Analyse de binaire

Désassemblage, outils d’analyse
Protections

Point de vue développeur

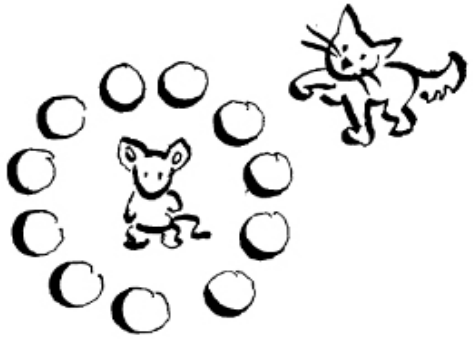
Ce qu'on a vu

Construire du code sûr :

- $\Rightarrow$ Détecter les vulnérabilités du code
- $\Rightarrow$ Utiliser les vérifications/protections du compilateur
- $\Rightarrow$ Protéger la plate-forme

Points de vue Auditeur/attaquant

- Comprendre, analyser du code qu'on ne connait pas (souvent le binaire)
  - Contient-il des vulnérabilités ?
  - Est-ce un code malicieux ?



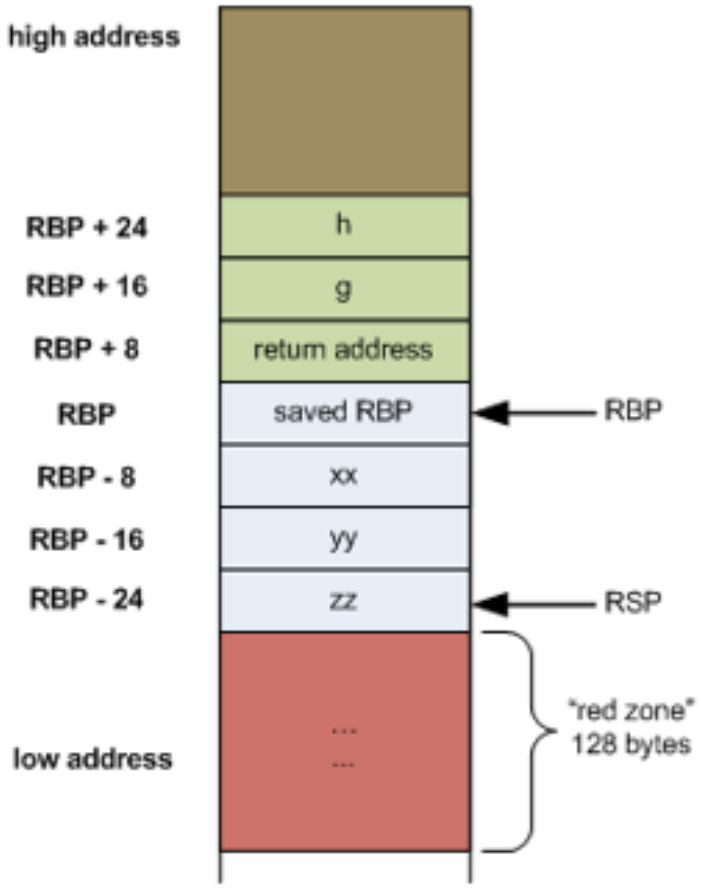
- Attaquer du code
  - Trouver des faiblesses
  - Mettre en place un exploit

  Ex ./bof_64 shellcode padding @shellcode

Analyse de binaire

- On n’a pas toujours le source
- L’exploitabilité dépend d’information bas niveau (layout mémoire)
- Le compilateur peut modifier le code (optimisations et protections)
- Le code peut être offusqué

Pile et registres x 86



RSP : pointeur de pile
RSP : pointeur de base
RIP : pointeur d'instruction

en 32 bits : ESP, EBP, EIP
en 16 bits : SP, BP, IP

Les difficultés pour l'auditeur/attaquant

- Comprendre le code :
  - Flot de données
  - Flot de contrôle (CFG)
  - Impact des entrées contrôlables
    - Analyse de teinte (dépendance)
    - Exploitabilité

En statique sur tout le programme / sur un ensemble de traces d'exécution

Reverse, protections

- Les outils
- Les algorithmes de désassemblage
- Protection contre le reverse

Les différentes classes d'outils

- dynamique : basé sur l'exécution d'une trace (avec/sans instrumentation). Ex : gdb

- Statique : sans exécution. Visualisation du code : désassemblage, graphe de flot de contrôle, de flot de données. Ex : objdump, ida, ghidra

=> dépend aussi de quoi on dispose : accès au code (boite blanche), à distance (boite noire)

Analyse dynamique

- Les + :
  - Instrumentation/inspection
  - Exécution fournissant des valeurs exactes

- Les - :
  - On peut ne pas voir certaines exécutions (malicieuses)
  - Demande d'exécuter

➔ Pintools, debugger, …, (smart) fuzzers

Analyse statique

- Les + :
  - Non invasif (n'exécute pas du code potentiellement malveillant)
  - Décrit toutes les exécutions
  - S'adapte à toute architecture

- Les - :
  - Peut beaucoup surapproximer
  - Code offusqué

=> Outils d'analyse de binaire (à la Frama-C sur du bas niveau)

strcmp

Le prototype, suivant la norme ISO/CEI 9899:1999, est le suivant :

```
int strcmp(const char *s1, const char *s2);
```

strcmp compare lexicalement les deux chaînes caractère par caractère et renvoie 0 si les deux chaînes sont égales, un nombre positif si s1 est lexicographiquement supérieure à s2, et un nombre négatif si s1 est lexicographiquement inférieure à s2.

