Sécurité Informatique

A not so short introduction...

Guillaume Duc
<guillaume.duc@telecom-paristech.fr>
2020–2021

Première partie I

Introduction générale

Sécurité Informatique

Définitions

- Sécurité ?
- Informatique ?

Sécurité Informatique

Définitions

- N'oubliez pas qu'un ordinateur ne se présente pas seulement sous la forme d'un ordinateur fixe, d'un ordinateur portable ou d'un serveur...
- On trouve des ordinateurs sous de très nombreuses formes : ce sont les *Systèmes Embarqués*
- Ces systèmes sont également très susceptibles (parfois même plus) aux problèmes de sécurité

Objectifs de sécurité & Modèle d'attaquant

- Avant d'essayer de sécuriser un système, il est important de se poser les questions suivantes
  - Quels sont les éléments de valeur dans mon système ?
  - Quelles propriétés de sécurité garantir sur ces éléments ?
  - Quels sont les moyens à la disposition de l'attaquant ?
- Les réponses à ces questions sont fondamentales (pour bien protéger ce qui est important, pour ne pas sous-dimensionner ou sur-dimensionner les protections)

Propriétés de sécurité

Confidentialité Garantir que seuls les personnes/programmes autorisés peuvent accéder à une information

Intégrité Garantir qu'une information ne peut pas être modifiée, involontairement ou volontairement, ou garantir la détection de cette altération

Authentification Prouver l'identité d'une personne, d'un programme, d'une machine, etc.

Disponibilité Garantir qu'un système reste en permanence utilisable par les personnes autorisées

Traçabilité Permettre de savoir qui a fait quoi sur un système informatique

Imputation, Non répudiation ...

Problème

- Sécurité informatique = Sécurité d'une chaîne = Sécurité du plus faible maillon
- Or il y a (vraiment) beaucoup de maillons dans un système informatique...

Objectifs de ce cours

- Donner un aperçu le plus large possible des domaines couverts par la sécurité informatique
  - Matériel
  - Système d'exploitation
  - Logiciel
- La partie réseau (également importante et très vaste) ne sera pas abordée (mais elle est bien évidemment tout aussi importante)
- Présenter quelques attaques de façon détaillée dans chacun des domaines

Objectifs de ce cours

- Ce cours n'est pas une référence exhaustive (un cours de référence couvrant tout le domaine ne peut pas être fait en 20 h, ni même en 100 h !)
- Ce cours n'est peut être déjà plus à jour au moment où vous le lirez (la sécurité informatique est un domaine en mouvement permanent, de nouvelles attaques apparaissent presque chaque jour...) !
- $\rightsquigarrow$Vous devrez vous tenir au courant de l'actualité du domaine

Découpage du cours

- Sécurité matérielle
- Sécurité des systèmes d'exploitation
- Sécurité des applications
- Conclusion et ouverture

Deuxième partie II

Sécurité matérielle

Plan

Plan

Le matériel, c'est dépassé ?

- Pourquoi parler du matériel à l'heure de la virtualisation et du cloud computing à outrance ?
- Les informations (programmes, données, etc.) sont toujours traitées par du matériel, y compris dans le "cloud"
  - Processeur
  - Mémoire vive
  - Stockage de masse
  - Bus
  - Périphériques (carte réseau, carte graphique, etc.)
- Et le matériel est vulnérable à des attaques qui peuvent remettre en cause la sécurité des couches logicielles
- Ces attaques sont très présentes dans le monde des systèmes embarqués mais concernent également les ordinateurs plus classiques et les serveurs

Plan

Exemple : Microsoft XBox I

Exploit [11]

- Objectif de sécurité : empêcher l'exécution de logiciels non prévus (OS ou applications)
- Chaîne de confiance : chaque étape déchiffre et vérifie la suivante avant de l'exécuter
  - Bloc de démarrage secret (ASIC Southbridge)
  - Bootloader (Flash)
  - Système d'exploitation (Flash)
  - Applications (Disque dur ou support optique)

Exemple : Microsoft XBox I

Architecture matérielle

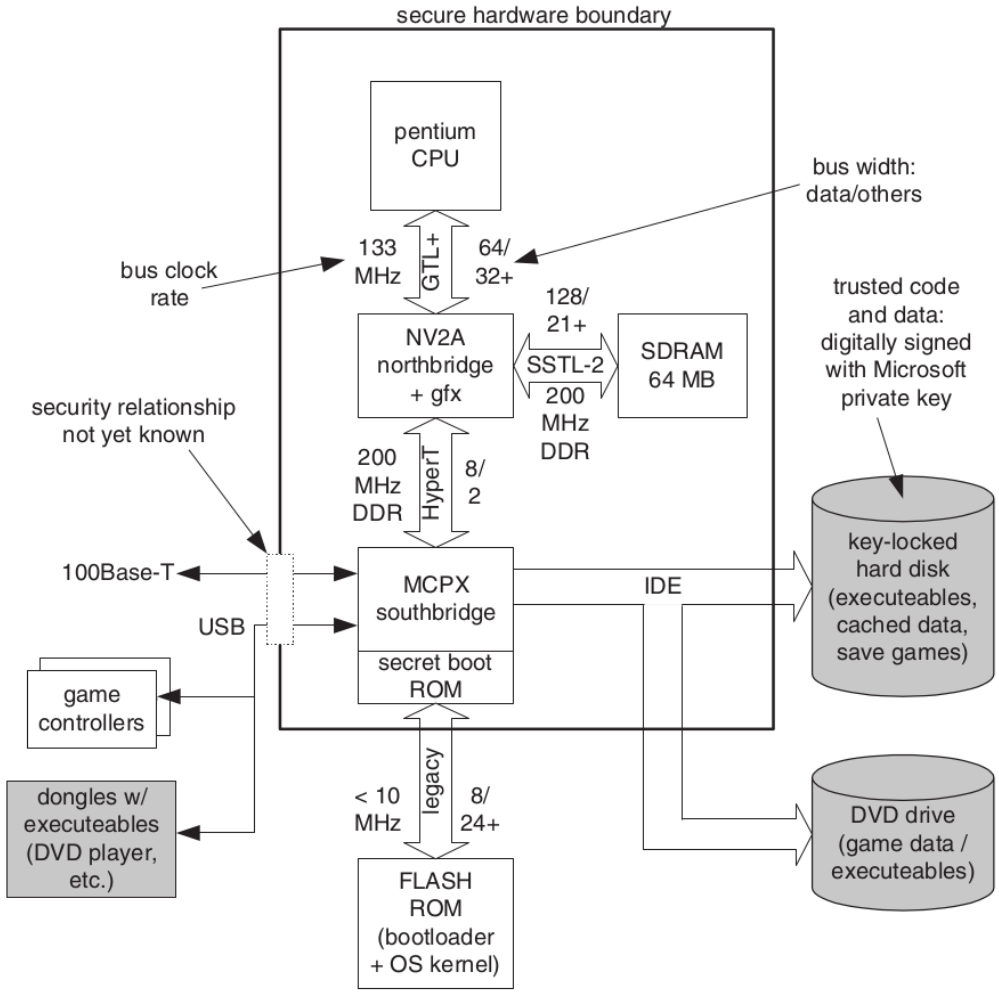


Fig. 1. Overview of the Microsoft Xbox hardware.

Source : Andrew “bunnie” Huang [11]

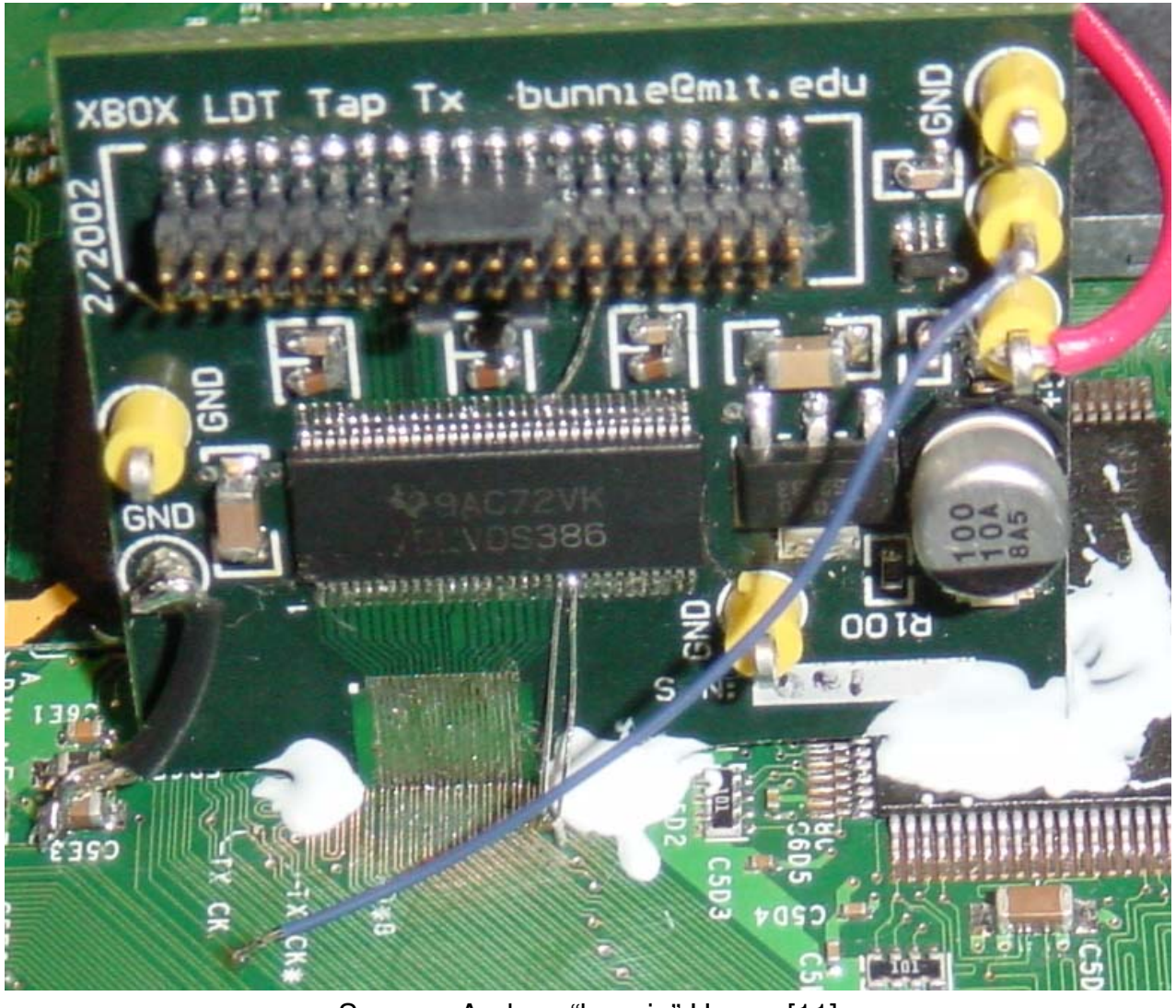
Exemple : Microsoft XBox I

Espionnage bus Northbridge/Southbridge

- Au démarrage, le code du bloc de démarrage secret transite par le bus Northbridge $\leftrightarrow$ Southbridge (*HyperTransport*) pour être exécuté par le processeur
- Ce code a été récupéré (voir [11]) en espionnant ce bus et analysé
- Il effectue plusieurs opérations
  - Initialisation du processeur et des périphériques (via une petite machine virtuelle exécutant des instructions en mémoire flash)
  - Chargement et déchiffrement du bootloader depuis la flash
  - Vérification de l'intégrité du bootloader
  - Si le bootloader est bien celui attendu : lancement de ce dernier

Exemple : Microsoft XBox I

Espionnage bus Northbridge/Southbridge



Source : Andrew “bunnie” Huang [11]

