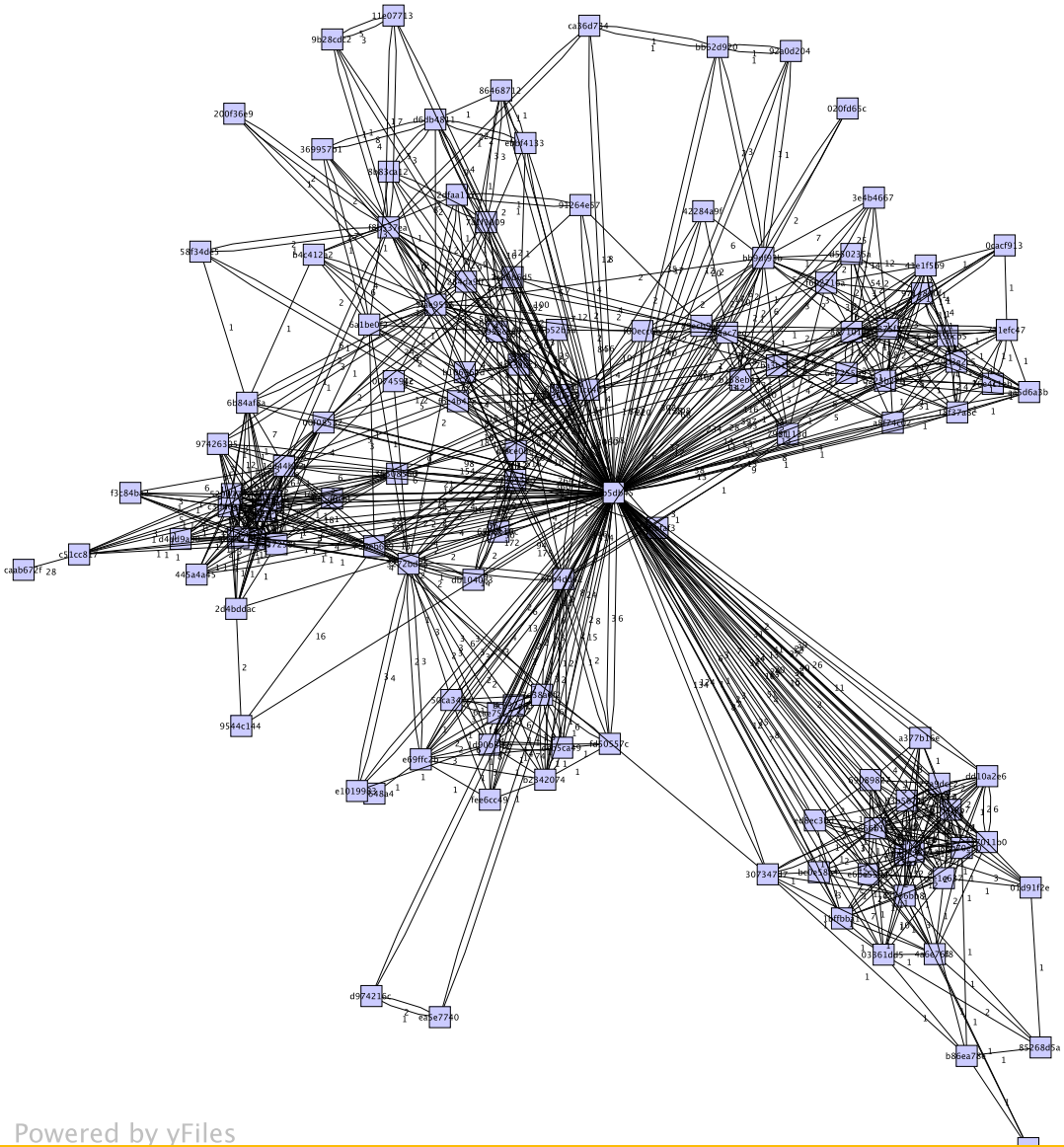
Avantages

- Obtenir davantage d'informations sur les fonctions (ex : bruteforce sur le mode opératoire)
- Grande majorité de l'analyse automatisée