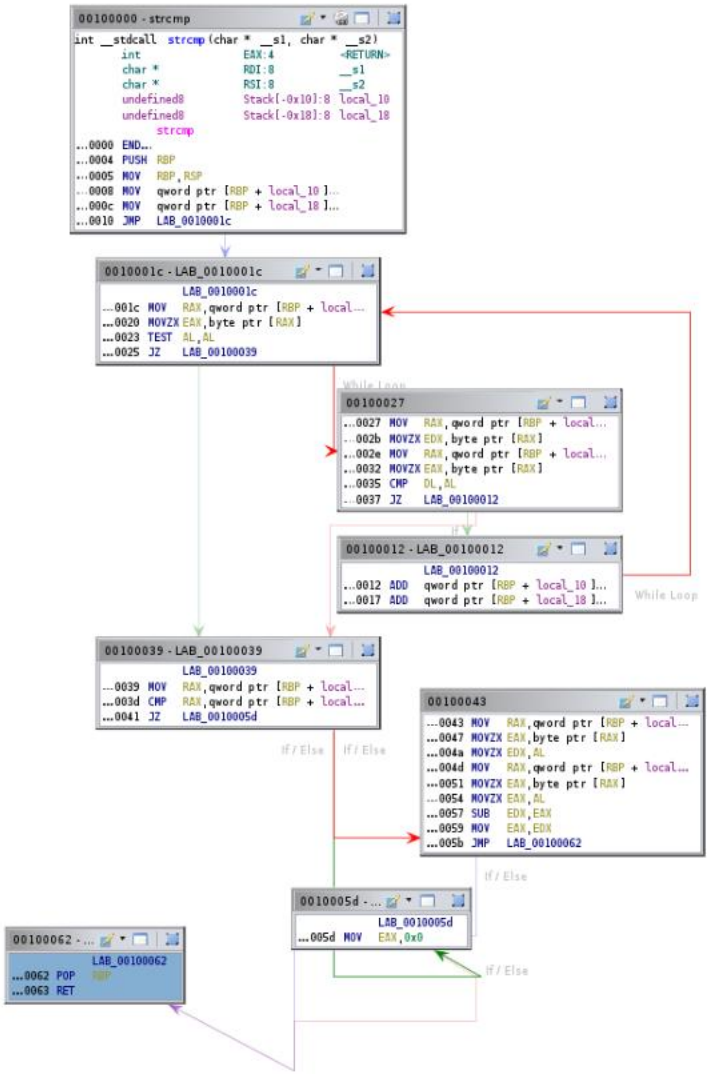
```
int strcmp (const char* s1, const char* s2)
{
    while (*s1 != '\0' && (*s1 == *s2)) {s1++; s2++;}
    return (s1==s2) ? 0 : (*(unsigned char *)s1 - *(unsigned char *)s2);
}
```

```
00100000 f3 0f 1e fa     ENDBR64
00100004 55              PUSH        RBP
00100005 48 89 e5        MOV         RBP,RSP
00100008 48 89 7d f8     MOV         qword ptr [RBP + local_10],__s1
0010000c 48 89 75 f0     MOV         qword ptr [RBP + local_18],__s2
00100010 eb 0a           JMP         LAB_0010001c

                     LAB_00100012
00100012 48 83 45        ADD         qword ptr [RBP + local_10],0x1
         f8 01
00100017 48 83 45        ADD         qword ptr [RBP + local_18],0x1
         f0 01

                     LAB_0010001c
0010001c 48 8b 45 f8     MOV         RAX,qword ptr [RBP + local_10]
00100020 0f b6 00        MOVZX       EAX,byte ptr [RAX]
00100023 84 c0           TEST        AL,AL
00100025 74 12           JZ          LAB_00100039
00100027 48 8b 45 f8     MOV         RAX,qword ptr [RBP + local_10]
0010002b 0f b6 10        MOVZX       EDX,byte ptr [RAX]
0010002e 48 8b 45 f0     MOV         RAX,qword ptr [RBP + local_18]
00100032 0f b6 00        MOVZX       EAX,byte ptr [RAX]
00100035 38 c2           CMP         DL,AL
00100037 74 d9           JZ          LAB_00100012

                     LAB_00100039
00100039 48 8b 45 f8     MOV         RAX,qword ptr [RBP + local_10]
0010003d 48 3b 45 f0     CMP         RAX,qword ptr [RBP + local_18]
00100041 74 1a           JZ          LAB_0010005d
00100043 48 8b 45 f8     MOV         RAX,qword ptr [RBP + local_10]
00100047 0f b6 00        MOVZX       EAX,byte ptr [RAX]
0010004a 0f b6 d0        MOVZX       EDX,AL
0010004d 48 8b 45 f0     MOV         RAX,qword ptr [RBP + local_18]
00100051 0f b6 00        MOVZX       EAX,byte ptr [RAX]
00100054 0f b6 c0        MOVZX       EAX,AL
00100057 29 c2           SUB         EDX,EAX
00100059 89 d0           MOV         EAX,EDX
0010005b eb 05           JMP         LAB_00100062

                     LAB_0010005d
0010005d b8 00 00        MOV         EAX,0x0
         00 00

                     LAB_00100062
00100062 5d              POP         RBP
00100063 c3              RET
```

Code décompilé (x 86)

RBP : pointeur de frame
RSP : sommet de pile

Graphe de flot de contrôle – Ghidra



Bloc de base : suite d'instructions qui s'exécutent en séquence

Flèche : les blocs successeurs

Désassemblage/décompilation

Retrouver un code assembleur ou source à partir d'un binaire pour :

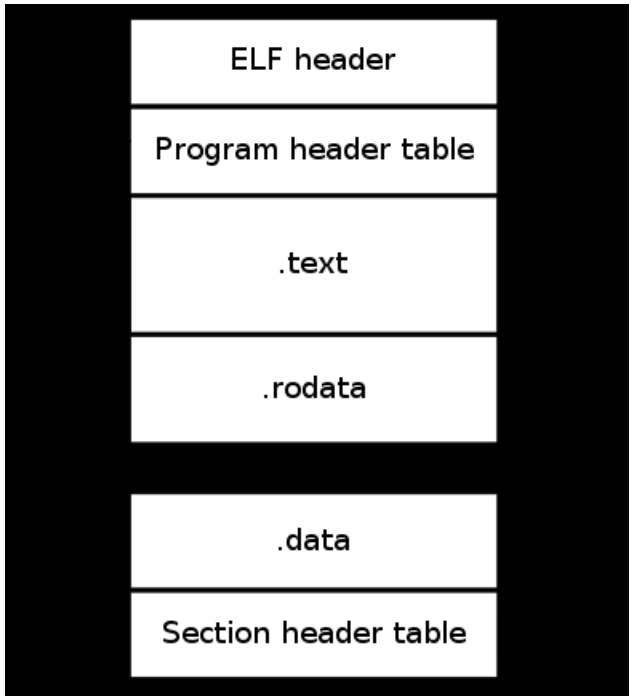
- Comprendre les fonctionnalités et l'algorithme
- Détecter des codes malicieux
- Vérifier des propriétés (pas de buffer overflow, …)

Désassembleur : du binaire à l'assembleur

Décompilateur : de l'assembleur au code source

elf format (linux)

Format d'exécutable standardisé : ELF (Linux), PE (Windows),
sections: text, initialized/unitialized data, symbol tables, relocation tables, etc.
stripped (no symbol table) vs verbose (debug info) executables . . .