Exemple : Microsoft XBox I

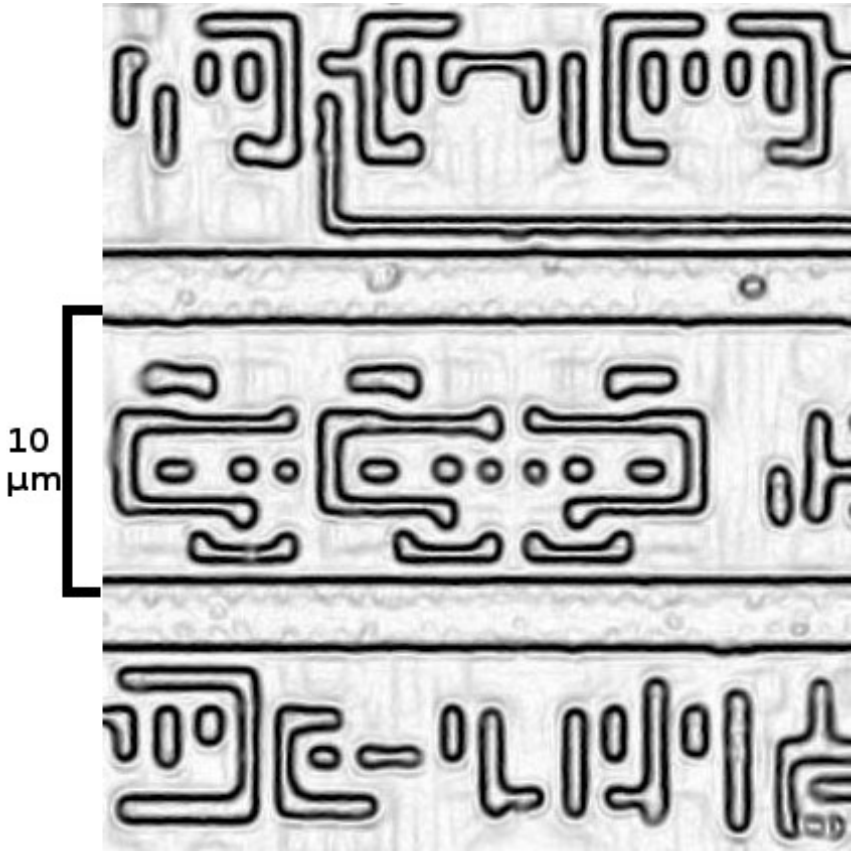
Vulnérabilités du bloc de démarrage

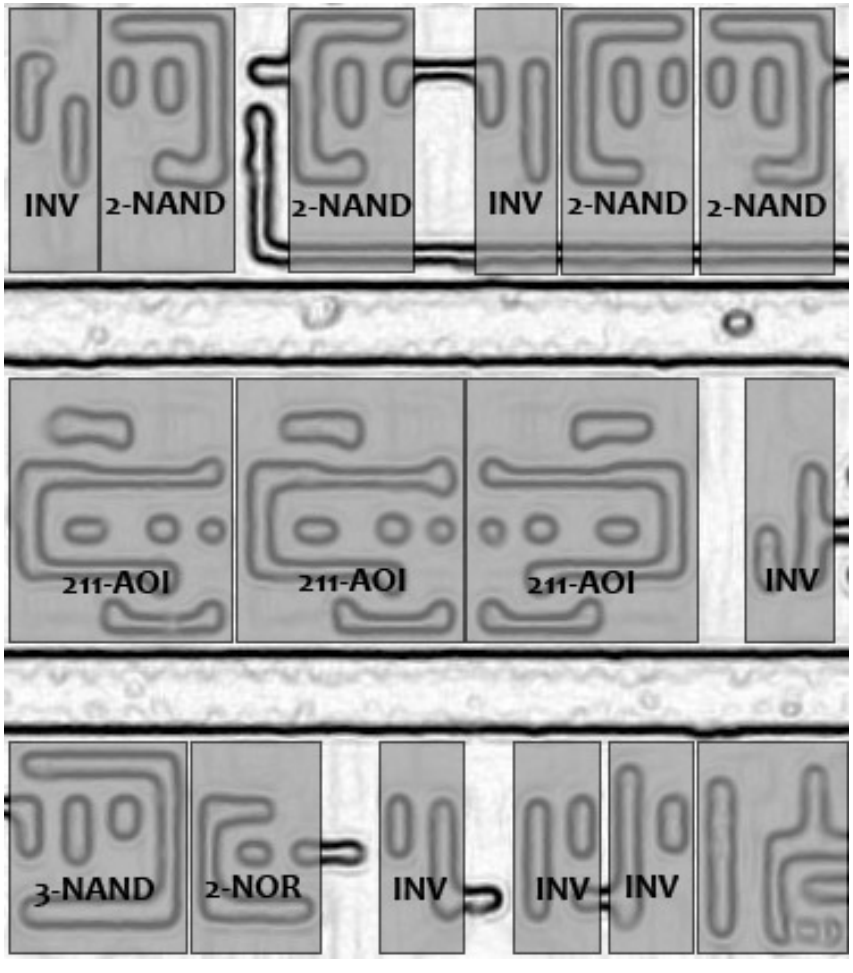
- Bootloader chiffré grâce à une variante de l'algorithme RC-4 et une clé contenue dans le code de démarrage (qui a été récupérée lors de l'espionnage du bus)
- Vérification d'intégrité : valeur magique à la fin du bootloader...
- Instructions pour l'initialisation du système chargées depuis une partie non protégée de la flash : possibilité de les modifier pour configurer le système de afin de démarrer même si la vérification du bootloader échoue
- $\rightsquigarrow$Avec ces informations, il est facile de générer un bootloader qui soit correctement déchiffré et vérifié (et donc accepté) par la XBox

MIFARE Classic

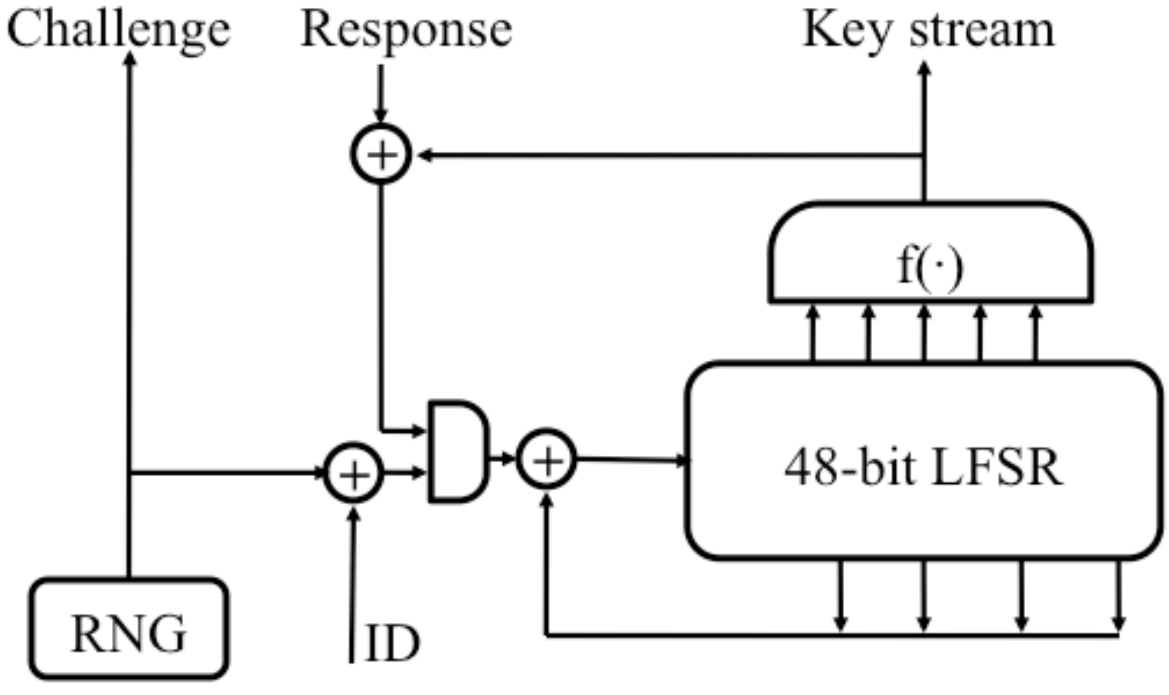
- Carte sans contact (RFID) contenant de la mémoire non volatile et un algorithme d'authentification secret (Crypto-1) pour en protéger l'accès
- Utilisé principalement pour du contrôle d'accès et du ticketing (métro de Londres)
- Henryk Plötz and Karsten Nohl [15] ont réussi à retrouver l'algorithme
  - Ils ont découpé et photographié les différentes couches du circuit puis reconstitué le schéma électronique du circuit
- En analysant l'algorithme, ils ont découvert plusieurs faiblesses cryptographiques dans cet algorithme
- `http://events.ccc.de/congress/2007/Fahrplan/events/2378.en.html`

MIFARE Classic [15]





MIFARE's CRYPTO1 exposed [15]



Images of MIFARE by Karsten Nohl, David Evans, Starbug and Henryk Plötz

Plan

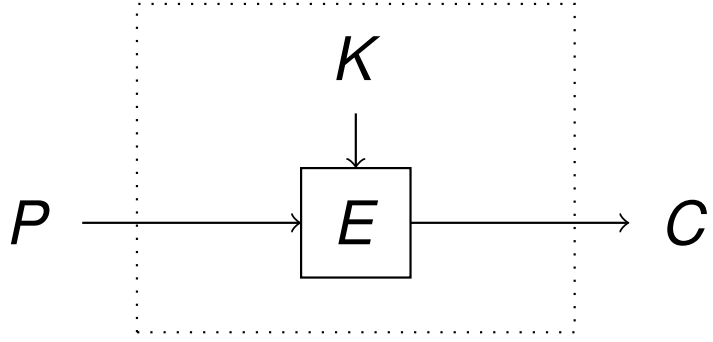
Plan

Introduction

Exemple

- Algorithme de chiffrement par bloc contenu dans un circuit électronique (implémenté en dur dans un ASIC ou un FPGA ou réalisé grâce à du logiciel embarqué)
- Entrée : message à chiffrer
- Sortie : message chiffré
- Par conception, la clé de chiffrement est embarquée dans le circuit et n'est pas accessible

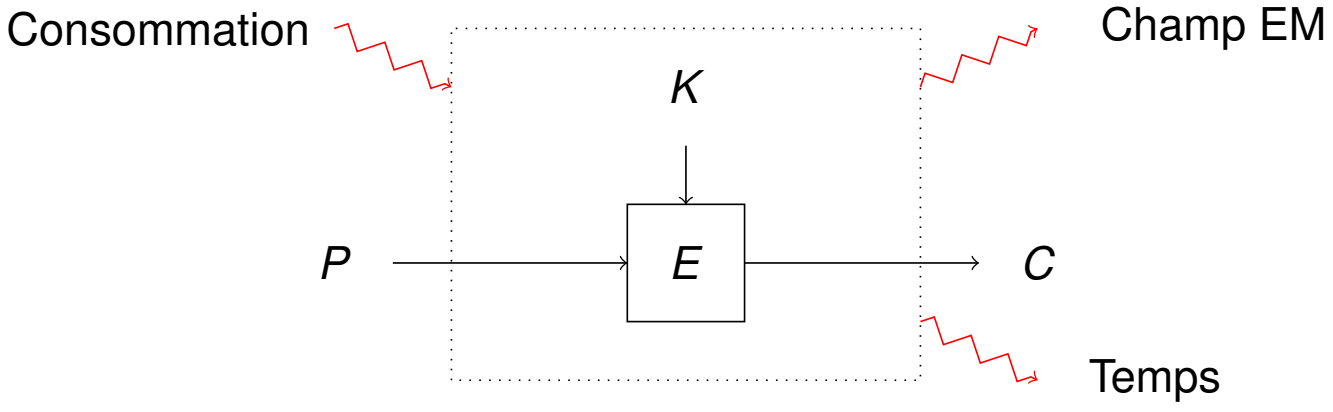
Vue mathématique



- Principes de Kerckhoffs : P, C et E sont publics et la sécurité ne dépend que de K qui est inconnue de l'adversaire
- Plusieurs algorithmes sont aujourd'hui considérés comme incassables en pratique dans ce modèle

Dans la vraie vie...

... le matériel



- Canaux auxiliaires (side channels)
  - Rayonnement EM
  - Consommation électrique
  - Temps de calcul
  - ...

Canaux auxiliaires

Problématique

- Des opérations différentes ou des opérations identiques sur des données différentes vont potentiellement entraîner un comportement différent du circuit sur un (ou plusieurs) de ses canaux auxiliaires (par exemple sa consommation)
- Le circuit va donc laisser fuir de l'information sur les opérations qu'il effectue ou les données qu'il manipule
- En observant ces canaux auxiliaires, un adversaire peut accumuler de l'information sur les opérations effectuées par le circuit qui peuvent être liées au secret...

Simple Power Analysis (SPA)

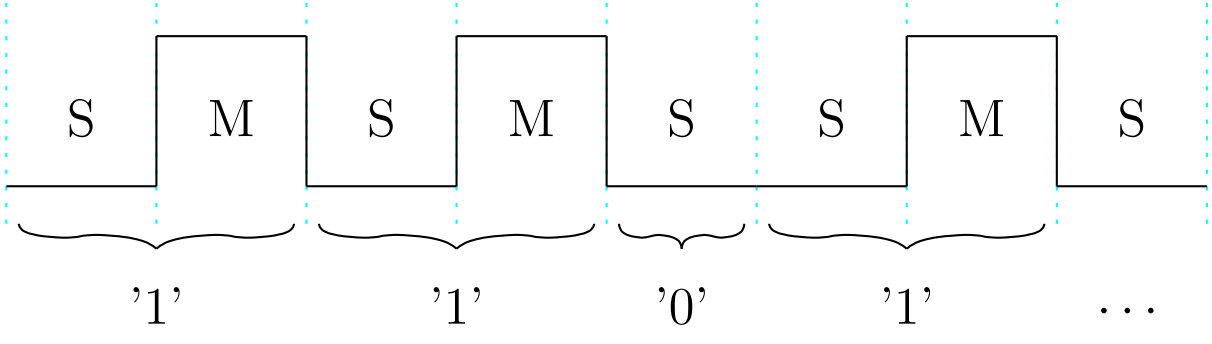
Exemple sur RSA

Square-and-Multiply RSA

Inputs : M , K
$R = 1$;
for $i = |K| - 1 ; i \geq 0 ; i - -$ **do**
    $R = R^2$;
    `/* Unbalanced branching */`
    if $K_i == 1$ **then**
        $R = R \times M$;
    end if
end for
Return $R = M^K$;

Montgomery ladder exponentiation

Inputs : M , K
$R = 1 ; S = M$;
for $i = |K| - 1 ; i \geq 0 ; i - -$ **do**
    `/* Balanced branching */`
    if $K_i == 1$ **then**
        $R = R \times S ; S = S^2$;
    else
        $S = S \times R ; R = R^2$;
    end if
end for
Return $R = M^K$;



Differential Power Analysis (DPA)

- Très souvent, la fuite sur le canal auxiliaire n'est pas aussi flagrante
- $\rightsquigarrow$Utilisation de nombreuses mesures (traces) et d'outils statistiques pour la détecter
- Attaques DPA (Différence de moyennes), CPA (Corrélation), VPA (Variance), MIA (Information mutuelle), etc.

SCA

Un peu de recul

- Consommation : nécessite un accès physique direct au composant
- Rayonnement EM : exploitable à faible distance
- Temps de calcul : exploitable à distance, y compris à travers un réseau
  - Vol de clés RSA à distance via le réseau...
- Plus récemment, les mécanismes d'optimisation présents dans les processeurs ouvrent la voie à d'autres canaux auxiliaires permettant d'aller voler des secrets d'une autre application ou d'une autre machine virtuelle
  - Caches (utilisés par les attaques *Spectre* et *Meltdown* notamment)

SCA contre les caches

- Le temps d'accès à une donnée varie en fonction de sa présence ou non dans les caches du processeur
- La présence ou l'absence d'une donnée dans le cache est donc une information qui peut être retrouvée par une attaque par canal auxiliaire basée sur la mesure du temps d'exécution d'un accès à cette donnée
- Les processeurs modernes intègrent souvent plusieurs niveaux de caches, en général trois
  - Un niveau L1, rattaché à un cœur, adressé virtuellement, avec séparation Instructions / Données
  - Un niveau L2 intermédiaire
  - Un niveau L3, commun à l'ensemble des cœurs, adressé physiquement et souvent inclusif

SCA contre les caches

- Le but de l'attaque est de détecter si un processus a fait un accès mémoire et à quelle adresse
- Le processus visé peut être un processus attaqué ou un processus complice qui souhaiterait faire fuir silencieusement de l'information (*covert-channel*)
- En cryptographie notamment, la connaissance des accès mémoires effectués par un processus peut permettre dans certains cas de retrouver des secrets
  - Utilisation de tables pré-calculées
  - Fonctions appelées...
- Plusieurs techniques d'attaques : Evict+Time, Prime+Probe, Flush+Reload...