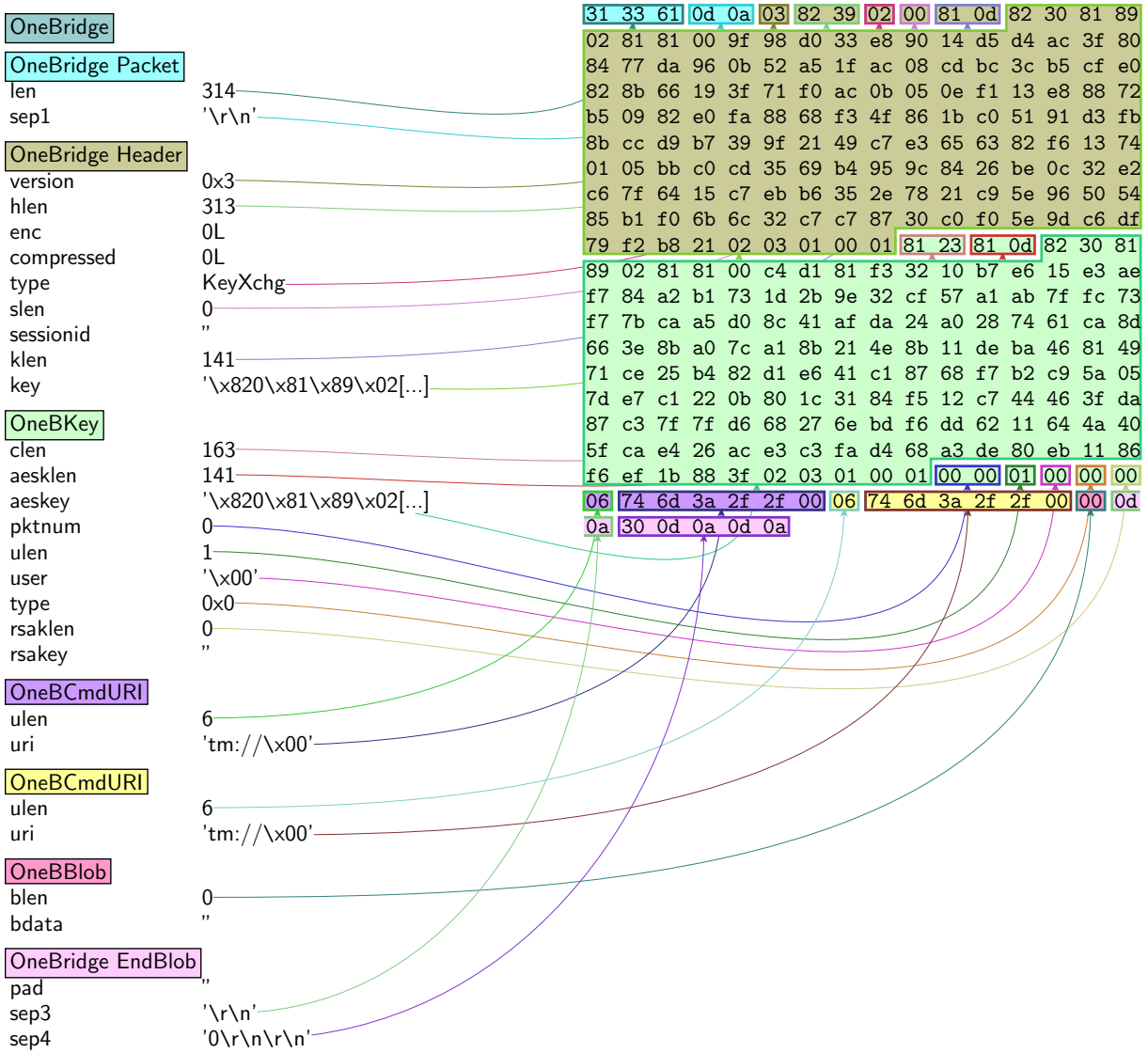
Démo . . . avec ou sans effet

Roadmap

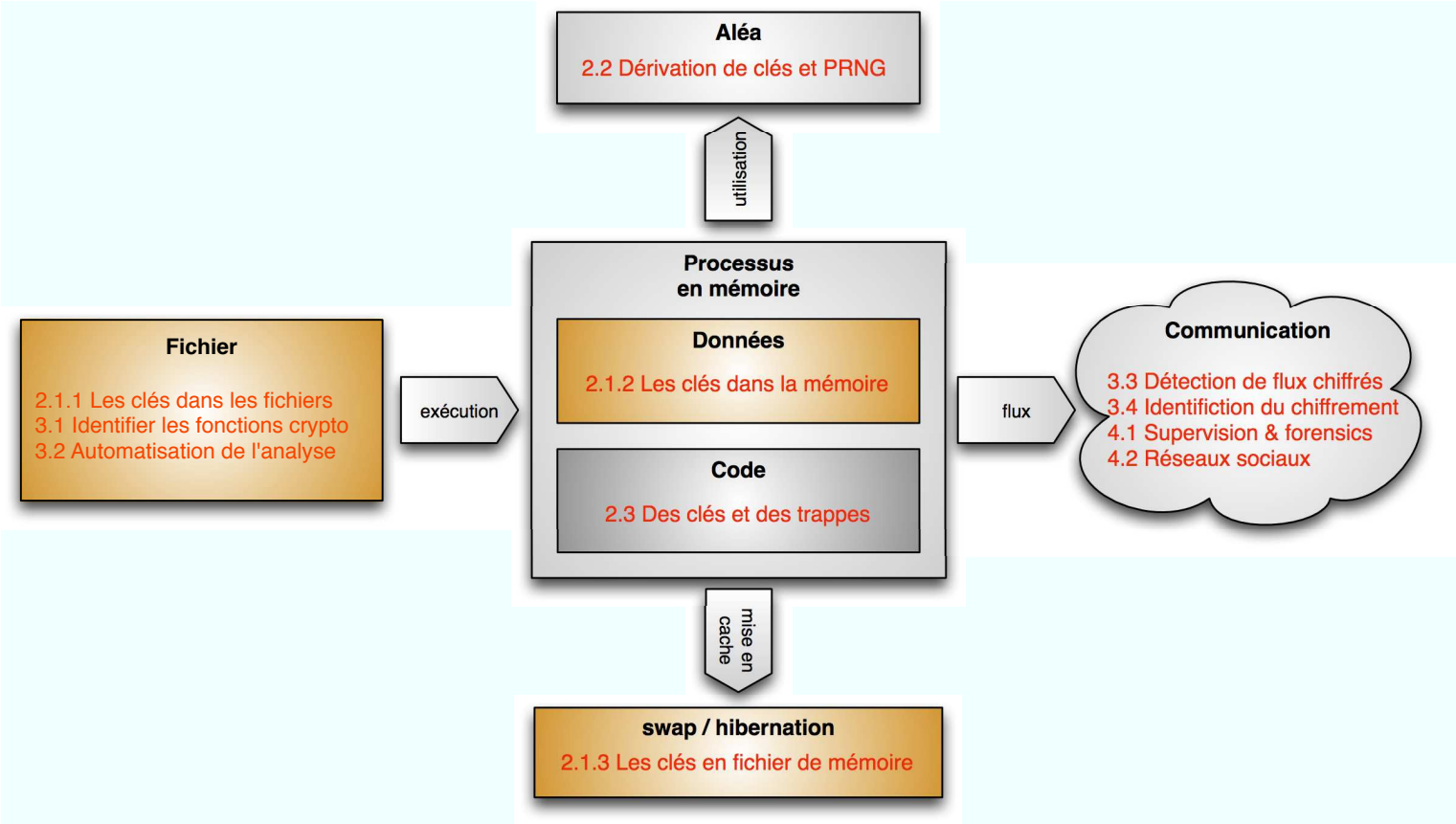
Attaques sur les flux



Exemple : la reconstruction d'un protocole fermé



Conclusion


Aléa
2.2 Dérivation de clés et PRNG
utilisation
Processus
en mémoire
Données
2.1.2 Les clés dans la mémoire
Code
2.3 Des clés et des trappes
Fichier
2.1.1 Les clés dans les fichiers
3.1 Identifier les fonctions crypto
3.2 Automatisation de l'analyse
exécution
flux
Communication
3.3 Détection de flux chiffrés
3.4 Identifiction du chiffrement
4.1 Supervision & forensics
4.2 Réseaux sociaux
mise en cache
swap / hibernation
2.1.3 Les clés en fichier de mémoire

Questions et (peut-être) réponses

- Merci de votre attention
- Réveillez vos voisins