.text : le code, .dat : les globales, .rodata les constantes

```
1b43:       55                      push   %rbp
1b44:       48 89 e5                mov    %rsp,%rbp
1b47:       41 54                   push   %r12
1b49:       53                      push   %rbx
1b4a:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
1b4e:       48 89 75 e0             mov    %rsi,-0x20(%rbp)
1b52:       48 89 55 d8             mov    %rdx,-0x28(%rbp)
1b56:       48 8b 5d e8             mov    -0x18(%rbp),%rbx
1b5a:       4c 8b 65 e0             mov    -0x20(%rbp),%r12
1b5e:       eb 39                   jmp    1b99 <_memcmp+0x5a>
1b60:       48 89 d8                mov    %rbx,%rax
1b63:       48 8d 58 01             lea    0x1(%rax),%rbx
1b67:       0f b6 10                movzbl (%rax),%edx
1b6a:       4c 89 e0                mov    %r12,%rax
1b6d:       4c 8d 60 01             lea    0x1(%rax),%r12
1b71:       0f b6 00                movzbl (%rax),%eax
1b74:       38 c2                   cmp    %al,%dl
1b76:       74 21                   je     1b99 <_memcmp+0x5a>
1b78:       48 8d 43 ff             lea    -0x1(%rbx),%rax
1b7c:       0f b6 10                movzbl (%rax),%edx
1b7f:       49 8d 44 24 ff          lea    -0x1(%r12),%rax
1b84:       0f b6 00                movzbl (%rax),%eax
1b87:       38 c2                   cmp    %al,%dl
1b89:       73 07                   jae    1b92 <_memcmp+0x53>
1b8b:       b8 ff ff ff ff          mov    $0xffffffff,%eax
1b90:       eb 1d                   jmp    1baf <_memcmp+0x70>
1b92:       b8 01 00 00 00          mov    $0x1,%eax
1b97:       eb 16                   jmp    1baf <_memcmp+0x70>
1b99:       48 8b 45 d8             mov    -0x28(%rbp),%rax
1b9d:       48 8d 50 ff             lea    -0x1(%rax),%rdx
1ba1:       48 89 55 d8             mov    %rdx,-0x28(%rbp)
1ba5:       48 85 c0                test   %rax,%rax
1ba8:       75 b6                   jne    1b60 <_memcmp+0x21>
1baa:       b8 00 00 00 00          mov    $0x0,%eax
1baf:       5b                      pop    %rbx
1bb0:       41 5c                   pop    %r12
1bb2:       5d                      pop    %rbp
1bb3:       c3                      retq
```

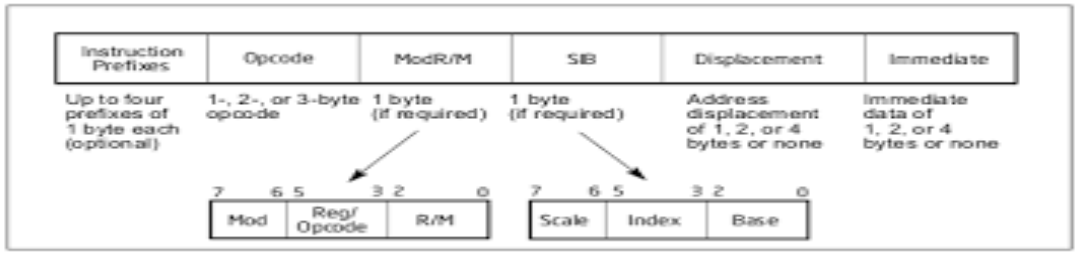


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

Désassemblage

Les difficultés :

- pas de distingo données et code
- retrouver la structuration du flot de contrôle
- Saut dynamique qui dépend d'une valeur à l'exécution (pointeur de fonction, switch, …)
- Appel et retour de fonctions (goto optimisation)
- code offusqué

$\Rightarrow$ Indécidable en général

$\Rightarrow$ Le désassemblage peut dépendre du flot de contrôle qui peut dépendre des valeurs qui peuvent dépendre du flot de contrôle …

Décompilation : retrouver le typage, les variables (exemple encodage de 2 valeurs dans un registre …)

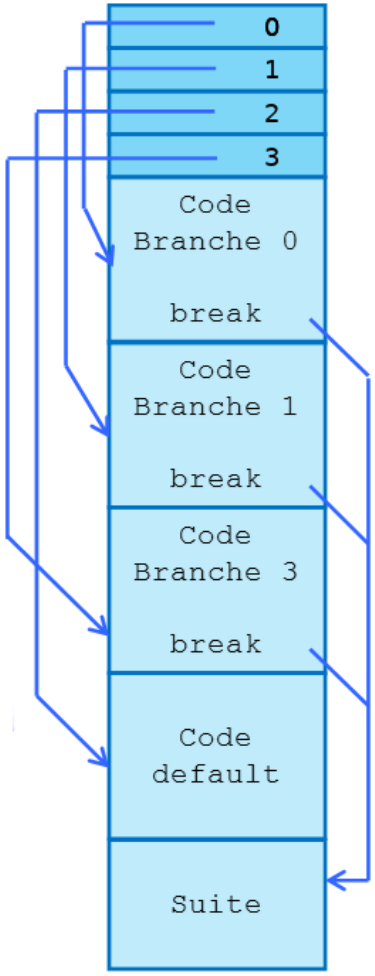
Exemple
Compilation d'un switch

switch (exp){
case v1: S1 ; break ;
case v2 : S2 ; break ;
...
case vn: Sn ; break ;
default : Sd ; break ; }

Compilé en :

- une suite de if-then-else
- si constantes consécutives une "jump-table"

(en data, dans le code)



Algorithme basique de désassemblage

⇒ Linear sweep Disassembly

1. On démarre sur le premier byte de la section code
2. On détermine le code op et en fonction de celui-ci la taille e le découpage des paramètres
3. On réitère sur le premier byte après la fin d'une instruction

\+ : on décode toute la zone code

\- : problème des données dans le code qui peuvent être décodées comme des instructions (exemple jump table)

⇒ Dans gdb, WinDbg, objdump

exemple

Lets disassemble this piece of binary code:

```
0804846c: eb04        jmp  0x804846e+4
0804846e: efbeadde    dd   0xdeadbeef   # Data hidden among instructions
08048472: a16e840408  mov  eax, [0x804846e]
08048477: 83c00a      add  eax, 0xa
```

```
0804846c: eb04        jmp   0x804846e+4
0804846e: ef          out   dx, eax
0804846f: beaddea16e  mov   esi, 0x6ea1dead
08048474: 840408      test  [eax+ecx], al
08048477: 83c00a      add   eax, 0xa
```

Algorithme de construction d'un CFG (algorithme de désassemblage récursif)

- Si instruction sans rupture de séquence lecture linaire du code
- Si instruction de branchement conditionnel : fin de bloc de base, démarrage d'un nouveau bloc en séquence, ajout de la seconde branche dans la liste des instructions a désassembler
- Si saut inconditionnel : déterminer l'adresse de branchement et ajout a la liste des instructions a désassembler

\+ : structure du code

\- : certaines parties du code peuvent ne pas être désassemblées

\- : indécidabilité (@retour, saut indirect ...)