SCA contre les caches

Exemple Flush+Reload [18]

- On suppose que le processus cible et le processus attaquant partagent une zone mémoire (explicitement via un segment mémoire partagé via `shmget`, ou implicitement : bibliothèques partagées)
- Les données situées dans cette zone sont potentiellement mappées à des adresses virtuelles différentes dans l'espace d'adressage des deux processus mais seront stockées aux mêmes adresses physiques

SCA contre les caches

Déroulement de l'attaque Flush+Reload [18]

- Le processus attaquant évince des différents caches les lignes intéressantes de la zone mémoire partagée (instruction x86 `clflush` qui ne nécessite pas de privilèges) puis laisse le processus cible s'exécuter
- Si le processus cible fait des accès dans cette zone, les données correspondantes vont être rapatriées dans le cache L3 du processeur
- Le processus attaquant reprend la main et mesure le temps d'accès à une donnée de la zone partagée
  - Si ce temps est court, cette donnée a été mise dans le cache lors de l'exécution du processus cible (accès direct ou spéculatif par le *prefetcher*)
  - Si le temps est long, la donnée n'a pas été accédée par le processus cible

Contre-mesures aux SCA

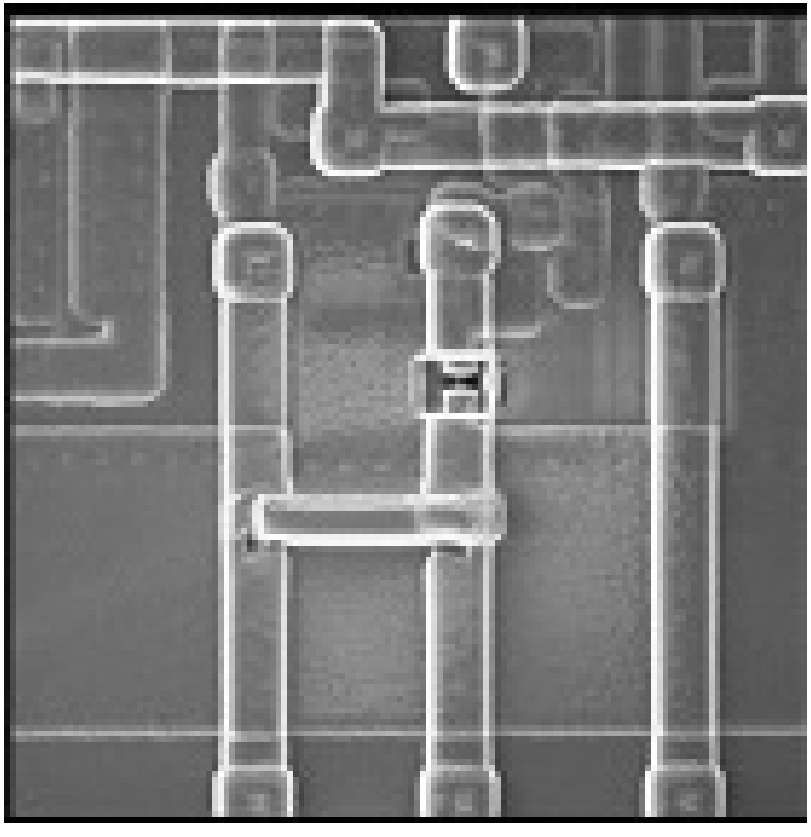
- Équilibrage : rendre la consommation/rayonnement/temps de calcul indépendante des données
- Masquage : obliger l'attaquant à récupérer plusieurs informations en même temps (attaques d'ordre élevé) en rendant inutilisable seule une information fuitée

Plan

Techniques

- Rétro-conception
- Modification
- Micro-sondes
- Injection de fautes
- ...

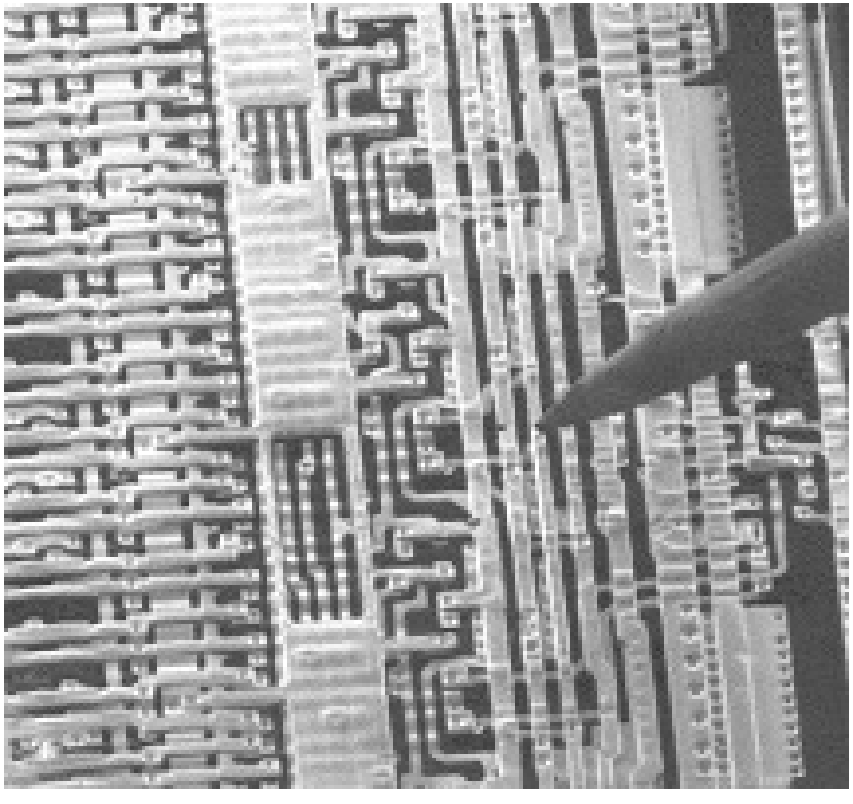
Modification d'un circuit à l'aide d'un FIB (Focused Ion Beam)



- Désactivation de capteurs de sécurité
- Restauration de fusibles (test mode...)
- Technique ultime

Image issue de la thèse de Christophe Giraud [7]

Micro sondes



- Permet par exemple de lire les données circulant sur un bus interne
- Technique très puissante
- Permet de lire que quelques fils en même temps et seulement sur le niveaux de métallisation supérieurs

Image issue de la thèse de Christophe Giraud [7]

Injection de fautes

- Principe
  - Il s'agit de perturber volontairement le fonctionnement du circuit pour lui faire commettre des erreurs
  - Ces erreurs peuvent mener à
    - Le saut d'un instruction (par exemple un test)
    - Un erreur dans le résultat (très dangereux en crypto)...
- Réalisation
  - Violation du chemin critique (augmentation de la fréquence de fonctionnement, augmentation de la température, diminution de la tension d'alimentation, etc.)
  - Injection de particules énergétiques (flash, laser, rayonnement ionisant)

Plan

Bugs matériels

- Les processeurs actuels comportent plusieurs centaines de millions de transistors et incluent de nombreux mécanismes pour accélérer les traitement
  - Exécution out-of-order
  - Prédiction de branchement
  - Multi-threading
  - Renommage de registre...
- Tout comme les logiciels, ils ne sont pas à l'abri de bugs, pouvant remettre en cause la sécurité

Bugs processeur

- Le bug le plus connu est celui de la division fautive du Pentium découvert en 1994
- Autre bug connu du Pentium : le F0 0F
  - L'instruction `lock cmpxchg8b eax` (en code machine `f0 0f c7 c8`) qui devrait normalement lever une exception fait planter le processeur qui doit alors être reinitialisé
  - Comme cette instruction peut être exécutée sans privilèges, cela crée un déni de service (un bug similaire existe dans les processeurs Cyrix)

Bugs processeur

- Les bugs des processeurs sont souvent inconnus du grand public
- Mais, s'ils touchent des fonctionnalités liées à la protection mémoire, ils peuvent avoir un impact important sur la sécurité
- Et quid des bugs qui n'en seraient pas (portes dérobées) ?
- Il est assez facile d'introduire une porte dérobée dans un processeur... (cf. [5])

Plan

Attaque *Meltdown* [13]

- Exécution des instructions dans le désordre (*out-of-order*)
  - Les processeurs disposent de plusieurs unités de calcul et de traitement (plusieurs ALU...)
  - Afin de maximiser l'utilisation de ces ressources, les instructions sont réordonnées (en conservant les dépendances éventuelles)
- Exécution spéculative
  - Des instructions peuvent être exécutées avant d'être certain qu'elles doivent l'être (i.e. après un saut conditionnel, avant d'avoir vérifié que les instructions précédentes ne génèrent pas d'exceptions...)
  - Si elles ne devaient pas l'être, leurs résultats sont annulés et aucun effet n'est visible par le logiciel
  - *Néanmoins, l'exécution de ces instructions peut laisser des traces au niveau micro-architecture qui peuvent être détectées*

Attaque *Meltdown* [13]

Exemple simple

- Considérons le code suivant (exemple tiré de [13]) :

```
exception();
accès mémoire(tableau[donnée * 4096]);
```

- Normalement la seconde instruction n'est pas exécutée
- Mais à cause de l'exécution spéculative dans le désordre, elle peut être exécutée avant que le processeur ne s'aperçoive qu'il ne devait pas le faire
- L'effet de cette instruction sera annulé (registre de destination restauré) mais elle aura eu un effet de bord sur le cache

Attaque *Meltdown* [13]

Exemple simple, suite

- En effet, l'accès mémoire va placer la ligne située à l'adresse `tableau[donnée * 4096]` dans le cache du processeur
- Il est possible pour un autre processus, via une attaque par canal auxiliaire contre le cache, de détecter ce chargement et de retrouver l'adresse accédée (et donc ici, la valeur de `donnée`)

Attaque *Meltdown* [13]

Application à la lecture de toute la mémoire

- Pour des questions de performance, dans de nombreux systèmes d'exploitation, le code et les données du noyau sont mappés dans l'espace d'adressage de chaque processus
- De même pour l'intégralité de la mémoire physique (sur les systèmes 64 bits)
- Ces adresses sont simplement marquées comme non accessibles depuis l'espace utilisateur au niveau de la MMU
- Donc si un processus essaie d'accéder à ces adresses, la MMU va déclencher une exception et le processus va être tué par le système d'exploitation

Attaque *Meltdown* [13]

Application à la lecture de toute la mémoire

- Exemple, cartographie mémoire sous Linux sur architecture x86_64 (cf. `Documentation/x86/x86_64/mm.txt`)

```
0000000000000000 - 00007fffffffffff (=47 bits) user space, different per mm
hole caused by [47:63] sign extension
ffff800000000000 - ffff87ffffffffff (=43 bits) guard hole, reserved for hypervisor
ffff880000000000 - ffffc7ffffffffff (=64 TB) direct mapping of all phys. memory
ffffc80000000000 - ffffc8ffffffffff (=40 bits) hole
ffffc90000000000 - ffffe8ffffffffff (=45 bits) vmalloc/ioremap space
ffffe90000000000 - ffffe9ffffffffff (=40 bits) hole
ffffea0000000000 - ffffeaffffffffff (=40 bits) virtual memory map (1TB)
... unused hole ...
ffffec0000000000 - fffffbffffffffff (=44 bits) kasan shadow memory (16TB)
... unused hole ...
fffffe0000000000 - fffffe7fffffffff (=39 bits) LDT remap for PTI
fffffe8000000000 - fffffeffffffffff (=39 bits) cpu_entry_area mapping
ffffff0000000000 - ffffff7fffffffff (=39 bits) %esp fixup stacks
... unused hole ...
ffffffef00000000 - ffffffffffffffff (=64 GB) EFI region mapping space
... unused hole ...
ffffffff80000000 - ffffffff9fffffff (=512 MB) kernel text mapping, from phys 0
ffffffffa0000000 - (fixmap start) ( 1526 MB) module mapping space (variable)
(fixmap start) - ffffffffff5fffff kernel-internal fixmap range
ffffffffff600000 - ffffffffff600fff (=4 kB) legacy vsyscall ABI
ffffffffffe00000 - ffffffffffffffff (=2 MB) unused hole
```

Attaque *Meltdown* [13]

Application à la lecture de toute la mémoire

```
; rcx = kernel address
; rbx = probe array
retry:
  mov al, byte [rcx]   ; RAX[7:0] = RAM[RCX] -> Exception
  shl rax, 0xc         ; RAX = RAX * 4096
  jz retry             ; if RAX == 0 goto retry
  mov rbx, qword [rbx + rax] ; RAM[RBX+RAX] = RBX
```

Attaque *Meltdown* [13]

Application à la lecture de toute la mémoire

- Pendant que la première instruction s'exécute, les trois instructions suivantes (qui en dépendent) sont mises en attente
- Dès que le résultat de la lecture est disponible, ces trois instructions commencent à s'exécuter
- La première instruction déclenche une exception (accès invalide à une adresse noyau depuis l'espace utilisateur)
- Si la dernière instruction a le temps de commencer à s'exécuter, la ligne dont l'adresse est `RBX+RAX` est chargée dans le cache du processeur
- Or cette adresse dépend de la donnée lue par la première instruction

Attaque *Meltdown* [13]

Application à la lecture de toute la mémoire

- Certes, le processus est tué dans l'opération (il est possible sur certaines architectures d'empêcher l'exception sans pour autant permettre l'accès direct au registre impacté)
- Mais un processus complice peut récupérer l'adresse accédée via une attaque SCA contre le cache et donc en déduire la donnée lue dans la zone noyau
- Comme l'intégralité de la mémoire physique est mappée dans l'espace noyau, il est ainsi possible de lire le contenu de toute la RAM physique

Attaque *Meltdown* [13]

Solutions

- Au niveau logiciel, il est possible de limiter l'impact de cette attaque en arrêtant de mapper les structures du noyau dans l'espace utilisateur
- KAISER [9]
- Impact non négligeable sur les performances car nécessité de modifier les tables de page à lors des basculements espace utilisateur / espace noyau

Attaque *Spectre* [12]

- Dans les différentes variantes de l'attaque *Spectre*, les mécanismes de prédiction de branchement du processeur sont visés ainsi que l'exécution spéculative qui en résulte
  - Lors d'une instruction de branchement conditionnel, le processeur fait une hypothèse sur le résultat de ce branchement et commence à exécuter spéculativement les instructions suivantes
  - Lors d'une instruction de saut, le processeur prédit également l'adresse de destination du saut
- Ces deux mécanismes fonctionnent avec apprentissage et cet apprentissage peut être utilisé par l'attaquant pour forcer l'exécution spéculative d'une portion de code qui ne devrait pas l'être
- Cette exécution spéculative va laisser des traces, notamment au niveau du cache, qui vont pouvoir être exploitées comme pour l'attaque *Meltdown*

Attaque *Spectre* [12]

Variante 1

- Dans le code du processus visé (exemple tiré de [12])

```
if (x < array1_size)
    y = array2[array1[x] * 256]
```

- Hypothèses
  - `x` est contrôlable par l'adversaire et est choisi (hors des limites du tableau) de manière à ce que `array1[x]` pointe vers une valeur secrète *k*
  - `array1_size` et `array2` ne sont pas présents dans le cache mais *k* l'est
  - Lors des exécutions précédentes du test, la valeur de `x` fournie était correcte et donc le prédicteur de branchement va considérer que le résultat du test sera probablement vrai

Attaque *Spectre* [12]

Variante 1

- Lors de l'exécution avec la valeur de `x` malicieuse, `array_size` n'étant pas dans le cache, le processeur se base sur la prédiction de branchement et se met à exécuter spéculativement l'instruction suivante en attendant la lecture de `array_size`
- La valeur de `array1[x]` (k) étant connue, le processeur peut commencer tout de suite la lecture à l'adresse `array2[array1[x] * 256]`, qui n'étant pas dans le cache, sera rapatriée depuis la mémoire
- Lorsque le résultat du test est finalement connu, l'effet de la seconde instruction est annulé
- Néanmoins, une ligne de cache dont l'adresse dépend de la valeur secrète k aura été chargée dans le cache et donc une attaque SCA contre ce dernier permettra de retrouver k

Attaque *Spectre* [12]

Variante 2

- L'adresse de destination d'un saut n'est pas toujours connue (ex. instruction `jmp [eax]`)
- Si la détermination de la destination du saut est retardée par un défaut de cache, le processeur va utiliser le prédicteur de branchement pour la deviner et exécuter spéculativement les instructions
- Si ce mécanisme peut être entraîné malicieusement par l'adversaire, ce dernier peut s'en servir pour faire exécuter spéculativement des instructions à des adresses qu'il a choisies (similaire au ROP)
- L'adversaire choisi ces instructions de telle sorte à ce qu'elles fassent fuir de l'information sensible via le cache

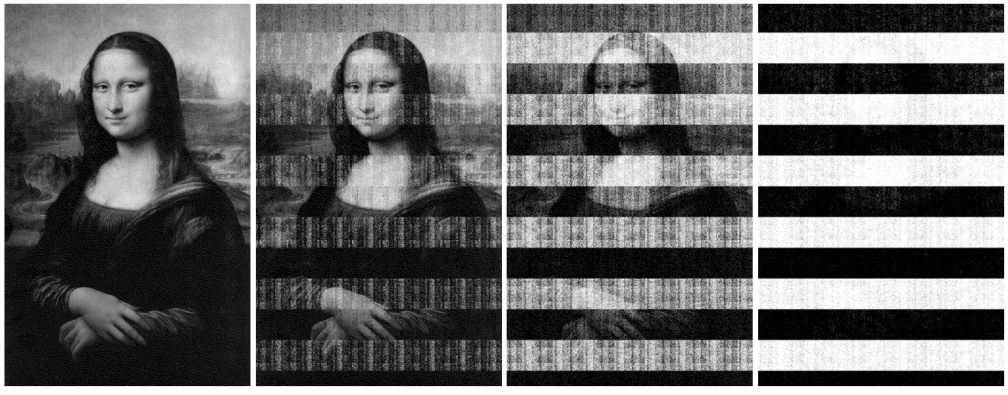
Plan

Espionnage du bus mémoire

- Si un adversaire peut espionner le bus mémoire (ou la mémoire directement), il peut
  - Récupérer le code exécuté
  - Récupérer les données manipulées par l'application
  - Perturber le code ou les données d'une application
- L'espionnage de la mémoire peut être plus ou moins intrusif en fonction des systèmes

Espionnage de la mémoire

Exemple : Cold Boot Attack [10]



- Lors d'une interruption d'alimentation, les données stockées dans une mémoire DRAM ne disparaissent pas instantanément (rémanence)
- Plus la température est basse, plus cette rémanence est importante
- Cette propriété peut être utilisée pour voler des données en RAM (exemple : clés de chiffrement de disque dur dans le contexte *evil maid*)

Plan

Plan

Problématique générale

Attaques DMA

- Pour des raisons de performance, les périphériques ont souvent un accès direct à la mémoire centrale
- Cet accès se fait principalement par l'intermédiaire du mécanisme de DMA (*Direct Memory Access*)
- L'accès contourne en général (sauf architectures disposant d'IOMMU : Intel VT-d par exemple) la MMU et donc les restrictions d'accès mises en place par l'OS
- Si l'adversaire arrive à prendre le contrôle d'un périphérique, il peut alors accéder (lecture / écriture) à toute la mémoire, y compris les zones mémoires occupées par l'OS

Exemple : Carte réseau moderne

- Elle fait plus que connecter un ordinateur au réseau
- Architecture matérielle complexe, presque un ordinateur à elle seule
  - Processeur
  - Mémoire
- Fait tourner un logiciel embarqué (*firmware*) qui fait beaucoup de choses
  - Soulage le processeur principal (calcul des sommes de contrôle, gestion de la fragmentation TCP...)
  - Gère la partie réseau avec parfois des contraintes temps réel (IEEE 802.11, 2/3G...)
  - Gère, sur les modèles haut de gamme, de l'administration à distance

Exemple : Carte réseau moderne

Impacts sur la sécurité

- Si un adversaire arrive à exécuter du code arbitraire sur la carte, il peut faire
  - Déni de service (suppression de tous les paquets ou de certains paquets)
  - Empoisonnement de cache ARP et/ou DNS
  - Et surtout : attaquer l'hôte via des accès en lecture ou écriture vers la mémoire principale

Exemple : Carte réseau moderne

Problème

- Sur certaines cartes, des failles de sécurité on été découvertes [6] dans certaines cartes réseau au niveau du traitement des paquets RMCP (*Remote Management and Control Protocol*)
- $\rightsquigarrow$Il était possible, à distance, d'injecter et de faire exécuter du code par la carte réseau cible...
- Au delà de cet faille, la complexité des firmwares augmentant, ce risque augmente également...

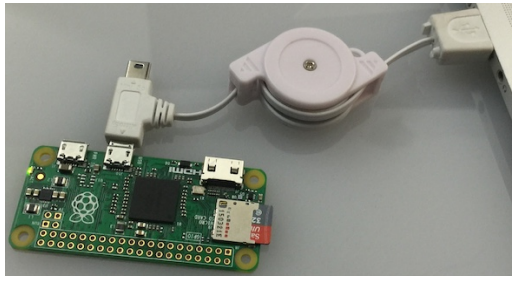
Plan

Périphériques USB malicieux

- Les périphériques USB sont un peu trop “plug and play”...
- Il est assez simple de faire passez un périphérique USB un peu intelligent (smartphone ou clé USB trafiquée) pour un autre périphérique [3]
  - Faire passer le périphérique pour un clavier par exemple et envoyer une combinaison de touches particulière (par exemple le raccourci clavier de la fonction *Exécuter* puis le nom d’une commande et la touche *Entrée*)

Périphériques USB malicieux

PoisonTap [1]



- Une fois branché sur le port USB, il se fait passer pour une interface réseau (*Ethernet over USB*)
- Il répond aux requêtes DHCP envoyées par l'ordinateur (automatiquement) en indiquant que l'ensemble de l'espace d'adressage IPv4 fait partie de son réseau local
- Tout le trafic réseau de l'ordinateur est alors redirigé automatiquement vers le PoisonTap (le trafic local est prioritaire)

Périphériques USB malicieux

PoisonTap [1]

- Si un navigateur fait un accès HTTP (par exemple un navigateur qui fonctionne en arrière plan), cet accès est intercepté par le PoisonTap
- La réponse HTTP est remplacé par une série d'*i-frames* vers de nombreux sites web connus
- Cela permet de voler les *cookies* d'authentification pour ces sites (y compris les cookies des sites utilisant HTTPS si le flag *Secure* n'est pas correctement positionné)
- Installe également une porte dérobée (*websocket* ouverte)...
- Facilement exploitable dans le contexte *evil maid* (PC verrouillé laissé sans surveillance)

Périphériques USB malicieux

USB killer [2]

- Petit périphérique USB ayant la forme d'une clé USB normale
- Charge des capacités à partir de l'alimentation 5 V de l'USB jusqu'à atteindre une tension de plusieurs centaines de volts
- Décharge ensuite ces capacités sur les lignes de données du bus USB
- Répète automatiquement l'opération jusqu'à ce que l'hôte soit grillé...
- Grille complètement de nombreux systèmes ayant des ports USB accessibles (ordinateurs, téléphones, voitures, etc.)

Clavier

- Certains claviers disposent de micro-contrôleurs et peuvent être reprogrammés...
- Exemple de certains clavier Apple [4]
  - Modifier le comportement du clavier (y compris déni de service)
  - Injection de frappes
  - Keylogger

Et la batterie ?

- À des fins de sûreté, les batteries actuelles embarquent de l'intelligence
  - *Safety is a primary design goal in the Smart Battery System specifications. The central concept behind the Smart Battery specifications is locating the primary intelligence of the system inside the battery pack itself* (Smart Battery System Specifications)
- Petit processeur s'occupant du contrôle de la batterie (profil de charge, protection en température, protection contre la décharge profonde, communication des paramètres de charge à l'OS, etc.)

Batterie (suite)

- Exploitation des batteries des MacBook par [14]
  - Mot de passe simple (par défaut) pour passer le contrôleur de batterie en mode *unsealed* puis en mode *full access*
  - Modification du firmware du contrôleur de batterie...
- Exploitation
  - Déni de service de la batterie voire explosion/incendie
  - Attaque de l'OS (nécessite une vulnérabilité dans l'OS) persistante au redémarrage/réinstallation

Thunderbolt

- Thunderstrike : `https://trmm.net/Thunderstrike_31c3`
- Rootkit très bas niveau (via modification de l'EFI) visant les Mac mis en place par un périphérique Thunderbolt malicieux
- Lors du démarrage, le BIOS (et l'UEFI) vont chercher des OptionROM (mécanisme remontant à l'IBM PC), mémoires contenant du code à exécuter et présentes sur des cartes ISA, PCI, PCI-X et... des périphériques Thunderbolt
- Exécution de code privilégié, contexte : *evil-maid*
- Dans le cas de Thunderstrike, les OptionROM sont également exécutée lors des mises à jour du firmware de boot et donc à un moment où l'accès à la flash est autorisée

Plan

Conclusion

- Se méfier du matériel lui-même
- Se méfier du fabriquant du matériel
- Se méfier du propriétaire du matériel
- $\rightsquigarrow$Mais à qui se fier ?

Troisième partie III

Sécurité OS

Plan

Plan

Rôle dans la sécurité

- Le système d'exploitation joue un rôle primordial dans la sécurité d'un ordinateur
- C'est lui qui est garant de l'isolation entre les processus (applications) et entre les utilisateurs
- Il s'appuie pour cela sur des fonctionnalités offertes par le matériel (MMU et niveaux d'exécution du processeur notamment)

Plan

Plan

Anneaux de protection

Protection rings

- Concept introduit dans le matériel développé pour le système *Multics* (8 anneaux utilisés)
- Mécanisme matériel offrant une certaine protection des données contre des fautes ou des comportements malicieux du logiciel
- Différentes parties du logiciel (système d'exploitation, pilotes de périphériques, bibliothèques, applications, etc.) vont s'exécuter dans des anneaux différents avec des privilèges différents

Anneaux de protection

Protection rings

- Hiérarchie entre les anneaux
  - L'anneau le plus privilégié (généralement numéroté 0) a accès à toutes les fonctionnalités et les données du système
  - Chaque anneau à accès aux fonctionnalités et aux données des anneaux moins privilégiés
  - Par contre, dans un anneau donné, l'accès aux fonctionnalités et aux données de niveaux plus privilégiés doit passer par des interfaces bien spécifiées (appels systèmes...)

Plan

Introduction

- Mécanisme introduit dans les années 1960 (monde des *mainframes*) mais disponible sur les PC depuis le 80386 d'Intel
- Avantages
  - Isolation entre les espaces mémoires de différentes applications
  - Partage explicite de certaines zones mémoires (bibliothèques de code, données partagées...)
  - Pagination (permet aux applications de disposer de plus de mémoire qu'il en existe physiquement)

Principes

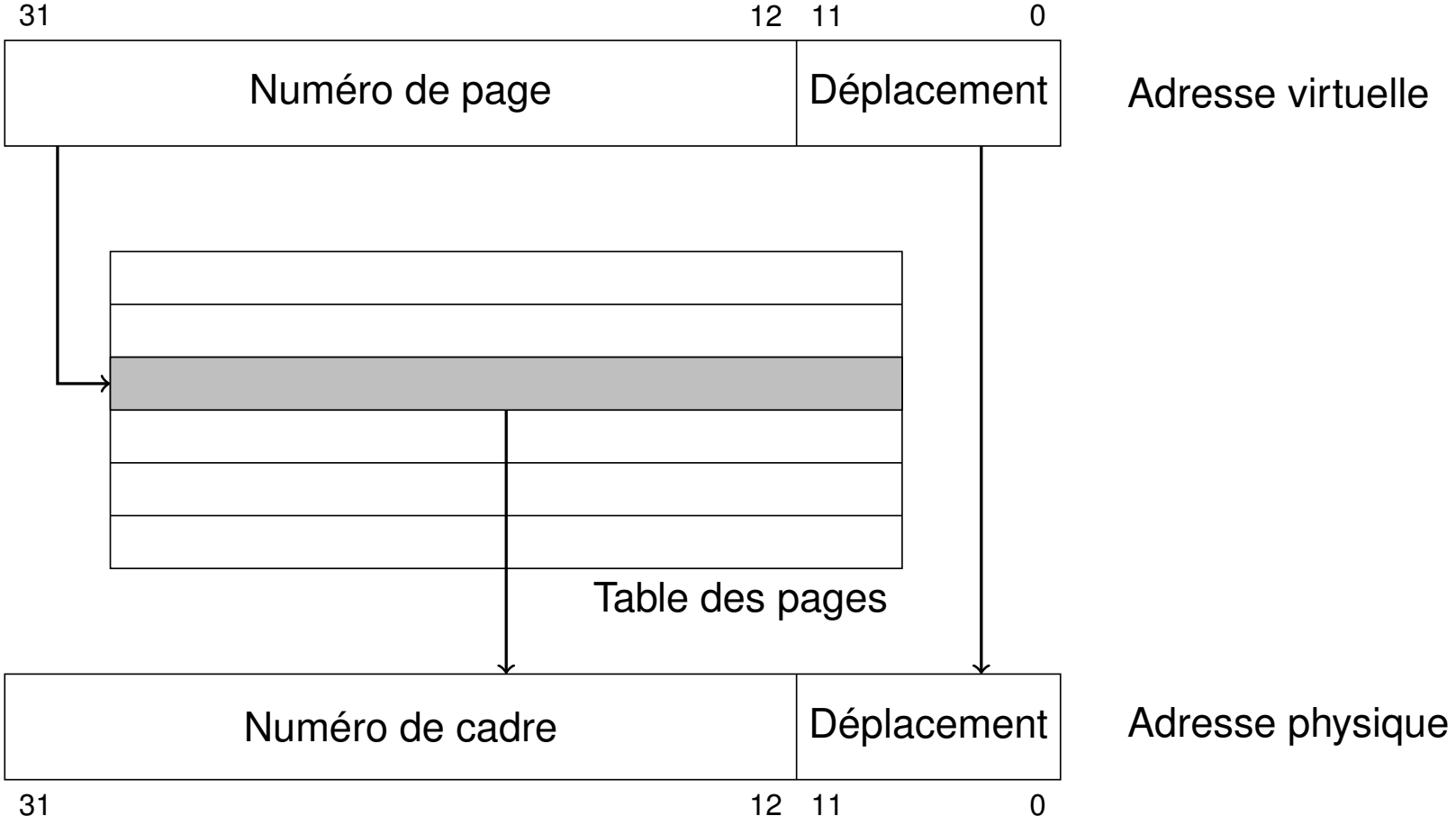
- Les applications ne manipulent pas directement des adresses *physiques* (adresses présentées aux puces de RAM) mais des adresses *virtuelles*
- Un mécanisme matériel, la MMU (*Memory Management Unit*), le plus souvent intégrée aux processeurs, se charge de faire la conversion entre les adresses virtuelles et les adresses physiques
- Cette conversion est effectuée en se basant sur une table de conversion, nommée la table des page (*page table*), le plus souvent mise en place par le système d'exploitation