Analyse plus fine

=> Toute analyse statique (simple) demande une analyse de valeurs. Soit le code source suivant :

```
int x, *p, y;
x = 3 ;
p = &x ;
y = *p + 4 ;   //y ne dépend pas des entrées
```

=> Code assembleur :

```
mov [ebp-4], 3 /* x=3 ; */
lea eax, [ebp-4]
mov [ebp-8], eax /* p = &x ;*/
mov eax, [ebp-8]
mov eax, [eax] /* y = *p+4 ; */
add eax, 4
mov [ebp-12], eax
```

=> Il faut tracer le contenu de l'emplacement mémoire d'adresse ebp-12

Les difficultés

- Raisonner statiquement sur du code demande :
  - de disposer de sa structure (son graphe de flot de contrôle) qui n'est pas toujours simple a construire
  - le typage étant perdu, on représente chaque valeur sous forme
  - d'un vecteur de bits, il faut donc pouvoir raisonner efficacement sur ce modèle ;
  - les objets ne sont plus identifiables par leur nom, il faut donc raisonner au niveau des adresses mémoire.
- Value Set Analysis : Travaux initiés par Reps et Balakrishnan

Technique de protection du code

- Pour la propriété intellectuelle
- Pour des malwares
- Pour protéger des données (numéro série, crypto boite blanche) ou une routine d'authentification

- Objectifs :
  - Rendre complexe le reverse
  - Rendre incompréhensible les algorithmes
  - Cacher du code malicieux

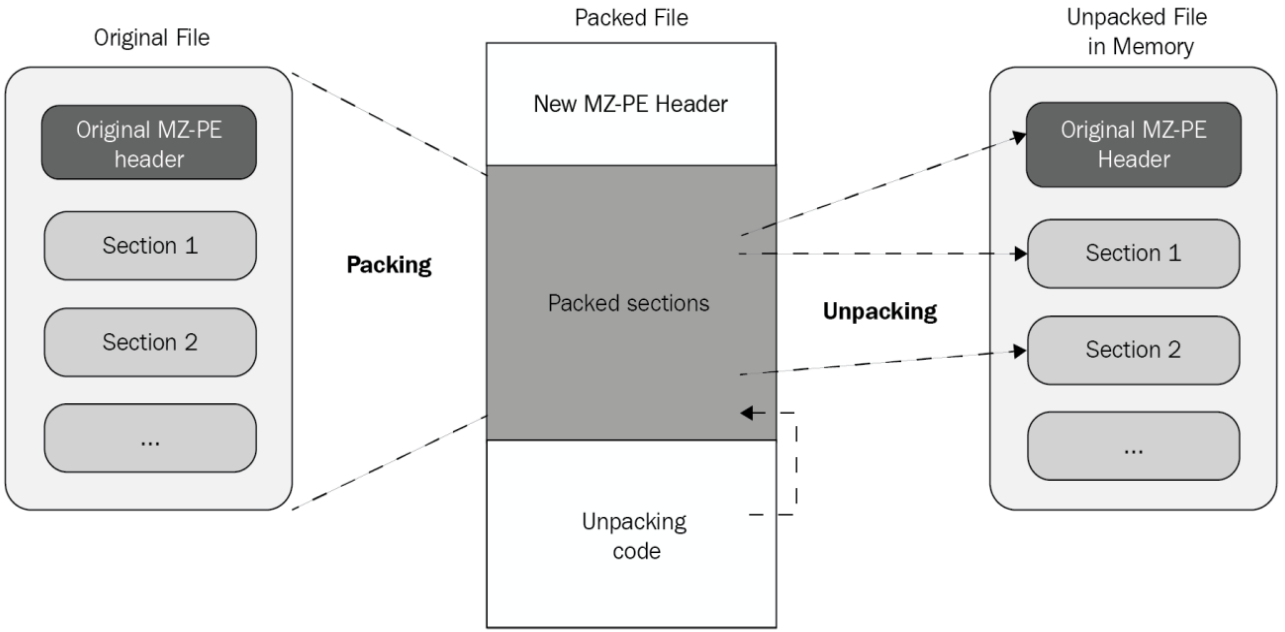
Les techniques (1)

- Détecter qu'on est sous un debugger, dans une VM (temps de calcul, …)

```
int main(){
 if (IsDebuggerPresent()){
  std::cout << "Debugger is present !!!" << std::endl;
  exit(-1);
 } else {
  std::cout << "Debugger is not present :)" << std::endl;
 }
 return 0;
}
```

Les techniques (2)

- Chiffrement du code (packer), code polymorphe

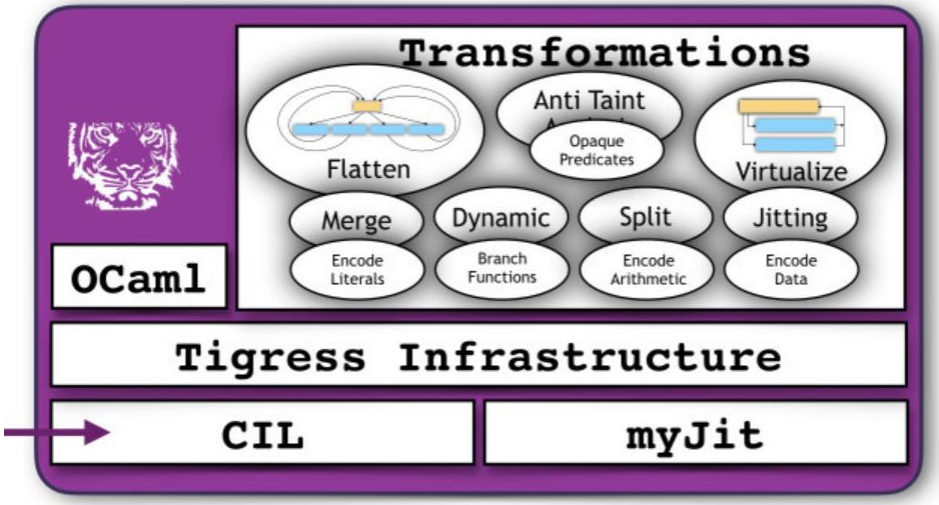


Les techniques (3)

- Rendre non compréhensible les calculs, les structures de données, le flot de contrôle …

- Virtualisation : le programme devient une entrée exécutée par un interpréteur

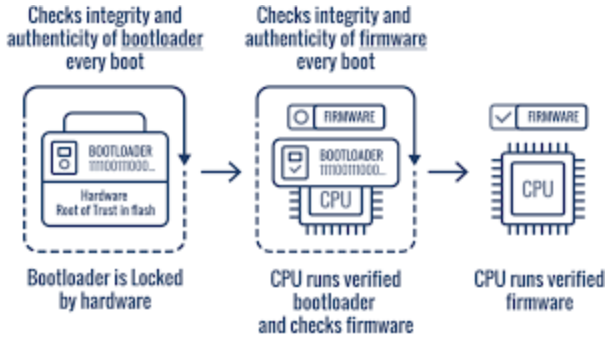
tigress.cs.arizona.edu



- TP : exemple de reverse d’un code chiffré (slides CrackmeSlides)

Code embarqué et sécurité

- Applications sensibles : secret, authentification, mise à jour à distance (secure boot, FU)



- Normes, certifications nationales (CSPN), mondiales (CC), européennes (Enisa)

VISA DE SÉCURITÉ

AGENCE NATIONALE DE LA SÉCURITÉ DES SYSTÈMES D'INFORMATION
ANSSI

http://iotworm.eyalro.net/

- Attaque sur les ampoules connectées : un drone faisant clignoter tout un bâtiment (DOS)
- Zigbee : un protocole de proximité à bas coût