Fonctionnement

- Pour des raisons pratiques, la traduction n'est pas réalisée avec la granularité d'une adresse virtuelle (il y a 2^{32} adresses virtuelles dans le cas d'un processeur 32 bits, donc une table des pages nécessiterait de stocker les 2^{32} adresses physiques correspondantes ; chaque adresse étant codée sur 32 bits, la table ferait 16 Gio...)
- L'espace d'adressage virtuel ainsi que l'espace d'adressage physique sont découpés en *pages* (au minimum 4 kio sur `x86`)
- La conversion fait correspondre un numéro de cadre physique à un numéro de page virtuelle, le déplacement au sein de la page physique étant égal au déplacement au sein de la page virtuelle

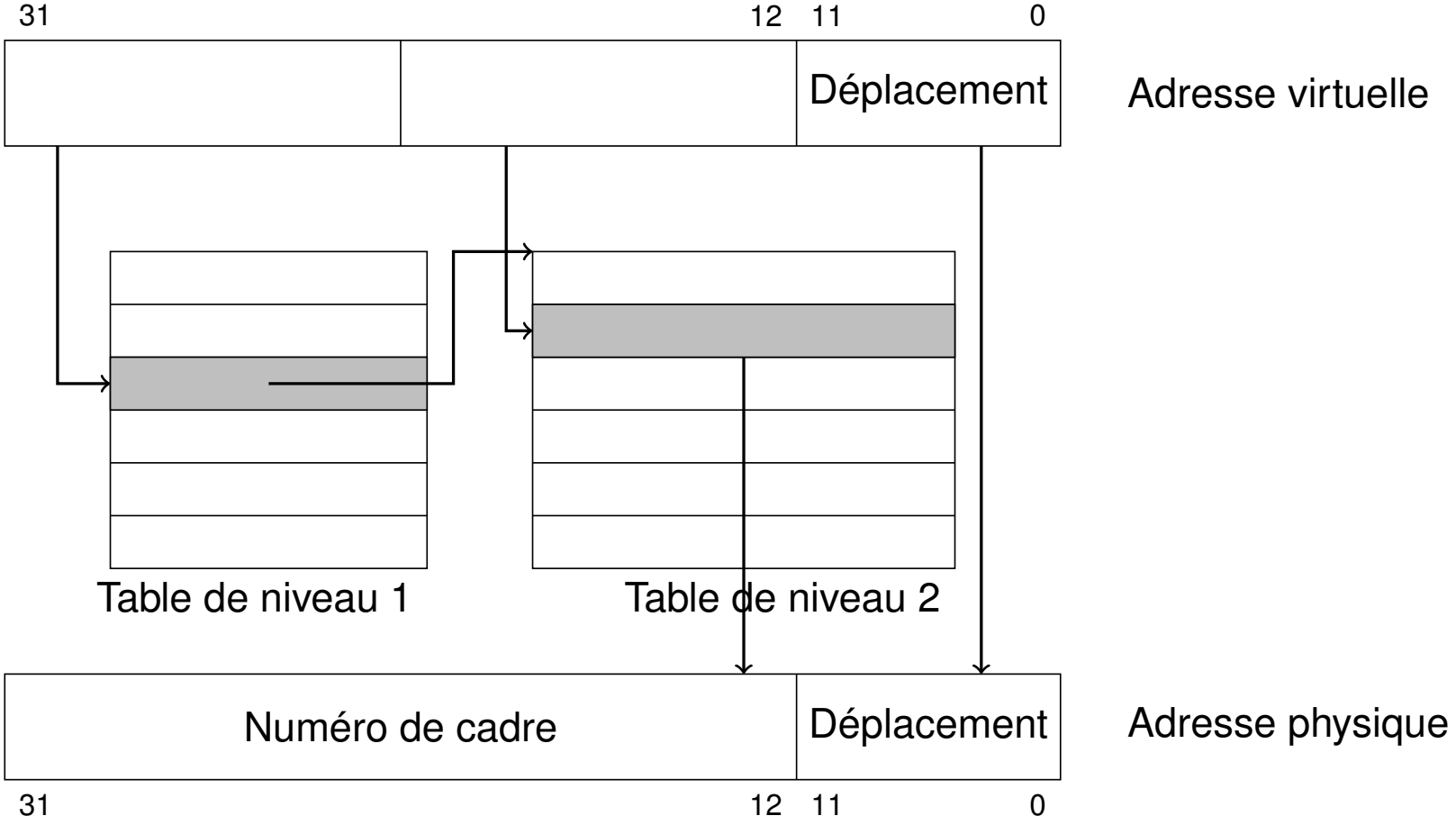
Fonctionnement

Exemple : adresses 32 bits, pages de 4 kio, 1 niveau de traduction



Fonctionnement

Exemple : adresses 32 bits, pages de 4 kio, 2 niveaux de traduction



Fonctionnement

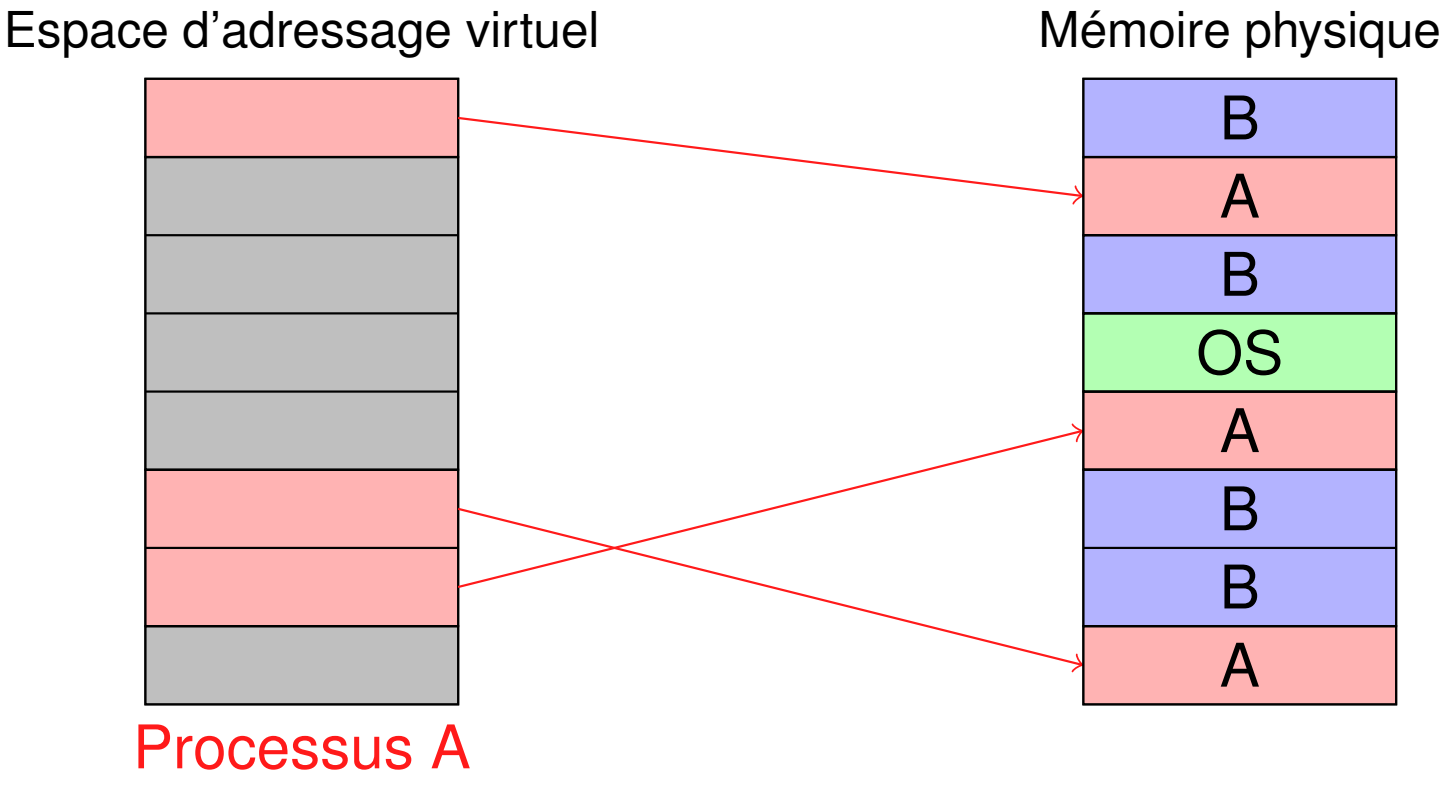
- Utilisation de plusieurs niveaux de tables de pages pour gérer plus efficacement les larges zones vides dans l'espace d'adressage virtuel d'un processus
- La MMU dispose d'un cache spécialisé (TLB, *Translation Lookaside Buffer*) destiné à conserver les traductions (page virtuelle, cadre physique) récemment utilisées
- En cas de défaut de TLB, le parcours de la table de pages (ou des différents niveaux de table) pour rechercher la traduction peut être effectué
  - Soit matériellement : la MMU connaît l'adresse de la table de premier niveau et va la parcourir elle-même (exemple architecture `x86`)
  - Soit logiciellement : la MMU déclenche une exception et c'est au système d'exploitation de charger explicitement une entrée du TLB avec la traduction demandée (exemple architecture MIPS)

Fonctionnement

- Dans une entrée de la table de pages (entrée de page, *page entry*) sont également stockées les informations suivantes
  - Validité (l'entrée est-elle valide ?)
  - Informations d'accès (les données de la page ont-elles été accédées et/ou modifiées ?)
  - Informations de protection
    - Page en lecture seule ou lecture/écriture
    - Accès en mode utilisateur ou en mode superviseur uniquement
    - Page contenant du code exécutable ou non (bit NX)

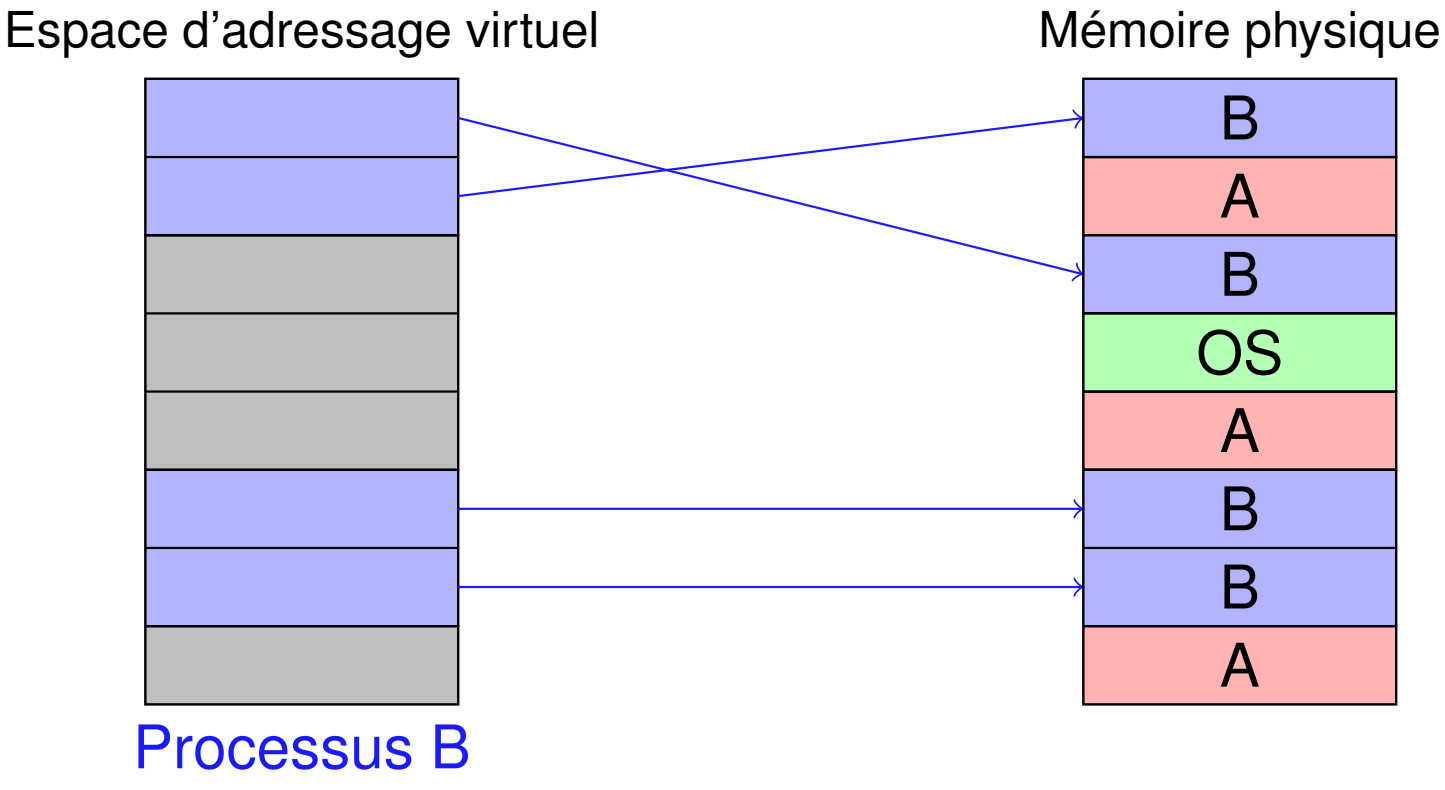
Utilisation

Flat memory model



Utilisation

Flat memory model



Utilisation

- Le système d'exploitation maintient une table de pages par processus
- Lorsque le SE bascule d'un processus à un autre, il configure la MMU pour utiliser la table de pages du nouveau processus
- La modification du registre de la MMU contenant le pointeur vers le premier niveau de la table de pages est une action qui est privilégiée (qui ne peut donc être réalisée que par le SE)
  - Lien avec les anneaux de protection
- Un processus ne peut donc pas modifier directement sa table de pages

Utilisation

- Lorsqu'un processus accède à une adresse virtuelle pour laquelle l'entrée correspondante dans la table des pages n'est pas valide (drapeau valide non positionné) ou si les informations de protection ne sont pas correctes (lecture seule pour un accès en écriture par exemple), la MMU déclenche une exception
- Le système d'exploitation reprend alors la main et peut
  - Tuer le processus en cas d'accès illégal (accès à une page mémoire non allouée, violation des attributs de protection...)
  - Dans certain cas, l'exécution peut reprendre après une action du système d'exploitation (restauration de la page demandée qui avait été évincée de la mémoire faute de place, mécanisme de *copy-on-write*...)

Utilisation

Isolation des processus

- Un processus ne peut donc pas accéder à une adresse physique si une traduction n'est pas explicitement mise en place par le système d'exploitation
- Donc le SE peut s'assurer qu'un processus ne peut pas accéder à des zones mémoires physiques appartenant à d'autres processus ou au SE lui-même

Utilisation

Pagination / *Swap*

- Si la mémoire vient à manquer, le système d'exploitation peut déplacer le contenu de certains cadres de mémoire physique vers un support de stockage de masse (disque dur) et marquer les pages de mémoire virtuelle correspondantes comme invalides
- Si un des processus dont la mémoire a été évincée essaie d'y accéder, la MMU déclenchera une exception
- Le SE reprend la main et voit que l'adresse demandée existe bien mais a été déplacée vers le disque
- Il peut remettre les données dans un cadre de mémoire physique et reconfigurer l'entrée de page correspondante
- Il relance alors l'instruction fautive du processus qui peut maintenant réussir

Plan

Plan

3 méthodes fondamentales

- Ce que l'on connaît
  - Mot de passe
  - Code PIN
- Ce que l'on possède
  - Clé
  - Badge
  - Carte à puce
- Ce que l'on est
  - Visage (photo d'identité)
  - Biométrie : empreinte digitale, image de la rétine, veines, etc.

Mots de passe

- Moyen le plus utilisé pour identifier un utilisateur
- Principe
  - L'utilisateur et le système se mettent d'accord sur un secret partagé P, le plus souvent une chaîne de caractères plus ou moins longue
  - Lorsque l'utilisateur veut s'authentifier, il présente P au système
  - Le système compare le secret présenté par l'utilisateur à sa copie locale
- Pour éviter les attaques par force brute (essai d'un grand nombre de mot de passe)
  - Bloquer le compte au bout de x essais infructueux (mais risque de déni de service)
  - Augmenter progressivement le temps minimum entre deux tentatives infructueuses

Mots de passe

Stockage en clair

- Le système doit pouvoir vérifier si un mot de passe présenté est correct
- Première solution : le système stocke le mot de passe en clair
  - Problème : si un intrus (ou un administrateur corrompu) accède à cette information, il peut récupérer le mot de passe en clair et s'en servir

Mots de passe

Stockage

- Solution : stocker une “empreinte” du mot de passe et pas le mot de passe lui-même
  - Soit h une fonction de hachage
  - Initialisation : l'utilisateur choisit P et le système stocke $H = h(P)$
  - Identification : l'utilisateur présente P' au système, qui calcule $H' = h(P')$ et compare avec le haché H stocké lors de la phase d'initialisation
  - Si $H = H'$, $P = P'$ (avec une très grande probabilité), sinon $P \neq P'$
- Dans ce cas un intrus a accès à H et doit trouver X tel que $H = h(X)$, ce qui est très difficile pour une bonne fonction de hachage (propriété de résistance à la première pre-image)

Mots de passe — Stockage

Attaque par dictionnaire

- Même avec cette solution, on peut identifier que deux mots de passe sont identiques
- On peut aussi construire des tables des mots de passe les plus fréquemment utilisés (attaque par dictionnaire)
  - Soit $E = P_1, \ldots, P_n$ une liste de mots de passe simples (mots du dictionnaire, dates de naissance, numéro d'immatriculation de voitures, prénoms, etc.)
  - L'attaquant calcule $\forall i \in [1..n], H_i = h(P_i)$ (ce calcul peut être fait hors ligne)
  - Si dans le fichier il trouve un des H_i qu'il a calculé, il peut savoir que le mot de passe correspondant est P_i
  - Cependant la taille de cette liste est proportionnelle à n, d'autres techniques existent...

Mots de passe — Stockage

Chaînes de hash

- Permet de réduire la taille nécessaire au stockage des tables pre-calculées au détriment du temps nécessaire pour les utiliser
- Soit P l'ensemble (fini) des mots de passe, h la fonction de hachage utilisée pour le stockage et r une fonction qui prend en entrée un haché et qui sort un élément de P ($r \neq h^{-1}$)
- On calcule des chaînes de longueur fixe n en partant d'un élément de P et en appliquant alternativement h et r
  - $\texttt{aaaaaa} \xrightarrow{h} \texttt{0A1B2C3D} \xrightarrow{r} \texttt{qsdfgh} \xrightarrow{h} \texttt{AA5512FF} \xrightarrow{r} \texttt{wxcvbn}$
  - $\texttt{bbbbbb} \xrightarrow{h} \texttt{11223344} \xrightarrow{r} \texttt{poiuyt} \xrightarrow{h} \texttt{9D55CC32} \xrightarrow{r} \texttt{mlkjhg}$
- On stocke la première et la dernière valeur de chaque chaîne

Mots de passe — Stockage

Chaînes de hash, suite...

- Si on récupère un mot de passe haché, ex. `9D55CC32`, on va appliquer alternativement r et h, au maximum n fois, et vérifier si on trouve une valeur stockée dans notre table
  - `9D55CC32` $\xrightarrow{r}$ `mlkjhg`
  - `mlkjhg` est dans notre table
- On repart alors du début de la chaîne en appliquant h et r jusqu'à retrouver notre mot de passe haché
  - `bbbbbb` $\xrightarrow{h}$ `11223344` $\xrightarrow{r}$ `poiuyt` $\xrightarrow{h}$ `9D55CC32`
  - Ici notre mot de passe en clair est `poiuyt`
- Problème : Fusion de deux chaînes
  - Mot de passe haché : `00110011`
  - `00110011` $\xrightarrow{r}$ `fghjkl` $\xrightarrow{h}$ `55665566` $\xrightarrow{r}$ `mlkjhg`
  - La réponse n'est pas dans la chaîne `bbbbbb`–`mlkjhg`

Mots de passe — Stockage

Rainbow tables

- Pour réduire le risque de collision, on remplace la fonction de réduction r par une série de fonctions $r_1, \ldots, r_k$ ainsi pour qu'une collision se propage, il faut qu'elle intervienne lors de la même itération
  - $\texttt{aaaaaa} \xrightarrow{h} \texttt{0A1B2C3D} \xrightarrow{r_1} \texttt{qsdfgh} \xrightarrow{h} \texttt{AA5512FF} \xrightarrow{r_2} \texttt{wxcvbn}$
  - $\texttt{bbbbbb} \xrightarrow{h} \texttt{11223344} \xrightarrow{r_1} \texttt{poiuyt} \xrightarrow{h} \texttt{9D55CC32} \xrightarrow{r_2} \texttt{mlkjhg}$
- La vérification implique le calcul de plusieurs chaînes
  - $\texttt{98765432} \xrightarrow{r_2} \texttt{aqwzsx?}$
  - $\texttt{98765432} \xrightarrow{r_1} \texttt{edcrfv} \xrightarrow{h} \texttt{77552211} \xrightarrow{r_2} \texttt{rfvtgb?}$
- Des rainbow tables sont disponibles (sur Internet) pour plusieurs fonctions de hachage de mots de passe

Mots de passe — Stockage

Attaque par force brute...

- Toutes ces techniques permettent d'attaquer plus rapidement les mots de passe
- Cependant, l'attaque par force brute devient de nouveau possible à l'aide du calcul massivement parallèle (notamment sur GPU)
- Exemple [8]
  - Cluster de 25 cartes graphiques AMD Radeon
  - NTML (ancien algorithme de hachage de Microsoft) : $348 \cdot 10^9$ essais par seconde, soit un peu plus de 5 heures pour tester tous les mots de passe à 8 caractères (95^8 essais)
  - SHA1 : $63 \cdot 10^9$ essais par seconde
  - bcrypt : seulement $71 \cdot 10^3$ essais par seconde

Mots de passe — Stockage

Attaque par force brute...

Source : [8]

Mots de passe — Stockage

Sel

- Solution : ajouter du “bruit” (sel) au calcul
  - Initialisation : l’utilisateur choisit P, le système choisit aléatoirement S et stocke $S, H(P\|S)$
  - Identification : l’utilisateur présente P' au système qui récupère S et calcule $H(P'\|S)$
  - L’attaquant, s’il veut faire le calcul de ses tables hors ligne, doit considérer toutes les valeurs possibles du sel S, ce qui peut augmenter considérablement la taille de l’espace nécessaire pour les stocker

Mots de passe — Stockage

Key stretching

- Autre solution : accroître le temps et/ou le matériel nécessaire pour calculer la fonction de hachage du mot de passe

```
result = ""
for 1 to 65536 do
   result = h(result + password + salt)
```

- La vérification d'un mot de passe reste relativement rapide (on peut par exemple admettre 1 seconde au lieu de 1 μs) mais le calcul des rainbow tables prend 1 000 000 fois plus de temps

Mots de passe — Stockage

Key stretching, suite...

- Cette technique est également utilisé pour générer des clés relativement robustes à partir de mots de passe simples (exemple pour le WPA)
- Exemples : PBKDF2 (Password-Based Key Derivation Function 2), bcrypt
- L'utilisation de FPGA réduit l'efficacité de ce mécanisme, d'où l'introduction d'algorithme nécessitant en plus un matériel (plus précisément un usage mémoire) important pour le calcul (exemple : l'algorithme scrypt)

Mots de passe à usage unique

- Problème lorsque l'on doit taper le mot de passe sur un terminal non digne de confiance ou s'il doit transiter en clair sur le réseau
- $\rightsquigarrow$Risque d'interception et de réutilisation
- Solution : le mot de passe à usage unique (*One-Time Password*)
- Le mot de passe à utiliser change à chaque demande d'authentification et l'interception d'un mot de passe ne donne aucune information sur les futurs mots de passe à utiliser

Mots de passe à usage unique

S/Key

- L'utilisateur choisit un mot de passe p_0 et le partage avec le serveur
- L'utilisateur et le serveur calculent $p_1 = H(p_0), p_2 = H(p_1) = H(H(p_0)), \ldots, p_n = H^n(p_0)$ et le serveur stocke p_n
- Lors de la première authentification, l'utilisateur doit fournir p_{n-1} au serveur. Ce dernier calcule $H(p_{n-1})$ et compare le résultat à p_n
- Si la comparaison réussie, le serveur stocke p_{n-1} et l'utilisateur devra fournir à l'authentification suivante p_{n-2}...

S/Key

Sécurité

- En espionnant, un adversaire voit passer p_i
- Pour réussir l'authentification suivante, il doit trouver p_{i-1} tel que $p_i = H(p_{i-1})$, ce qui est difficile (propriété de résistance à la première pre-image)
- Néanmoins, S/Key est vulnérable à une attaque *Man-in-the-middle*

Autres considérations

- Un bon mot de passe est un mot de passe compliqué (comportant des lettres, des chiffres, des majuscules, des caractères spéciaux)
  - Or, un utilisateur, aura tendance à choisir un mot de passe relativement simple pour s'en souvenir, ou bien, il le notera par écrit
- De plus, l'utilisateur aura tendance à utiliser un seul mot de passe pour ses différents comptes (professionnels, comptes personnels sur Internet, etc.)
  - Il suffit d'une interception pour permettre à un attaquant d'accéder à l'ensemble de ses comptes...
  - Solutions
    - Racine commune + prefixe/sufixe propre à chaque compte
    - Gestionnaire de mots de passe
    - H(mot de passe unique, nom du compte)

Autres considérations

- Se méfier des procédures de récupération de mot de passe qui sont parfois plus simples que de trouver le mot de passe
  - Questions secrètes trop simples (nom du chien, ville de naissance...), soit la réponse peut être facilement connue de l'attaquant, soit l'espace de recherche est restreint
  - Administrateur système qui ne vérifie pas l'identité de la personne qui vient lui demander de réinitialiser son mot de passe
- Sondage au Royaume-Uni en 2004 (`http://news.bbc.co.uk/2/hi/technology/3639679.stm`)
  - 70 % des sondés donneraient leurs mots de passe en échange d'une barre de chocolat
  - 34 % les donneraient même sans rien en échange
  - Espérons que les mentalités évoluent...