Les étapes

- Etape 1 : découvrir et exploiter un bug dans l'implémentation du protocole
- Etape 2 : une attaque permettant de trouver la clé AES permettant d'encrypter et authentifier un nouveau firmware.
- Etape 3 : construire une « mise à jour » du logiciel (over the air) qui exploite la faille
- Résultat : un déni de service
- Correction : correction du bug, meilleure protection des clés

Recommandations

https://www.ssi.gouv.fr/uploads/2021/03/anssi-guide-selection_crypto-1.0.pdf

- Utiliser la cryptographie à l'état de l'art
- Ne pas réimplémenter des algorithmes de crypto mais utiliser des bibliothèques éprouvées

2.2.5 Utiliser des bibliothèques éprouvées

L'implémentation logicielle de mécanismes cryptographiques est une tâche délicate qui ne peut pas être effectuée correctement par des non-spécialistes. Il est en effet non seulement nécessaire de s'assurer que les opérations réalisées sont correctes en toutes circonstances, y compris pour ce qui concerne le traitement des erreurs, mais également de prendre en compte les fuites d'information qui peuvent survenir via des canaux inattendus [5]. C'est pourquoi il est impératif de n'employer que des bibliothèques éprouvées bénéficiant d'un suivi de leur sécurité pour tout appel à des mécanismes cryptographiques. C'est également vrai pour la génération de clés symétriques ou asymétriques. La génération de clés asymétriques fait en particulier appel à des méthodes mathématiques non triviales, ce qui est une raison supplémentaire de ne pas la réimplémenter soi-même.

Attaques en lien avec le HW

- Attaques matérielles

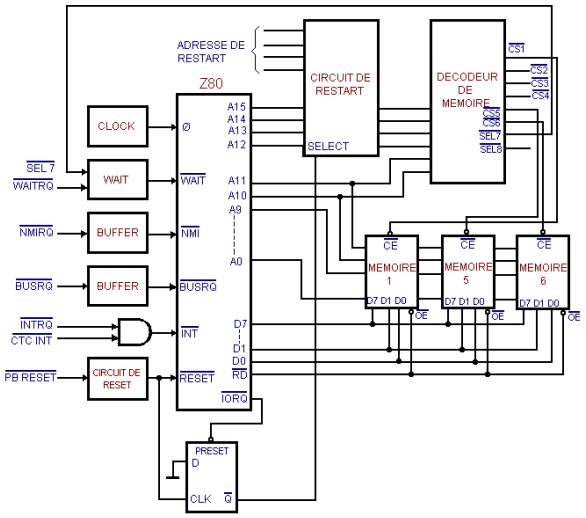
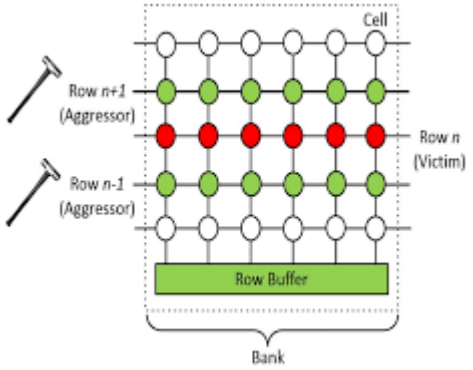


Fig. 30. - Schéma synoptique simplifié de la carte CPU.

- Attaques sur la micro-architecture
  - Caches, pipeline
  - Exécution spéculative (Spectre)



Secure element
and embedded security

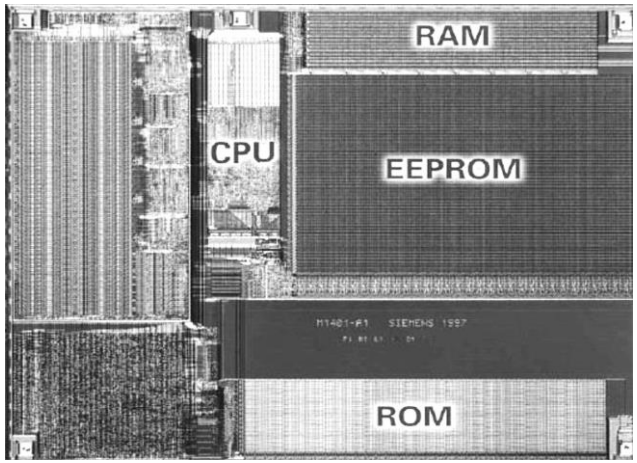
Fonctionnalités

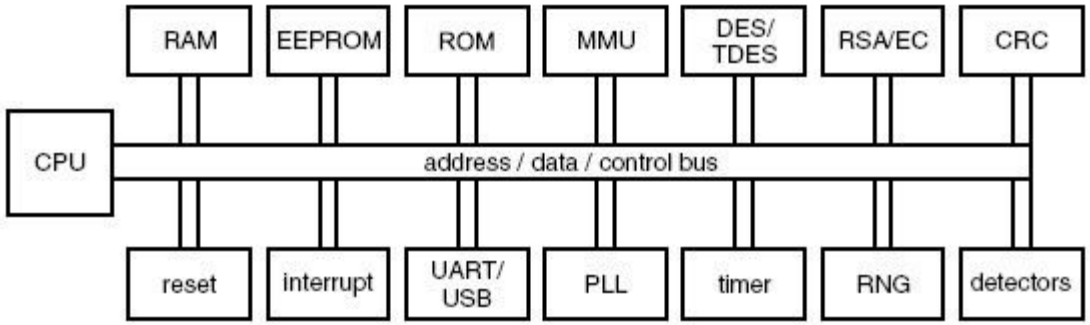
- Identification et authentification du porteur
  - Code PIN qui ne sort jamais
- Authentification de la carte
  - Exécuter un calcul cryptographique qui dépend d'un secret interne
- Certification d'information
- Cryptographie

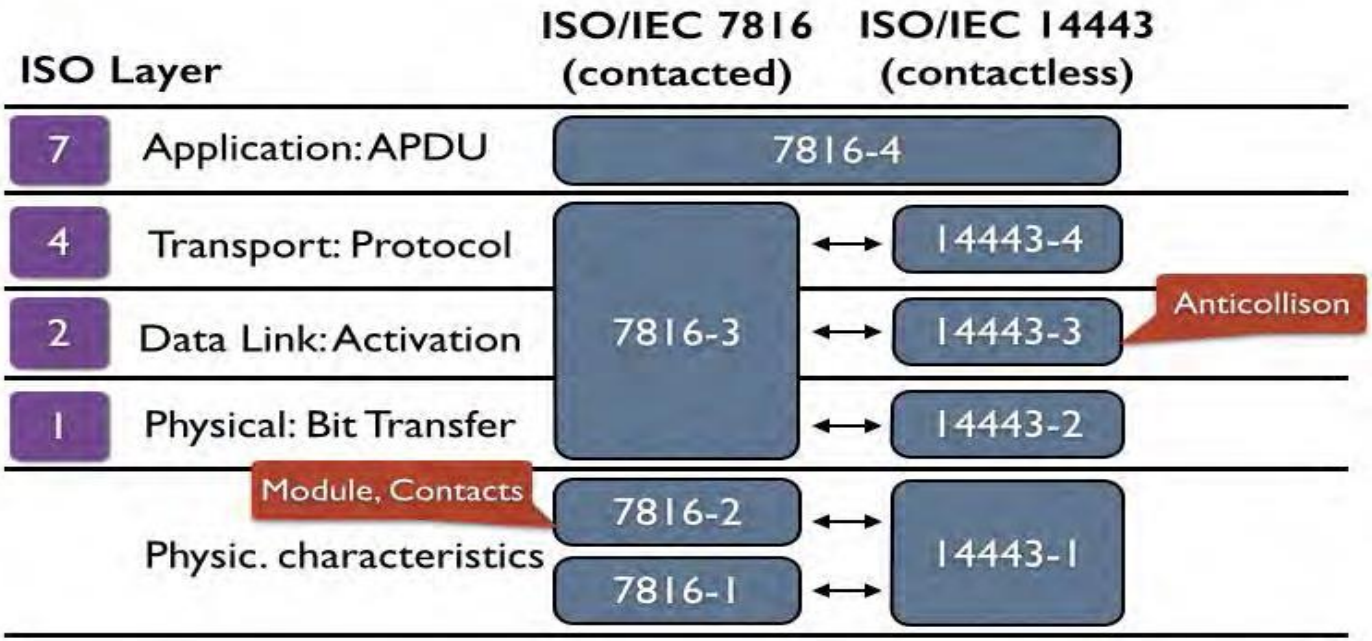
Sécurité de la carte

⇒ repose sur son caractère non reproductible (et donc inviolable) :

- Contient des secrets inviolables
- Des protections matérielles contre le reverse : intégration sur un unique ship (complexe de suivre les flux), type de mémoire particulière (EPROM/EEPROM), glue et capteurs
- Impossibilité d'accéder directement aux mémoires (opérations lecture/écriture contrôlées par le micro-processeur)
- Potentiellement non reproductible par l'émetteur (signature électronique)


RAM
CPU
EEPROM
ROM


RAM
EEPROM
ROM
MMU
DES/ TDES
RSA/EC
CRC
CPU
address / data / control bus
reset
interrupt
UART/ USB
PLL
timer
RNG
detectors


ISO Layer
ISO/IEC 7816 (contacted)
ISO/IEC 14443 (contactless)
7
Application: APDU
7816-4
4
Transport: Protocol
14443-4
2
Data Link: Activation
7816-3
14443-3
Anticollison
1
Physical: Bit Transfer
14443-2
Module, Contacts
7816-2
Physic. characteristics
14443-1
7816-1

Different Types of Memory …

- ROM : CPU only NO ACCESS !
  - used for embedded Operating System
- EPROM : Write once, read FOR EVER !
  - Used for initialization area (eg. Lock bytes)
- EEPROM : Write, erase, read FLEXIBLE !
  - used to store applicative data or added functionalities
- RAM :Write, erase, read TEMPORARY !
  - used during power on sessions only

Attaques matérielles

Attaques non-invasives

- Attaque par analyse de temps
- Attaque par force brute
- Attaque par analyse de consommation
- Analyse électromagnétique (EMA)
- Rémanence des données (lire la RAM)

Attaques Semi-Invasives

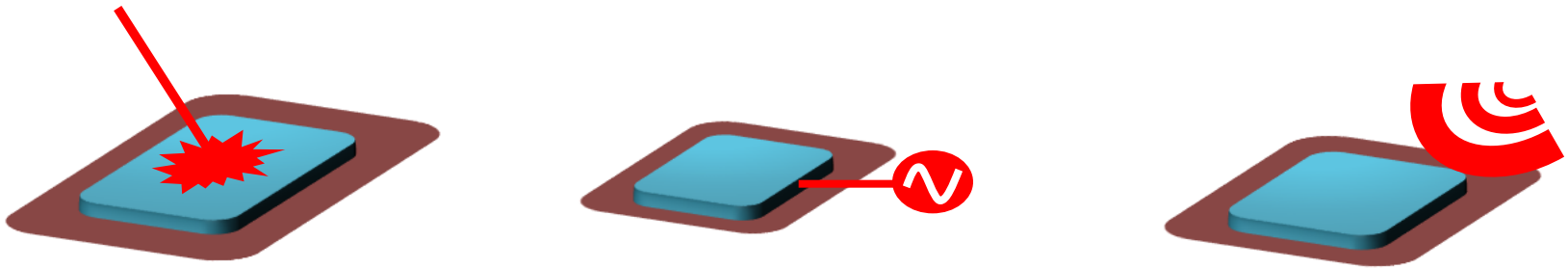
- Attaque par injection de fautes
- Attaque par rayonnement ultraviolet

Attaques Invasives

- Reverse de la puce

Exemples

- Canaux cachés et SPA/DPA
- Injection de fautes (laser, power glitch, EM)



Timing attaque

Mesure du temps de calcul pendant l'exécution d'une commande.
Exemple de vérification de PIN.

```
int verify_pin(char buffer[4]) {
    int i;
    authenticated = 1;
    for(i = 0; i < 4; i++)
        if(buffer[i] != pin[i]) {
            authenticated = 0;
            break;
        }
    return authenticated;
}
```

Reference: P. Kocher, *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*, CRYPTO 1996

Timing attaque

Mesure du temps de calcul pendant l'exécution d'une commande.
Contre-mesure : boucle en temps constant.

```
int verify_pin(char buffer[4]) {
    int i;
    authenticated = 1;
    for(i = 0; i < 4; i++)
        if(buffer[i] != pin[i]) {
            authenticated = 0;
            break;
        }
    return authenticated;
}
```

Reference: P. Kocher, *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*, CRYPTO 1996

Simple Power Analysis (SPA)

Mesure de la consommation pendant les calculs : exponentiation binaire (RSA)

```
for (i = k -2; i > 0 ; --i) {
    // Square
    C = C*C % N;
    if (e[i] == 1) {
        // Mult
        C = C*M % N;
    } else { // Nothing
           }
return (C) }
```