Mots de passe les plus utilisés

Source : SplashData, 2014

- 123456
- password
- 12345
- 12345678
- qwerty
- 1234567890
- 1234
- baseball
- dragon
- football

Plan

Plan

DAC vs. MAC

- Les permissions Unix standards et les ACL
  $\rightsquigarrow$ *Discretionary Access Control*
- Problèmes
  - L'administrateur ne contrôle pas les utilisateurs : ces derniers peuvent définir les permissions qu'ils veulent sur leurs fichiers (ex. permissions 777 sur leur répertoire `~/.ssh`)
  - Les processus héritent des droits de l'utilisateur et peuvent faire n'importe quoi (lire des fichiers privés, modifier les permissions, etc.)
- Solution : *Mandatory Access Control*
  - N'accorder aux processus que les privilèges nécessaires
  - Tout ce qui n'est pas autorisé est interdit

Quelques implémentations de MAC dans Linux

- SELinux
  - Crée par la NSA
  - Inclus dans le noyau 2.6+
  - Supporté par de nombreuses distribution Linux
  - Chaque fichier (au sens large) et processus a un contexte (utilisateur, rôle, type)
  - Toute action d'un processus sur un fichier (au sens large : fichier, périphérique, socket, etc.) va déclencher une vérification par le noyau de la présence d'une règle qui l'autorise (en fonction du contexte du processus et du fichier)
- TOMOYO Linux
- Smack (*Simplified Mandatory Access Control Kernel*)...
- Souvent complexes à configurer pour l'administrateur !

Plan

Introduction

- Très à la mode depuis quelques années, surtout côté serveur (mais ancien : années 60)
- Permet de faire fonctionner plusieurs machines virtuelles sur une machine physique
- Chaque machine virtuelle fait tourner un système d'exploitation et plusieurs applications
- Chaque machine virtuelle est isolée des autres (et de la machine physique)
- Exemple : sur un serveur physique, une machine virtuelle par service (bénéfique d'un point de vue sécurité)
- Autre avantage : migration d'une machine physique vers une autre

Sécurité

- Hyperviseur + Support matériel
- Problèmes de sécurité
  - Bug possible dans l'hyperviseur (exemple, CVE-2014-7188 dans Xen, octobre 2014)
  - Périphériques matériels non prévus pour la virtualisation (notamment ceux écrivant directement en mémoire comme le DMA, carte graphique...)
    - Soit émulation par l'hyperviseur (peu performant)
    - Protection via un mécanisme de IO-MMU (ex. Intel VT-d)

Quatrième partie IV

Sécurité logicielle

Plan

Plan

Top 25 Most Dangerous Software Errors

- Source : 2019 CWE Top 25 Most Dangerous Software Errors, `http://cwe.mitre.org/top25/`
- Page intéressante à lire car apporte de nombreux détails sur chacune des vulnérabilités et sur les solutions possibles

Top 25 Most Dangerous Software Errors I

1. *Improper Restriction of Operations within the Bounds of a Memory Buffer*
2. *Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')*
3. Improper Input Validation
4. Information Exposure
5. Out-of-bounds Read
6. *Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')*
7. Use After Free
8. Integer Overflow or Wraparound
9. *Cross-Site Request Forgery (CSRF)*

Top 25 Most Dangerous Software Errors II

10. Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
11. Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
12. Out-of-bounds Write
13. Improper Authentication
14. NULL Pointer Dereference
15. Incorrect Permission Assignment for Critical Resource
16. Unrestricted Upload of File with Dangerous Type
17. Improper Restriction of XML External Entity Reference
18. Improper Control of Generation of Code ('Code Injection')
19. *Use of Hard-coded Credentials*

Top 25 Most Dangerous Software Errors III

20. Uncontrolled Resource Consumption
21. Missing Release of Resource after Effective Lifetime
22. Untrusted Search Path
23. Deserialization of Untrusted Data
24. Improper Privilege Management
25. Improper Certificate Validation

Plan

Plan

Introduction

- Erreur extrêmement répandue et dangereuse
- Liée à l'utilisation de langages ne vérifiant pas lors de l'exécution les limites des tableaux et des tampons
- En lisant ou en écrivant à un index en dehors des limites d'un tampon, on peut récupérer des informations non prévues ou perturber l'exécution du programme

Premier exemple

```c
/* test0.c */
#include <string.h>
#include <stdlib.h>

void my_function (char *request_from_user) {
  char buffer [16];
  strcpy (buffer, request_from_user);
  /* ... */
}

int main (int argc, char **argv) {
  my_function (argv[1]);
  exit (EXIT_SUCCESS);
}
```

Premier exemple — Suite

```
$ gcc -fno-stack-protector -o test0 test0.c
$ ./test0 12345
$ ./test0 12345678901234567890123456789O
Erreur de segmentation
$
```

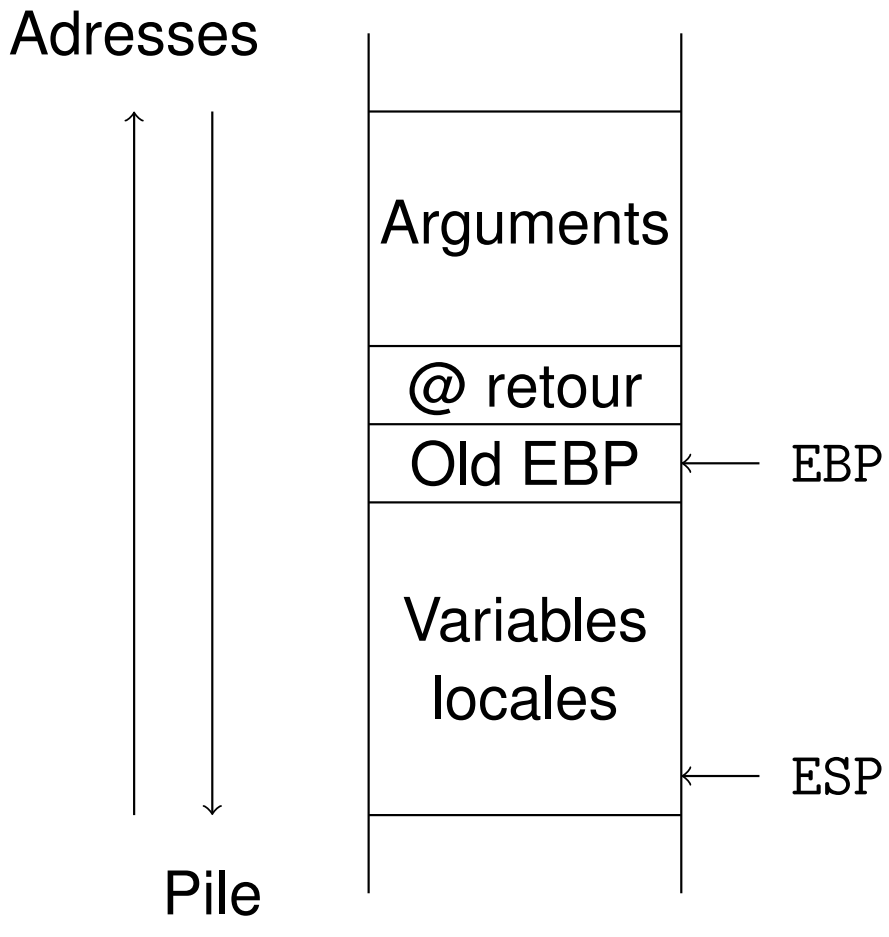
- Que s'est-il passé ?

Premier exemple — Explications

```
void my_function (char *request_from_user)
{
  char buffer [16];
  strcpy (buffer, request_from_user);
}
```

- `strcpy` copie la chaîne `request_from_user` (en incluant le 0 final) à l'adresse `buffer`
- `buffer` ne peut normalement recevoir que 16 octets au total or, dans le deuxième exemple, `request_from_user` contient 30 octets
- `strcpy` va donc écraser ce qui vient juste après `buffer` en mémoire, ce qui peut poser des problèmes

État de la mémoire (entrée d'une fonction)



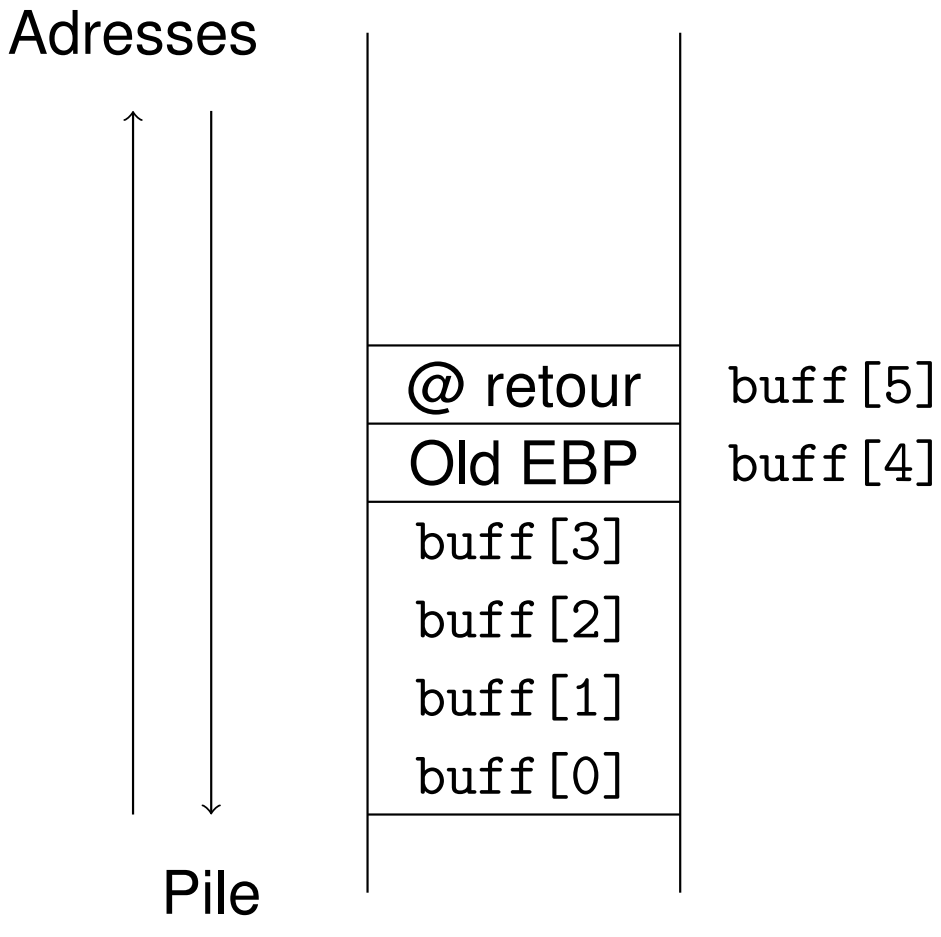
Modification de l'adresse de retour

```c
/* test1.c */
#include <stdio.h>
#include <stdlib.h>

void foo () {
  printf ("In foo!\n");
  exit (EXIT_FAILURE);
}

void bar () {
  int buff[4];
  buff[5] = (int)(foo);
}

int main (int argc, char **argv) {
  bar ();
  exit (EXIT_SUCCESS);
}
```

Modification de l'adresse de retour

```
$ gcc -g -m32 -mtune=i386 -o test1 test1.c
$ ./test1
In foo!
$
```

Modification de l'adresse de retour



Exploitation

- L'exploitation la plus courante est d'exploiter le débordement d'un tampon situé sur la pile (variable locale) pour modifier l'état de celle-ci, et notamment l'adresse de retour, et ainsi détourner le flot d'exécution du programme
  - Saut vers un morceau de code (par exemple *shellcode*) fourni par l'attaquant (dans le même tampon)
  - Saut vers une fonction de la bibliothèque standard (souvent `system` ou `execve`) en ayant au préalable configuré la pile pour passer à cette fonction des paramètres intéressants (*return to libc*)
  - Chaîne de sauts vers des portions de code (quelques instructions suivies d'une instructions `return`) : *Return Oriented Programming*
- Résultat : Exécution de code avec les privilèges de l'application de départ

Protection

- Éviter un certain nombre de fonctions dangereuses (`strcpy`, `gets`...) et utiliser les fonctions *n* (`strncpy`...)
- Utilisation de valeurs *canaries* permettant de détecter une tentative de corruption de la pile (ex. StackGuard...)
- Pile non exécutable (via bit NX par exemple) mais ne protège pas contre les attaques *return to libc*
- ABI x86_64 : le premier argument d'une fonction est passé dans un registre et non dans la pile
- *Address Space Layout Randomization*
- Langages vérifiant dynamiquement le dépassement des tampons (Java, C#...)

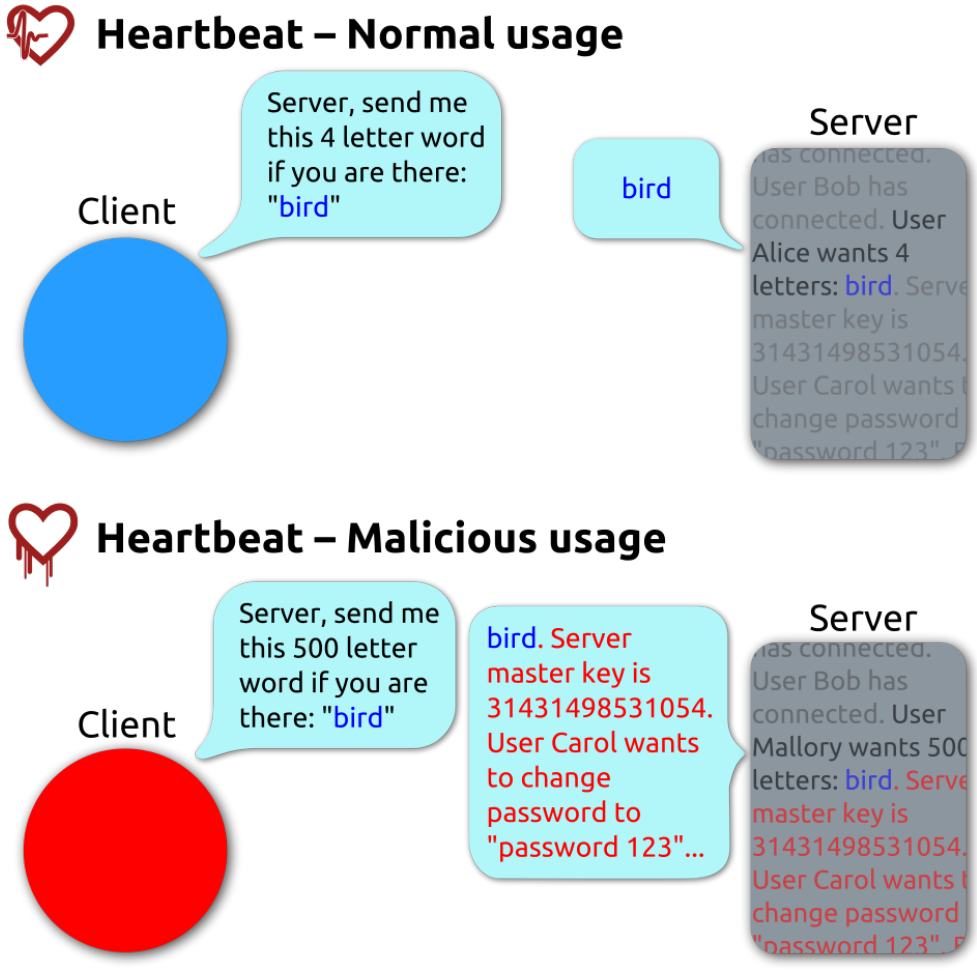
Heartbleed 1/3

- Faille critique d'OpenSSL découverte en avril 2014
- Liée à un débordement de tampon en lecture
- Permet à l'attaquant de récupérer des informations secrètes et notamment fréquemment la clé secrète du serveur, des informations envoyées par un autre client (ex. mots de passe)...
- Lié à l'extension Heartbeat
  - Le client envoie : n (16 bits) et n octets de données
  - Le serveur renvoie les n octets de données envoyées par le client