Square

Mul

Reference: T.S. Messerges, E.A. Dabbish and R.H. Sloan, *Power Analysis Attacks of Modular Exponentiation in Smartcards*, CHES 1999

$$C = P^e \bmod N$$
$$P = C^d \bmod N$$

C: Cipher Text

P: Plain Text

e: Public Key

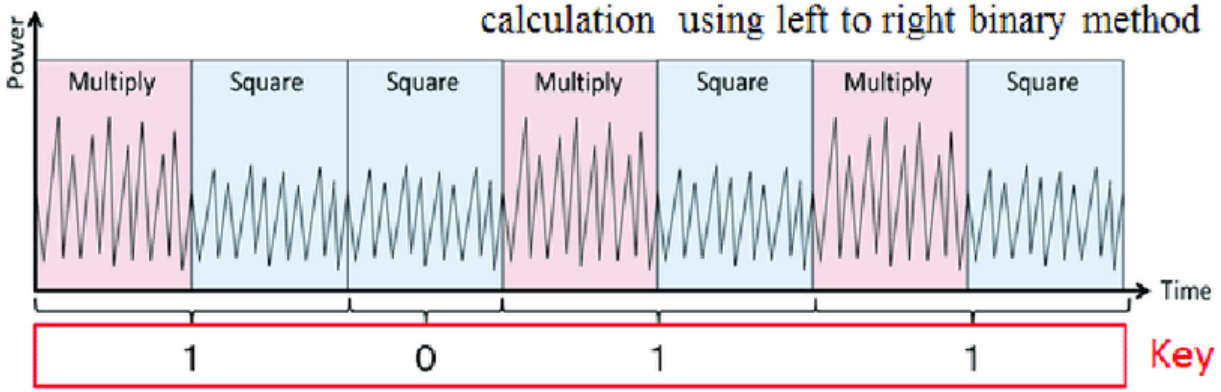
d: Private Key

N: modulo

(a) RSA crypto algorithm

input : X, N, $d = (d_{k-1}, d_{k-2}, \ldots\ldots, d_0)$
output: $Z = X^d \bmod N$
$Z \leftarrow 1$;
For $i = k - 1$ down to 0 do
  $Z \leftarrow Z \times Z \bmod N$; //Square
    if ($d_i = 1$) then
    $Z \leftarrow Z \times X \bmod N$; //Multiply
  end
end
return Z;

(b) Modular exponentiation (X^d mod N) calculation using left to right binary method



(c) Power dissipation model during modular exponentiation

Simple Power Analysis (SPA)

Contre-mesure : operation sans effet

```
for (i = k -2; i > 0 ; --i) {
     R = rand();
     C = C*C % N;
     if (e[i] == 1) {
          // Mult
          C = C*M % N;
     } else { R = R*M % N;
              }
return (C) }
```

Reference: T.S. Messerges, E.A. Dabbish and R.H. Sloan, *Power Analysis Attacks of Modular Exponentiation in Smartcards*, CHES 1999

Injection de fautes

```
for (i = k -2; i > 0 ; --i) {
    R = rand();
    C = C*C % N;
    if (e[i] == 1) {
        // Mult
        C = C*M % N;
    } else { R = R*M % N; }
return(C) }
```

Supposons qu'on puisse modifier une valeur lue. Que gagne t-on ?

Injection de fautes

```
for (i = k -2; i > 0 ; --i) {
     R = rand();
     C = C*C % N;
     if (e[i] == 1) {
          // Mult
          C = C*M % N;
     } else { R = R*M % N; }
return(C) }
```

Supposons qu'on puisse modifier une valeur lue. Que gagne t-on ?

On attaque la valeur lue `e[i]` : si pas de modification du résultat on avait `e[i]`=0 sinon on avait `e[i]`=1.

Injection de fautes

```
for (i = k -2; i > 0 ; --i) {
    R = rand();
    C = C*C % N;
    if (e[i] == 1) {
        // Mult
        C = C*M % N;
    } else { R = R*M % N; }
return(C) }
```

Supposons qu'on puisse modifier une valeur lue. Que gagne t-on ?

Contre-mesure : échelle de Montgomery.

Contre-mesures

Contre-mesures logicielles

- Génération d'aléa dans l'horloge
- Dispersion des informations
- Compteur d'erreur
- Redondance des calculs, compteur d’instruction …

Contre-Mesures matérielles

- Sécurité par l'obscurité
  - Glue logique
  - Démarquage des puces
  - Miniaturisation
  - Mélange des circuits et de la mémoire
- Ajout de sondes
- Ajout d'une couche protectrice

Programmation sécurisée

- Maitrise fine du langage (pb conversion signed/unsigned …) en particulier cas embarqué
- Vérifier les données (valeurs, nombre …) avant toute chose
- Ne pas donner trop d'information en cas d'erreurs
- Attention à la programmation Crypto

Programmation sécurisée (suite)

- Classifier les objets (sensible ou non) et leur durée de vie (transient/persistant)
- Mettre en place des transactions lorsque nécessaire (état cohérent en cas de problème)
- Effacer les données lorsque plus utiles (challenge, clé de session …)
- Robustesse aux attaques canaux cachés et injection de fautes

Un verifypin durci

```
1.  boolean verify (byte[] buffer, short ofs, byte len)
2.  {
3.   // No comparison if PIN is blocked
4.   if (triesLeft < 0)
5.    return false ;
6.   // Main comparison
7.   for(short i=0; i < len; i++)
8.    if (buffer[ofs+i] != pin[i])
9.    {
10.    triesLeft-- ;
11.    authenticated[0] = false ;
12.    return false ;
13.   }
14.  // Comparison is successful
15.  triesLeft = maxTries ;
16.  authenticated[0] = true ;
17.  return true ;
18. }
```

Faiblesses ?

Les faiblesses

=> Les objets à protéger/garantir : triesleft, résultat de l'authentification

- Attaque sur le temps
  - Boucle en temps constant
- Pas de résistance à l'arrachage
  - Décrémentation a priori, mise à faux, absence de transactions
- Injection de fautes

Correction 1

```
1.  {
2.    authenticated[0] = false ;
3.  // No comparison if PIN is blocked
4.    if (triesLeft < 0)
5.     return false ;
6.    // First decrements the number of remaining tries
7.    triesLeft-- ;
8.    // Main comparison
9.    boolean equal = true ;
10.   for(short i=0; i < len; i++)
11.    equal = (equal && (buffer[ofs+i] != pin[i])) ;
12.   if (!equal)
13.   {
14.    // Comparison failed
15.    authenticated[0] = false ;
16.    return false ;
17.   } else {
18.    // Comparison is successful
19.    triesLeft = maxTries ;
20.    authenticated[0] = true ;
21.    return true ;
22.   }
23. }
```

Correction 2

- The Java Card runtime environment maintains an atomic transaction commit buffer which is initialized on card reset (or power on). When a transaction is in progress, the Java Card runtime environment journals all updates to persistent data space into this buffer so that it can always guarantee, at commit time, that everything in the buffer is written or nothing at all is written. The JCSystem includes methods to control an atomic transaction.

```
1.   {
2.   if (JCSystem.getTransactionDepth() != 0) TransactionException.throwIt((short)1);
3.    authenticated[0] = false ;
4.  // No comparison if PIN is blocked
5.    if (triesLeft < 0)
6.     return false ;
```

Protection contre les fautes

⇒bit-set/bit-reset sur les data

- Encoder les booléens en :

```
public final static short BOOL_TRUE  = (short)0x5a5a ;
public final static short BOOL_FALSE = (short)0xa5a5 ;
short equal = BOOL_TRUE ;
```

```
1.      for(short i=0; i < len; i++)
2. If (short)(equal & (buffer[ofs+i] != [i])) equal= BOOL_TRUE else equal= BOOL_FALSE ;
3. if (equal == BOOL_TRUE)
4. {
5. // Comparison is successful
6. triesLeft = maxTries ;
7. authenticated[0] = true ;
8. return true ;
9. } else {…
```

5 : 0101 a : 1010

- Copie en local (ram/eeprom)
- valeur inversée
- mise à jour atomique

=> Même technique que la tolérance aux fautes (radiation, perturbations électromagnétiques, …)

```
1.  {
2.   // First checks the integrity of the variable
3.   byte tl = triesLeft ;
4.   if (tl != (short)(~triesLeftBackup))
5.    takeCountermeasure() ;
6.   // No comparison if PIN is blocked
7.   if (tl < 0)
8.    return false;
9.   // First decrements the number of remaining tries
10.  JCSystem.beginTransaction() ;
11.   triesLeft = --tl ;
12.   triesLeftBackup++ ;
13.  JCSystem.commitTransaction() ;
14.  // Verifies the new value
15.  if (triesLeft != (short)(~triesLeftBackup))
16.   takeCountermeasure() ;
```

Pour aller plus loin

=> ajout intégrité du flot de contrôle

```
1.  // Main comparison
2.  boolean equal = true ;
3.  for(short i=0; i < len; i++)
4.   equal = (equal && (buffer[ofs+i] != pin[i])) ;
5.  if (i !=len) countermeasure();
```

- Compter les instructions
- Vérifier l'enchainement des blocs de base
- Vérifier les appels / retour de fonctions

⇒ calcul statique à la compilation

⇒ implémentées en optionnel par certains compilateurs (clang -fsanitize=cfi)

```
1.  boolean verify(byte[] buffer, short ofs, byte len)
2.  {
3.   // Initializes the step counter
4.   short stepCounter = INITIAL_COUNTER ;
5.   // First checks the integrity of the variable
6.   byte tl = triesLeft ;
7.  stepCounter++ ;
8.   if (tl != (short)(~triesLeftBackup))
9.    takeCountermeasure() ;
10.  stepCounter++ ;
11.  // No comparison if PIN is blocked
12.  if (tl < 0)
13.   return false ;
14.  stepCounter++ ;
15.  // First decrements the number of remaining tries
16.  JCSystem.beginTransaction() ;
17.   triesLeft = --tl ;
18.   stepCounter++ ;
19.   triesLeftBackup++ ;
20.  JCSystem.commitTransaction() ;
21.  stepCounter++ ;
22.  // Verifies the new value
23.  if (triesLeft != (short)(~triesLeftBackup))
24.   takeCountermeasure() ;
25.  stepCounter++ ;
26.  // Main comparison
27.  short equal = BOOL_TRUE ;
28.  stepCounter++ ;
29.      for(short i=0; i < len; i++)
30.      If (short)(equal & (buffer[ofs+i] != [i])) equal= BOOL_TRUE else equal=
   BOOL_FALSE ;
31.   equal = equal && (buffer[ofs+i] != pin[i]) ;
32.  stepCounter++ ;
33.  if (equal == BOOL_TRUE)
34.  {
35.   // Comparison is successful
36.   // Reset the remaining tries to the max
37.   stepCounter++ ;
38.   JCSystem.beginTransaction() ;
39.    triesLeft = maxTries ;
40.    triesLeftBackup = (byte)(~maxTries) ;
41.   JCSystem.commitTransaction() ;
42.   stepCounter++ ;
43.   // Verifies the new value
44.   if ( (triesLeft != (short)(~triesLeftBackup)) ||
45.      (triesLeft != triesLeftBackup) )
46.    takeCountermeasure() ;
47.   stepCounter++ ;
```

Crypto coding rules

https://github.com/veorq/cryptocoding

1. compare secret strings in constant time
2. avoid branchings controlled by secret data
3. avoid table look-ups indexed by secret data
4. avoid secret-dependent loop bounds
5. prevent compiler interference with security-critical operations
6. prevent confusion between secure and insecure APIs
7. avoid mixing security and abstraction levels of cryptographic primitives in the same API layer
8. use unsigned bytes to represent binary data
9. use separate types for secret and non-secret information
10. use separate types for different types of information
11. clean memory of secret data
12. use strong randomness

Compare secret strings in constant times

- Built-in comparison functions such as C's memcmp, Java's Arrays.equals, or Python's == test may not execute in constant time.
- Use a constant-time comparison function:
  - With OpenSSL, use CRYPTO_memcmp
  - In Python 2.7.7+, use hmac.compare_digest
  - In Java, use java.security.MessageDigest.isEqual
  - In Go, use package crypto/subtle

A portable C solution, for non-buggy compilers, follows:

```
void burn( void *v, size_t n )
  { volatile unsigned char *p = ( volatile unsigned char *
)v;
    while( n-- ) *p++ = 0; }
```

Unfortunately, there's virtually no way to reliably clean secret data in garbage-collected languages (such as Go, Java, or JavaScript), nor in language with immutable strings (such as Swift or Objective-C).

Cryptographie

https://github.com/veorq/cryptocoding/

Contents

1 Compare secret strings in constant time

2 Avoid branchings controlled by secret data

3 Avoid table look-ups indexed by secret data

4 Avoid secret-dependent loop bounds

5 Prevent compiler interference with security-critical operations

6 Prevent confusion between secure and insecure APIs

7 Avoid mixing security and abstraction levels of cryptographic primitives in the same API layer

8 Use unsigned bytes to represent binary data

9 Use separate types for secret and non-secret information

10 Use separate types for different types of information

11 Clean memory of secret data

12 Use strong randomness

…

When possible, consider disabling compiler optimizations that can eliminate or weaken security checks.

To prevent the compiler from "optimizing out" instructions by eliminating them, a function may be redefined as a volatile pointer to force the function pointer dereference. This is for example used in libottery by redefining memset to

void * (*volatile memset_volatile)(void *, int, size_t) = memset;

Note that such workarounds may not be sufficient and can still be optimized out.

C11 introduced memset_s with a requirement that it is not optimized out. It's an optional feature that can be requested when including string.h.

Security Policy

Reporting security issues

If you wish to report a possible security issue in OpenSSL please notify us.

Issue triage

Notifications are received by the OMC and OTC. We engage resources within OpenSSL to start the investigation and prioritisation. We may work in private with individuals who are not on the OMC or OTC as well as other organisations and our employers where we believe this can help with the issue investigation, resolution, or testing.

Threat Model

Certain threats are currently considered outside of the scope of the OpenSSL threat model. Accordingly, we do not consider OpenSSL secure against the following classes of attacks:

- same physical system side channel
- CPU/hardware flaws
- physical fault injection
- physical observation side channels (e.g. power consumption, EM emissions, etc)

Mitigations for security issues outside of our threat scope may still be addressed, however we do not class these as OpenSSL vulnerabilities and will therefore not issue CVEs for any mitigations to address these issues.

We are working towards making the same physical system side channel attacks very hard.

Prior to the threat model being included in this policy, CVEs were sometimes issued for these classes of attacks. The existence of a previous CVE does not override this policy going forward.