Heartbleed 2/3

- OpenSSL renvoie *n* octets à partir du début du tampon stockant les données reçues
- Mais OpenSSL ne vérifie pas que le client a bien envoyé *n* octets initialement
- Donc si le client a envoyé moins d'octets, le serveur va envoyer au client les données stockées en mémoire après le tampon de réception (jusqu'à 64 ko)

Heartbleed 3/3



Source : FenixFeather, Wikimedia Commons

Plan

Quel est le problème ?

```c
/* test2.c */
#include <string.h>
#include <stdlib.h>

int i = 1234;

int main (int argc, char **argv) {
  char buf[1024];
  char *pwd = "password";

  strncpy (buf, argv[1], sizeof (buf));
  printf (buf);
  printf ("\n@pwd = %08x\ni = %d, @i = %08x\n",
          pwd, i, &i);
  exit (EXIT_SUCCESS);
}
```

Premier test

```
$ gcc -g -m32 -mtune=i386 -o test2 test2.c
$ ./test2 Hello
Hello
@pwd = 080485a0
i = 1234, @i = 0804a028
```

Exploitation pour lire la pile

```
$ ./test2 %08x
fff135a4
@pwd = 080485a0
i = 1234, @i = 0804a028
$ ./test2 `perl -e 'print "%08x."x20;'`
fff135a4.00000400.00000004.00000174.00000174.00000174
.78383025.3830252e.30252e78.252e7838.2e783830[...]
@pwd = 080485a0
i = 1234, @i = 0804a028
```

Exploitation pour lire la mémoire

```
$ ./test2 AAAA.%08x.%08x.%08x.%08x.%08x.%08x.%08x
AAAA.fff135a4.00000400.00000004.00000174.00000174.00000174.41414141
@pwd = 080485a0
i = 1234, @i = 0804a028

$ ./test2 ‘perl -e ’print "\xa0\x85\x04\x08";’‘\
.%08x.%08x.%08x.%08x.%08x.%08x.%08x
xxxx.fff135a4.00000400.00000004.00000174.00000174.00000174.080485a0
@pwd = 080485a0
i = 1234, @i = 0804a028

$ ./test2 ‘perl -e ’print "\xa0\x85\x04\x08";’‘\
.%08x.%08x.%08x.%08x.%08x.%08x.%s
xxxx.fff135a4.00000400.00000004.00000174.00000174.00000174.password
@pwd = 080485a0
i = 1234, @i = 0804a028
```

Écriture en mémoire

```
$ ./test2 ‘perl -e ’print "\x28\xa0\x04\x08";’‘\
.%08x.%08x.%08x.%08x.%08x.%08x.%n
xxxx.fff135a4.00000400.00000004.00000174.00000174.00000174
@pwd = 080485a0
i = 59, @i = 0804a028
```

Solution

- Utiliser `printf("%s", buf)` au lieu de `printf(buf)`

Plan

Use of Hard-coded Credentials

- CVE-2014-2969 (07/07/2014)
  - NETGEAR GS108PE Prosafe Plus switches with firmware 1.2.0.5 have a hardcoded password of debugpassword for the ntgruser account, which allows remote attackers to upload firmware or read or modify memory contents, and consequently execute arbitrary code

Plan

Plan

XSS

Définition

- Se produit quand une application web ne neutralise pas correctement des entrées contrôlables par un utilisateur avant de les utiliser pour générer une page web visible par d'autres utilisateurs
- Scénario
  - Des données non dignes de confiance sont passées à l'application web
  - L'application génère dynamiquement une page contenant ces données
  - L'application ne vérifie pas si ces données contiennent du code exécutable par le navigateur (JavaScript, HTML, Flash, ActiveX, etc.)
  - La victime visite la page générée et son navigateur exécute le code en question, dans le contexte du domaine de l'application

XSS

Risques

- Vol de cookie, de données privées et d'informations de session (puisque le script semble provenir de l'application web, elle s'exécute avec ses privilèges)
- Exécution de requêtes à la place de l'utilisateur (dangereux si l'utilisateur est administrateur par exemple)

XSS

Types

- Classification proposée par `http://cwe.mitre.org`
- Type 1 : Reflected XSS (non persistant) : Le serveur lit la donnée directement depuis la requête HTTP et l'utilise dans la réponse. L'attaquant doit amener la victime à fournir elle-même le contenu vulnérable à l'application (en lui fournissant l'URL par exemple)
- Type 2 : Stored XSS (persistant) : L'adversaire fait stocker à l'application la donnée malicieuse. Plus tard, l'application relit cette donnée (par exemple depuis une DB) et l'utilise pour générer une page.
- Type 0 : DOM-Based XSS : L'injection se fait via un script exécuté par le client qui lit une entrée utilisateur et modifie la page à l'aide de celle-ci, sans passer par le serveur

XSS

Solutions

- Comme toujours, ne jamais faire confiance à une information qui peut être manipulée par un utilisateur
- Protéger les entrées utilisateur correctement (se servir des fonctions fournies par les langages), plus de détails [17]

Plan

SQL Injection

Principe

- Se produit quand une application génère une requête SQL comportant des parties influencées par l'utilisateur sans neutraliser les éléments qui pourraient avoir une influence sur le sens de la requête

SQL Injection

Exemple 1

- Lors d'une authentification, le serveur utilise la requête suivante `SELECT * FROM users WHERE login='<login>' AND password='<pass>'`
- L'utilisateur saisit comme mot de passe `x' OR 'a'='a`
- La requête se transforme donc en `SELECT * FROM users WHERE login='admin' AND password='x' OR 'a'='a'`, ce qui se simplifie en `SELECT * FROM users WHERE login='admin'`

SQL Injection

Exemple 2

- Le serveur veut faire la requête suivante `SELECT * FROM products WHERE id='<id>'`
- L'utilisateur entre l'identifiant produit suivant `x'; DELETE FROM products; --`
- La requête envoyée au serveur se transforme donc en :

```
SELECT * FROM products WHERE id='x';
DELETE FROM products;
--'
```

ce qui peut être embêtant...

SQL Injection

Solutions

- Protéger les caractères spéciaux dans les entrées modifiables par l'utilisateur (en utilisant notamment les fonctions dédiées des API de communication avec les DB)
- Plus de détails [16]

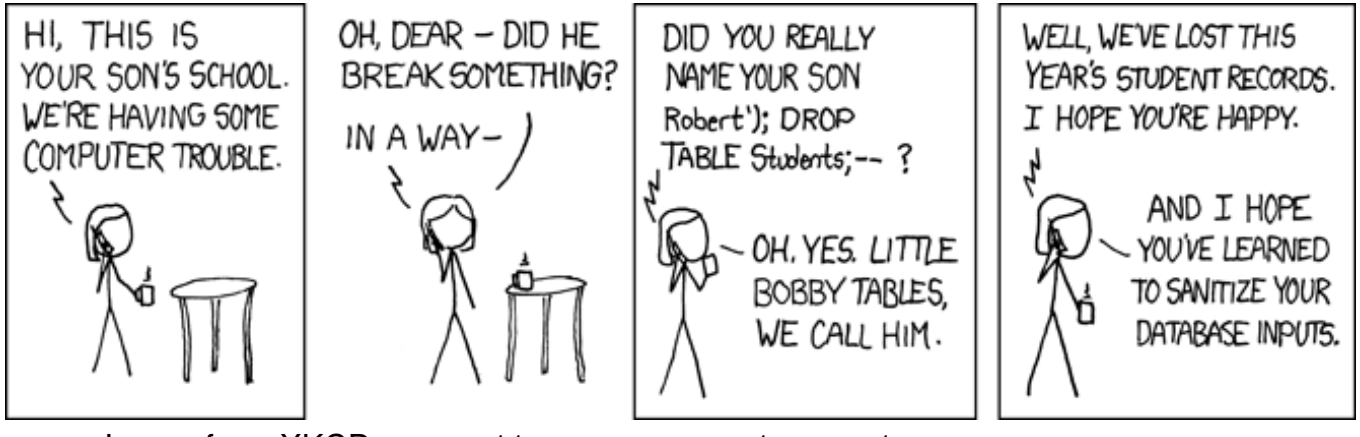


Image from XKCD : `http://imgs.xkcd.com/comics/exploits_of_a_mom.png`

Plan

Cross-Site Request Forgery

Principes

- Se produit quand une application web est conçue pour recevoir une requête depuis un client sans mécanisme pour vérifier qu'il l'a envoyée intentionnellement
- Il est alors possible pour un attaquant d'amener l'utilisateur à faire une requête vers cette application sans le vouloir (par exemple en visitant un site malicieux)

Plan

Bonnes pratiques 1

- Établir et garder le contrôle sur toutes les entrées d'une application
- Entrées : paramètres, arguments, cookies, variables d'environnement, n'importe quoi lu depuis le réseau, entête de requête, composants de l'URL, email, fichiers, DB, résolution DNS inversée, etc.
- Valider chaque entrée
  - Longueur
  - Type
  - Syntaxe
  - Entrées manquantes ou en supplément
  - Règles métier

Bonnes pratiques 2

- Établir et garder le contrôle sur toutes les sorties d'une application
- Sorties : Requêtes SQL, pages Web, email, etc.
- Vérifier qu'une donnée ne modifie par le sens de la sortie (ex. Injection SQL)

Bonnes pratiques 3

- Verrouiller l'environnement
- Exécuter l'application en tant qu'utilisateur non privilégié
- Désactiver les messages d'erreur verbeux
- Utiliser des mécanismes comme la virtualisation, les bacs à sable, etc.

Bonnes pratiques 4

- Assumer que les composants externes peuvent être corrompus et que votre code peut être lu par n'importe qui

Bonnes pratiques 5

- Utiliser des mécanismes de sécurité reconnus par l'industrie au lieu d'inventer les siens

Cinquième partie V

Conclusion

Conclusion

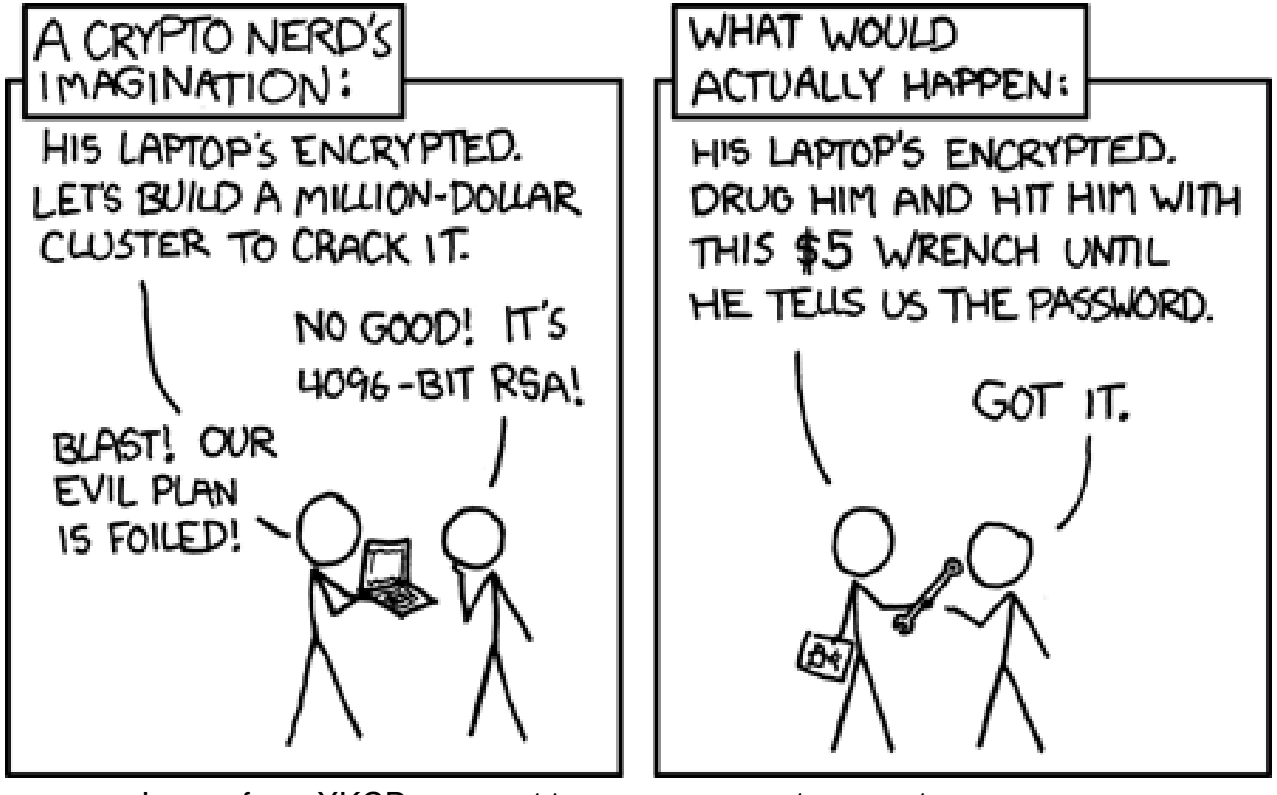


Image from XKCD : `http://imgs.xkcd.com/comics/security.png`

Conclusion personnelle

- Travailler dans la sécurité nécessite
  - Une connaissance technique complète et parfaite du fonctionnement de ce que l'on cherche à sécuriser
  - Une vision globale de tout le système dans lequel s'inscrit le produit
  - Une mise à jour constante de ses connaissances

Sixième partie VI

Annexe

Plan

References

References I

[1] Poisontap.
`https://samy.pl/poisontap/`, December 2016.

[2] Usb killer.
`https://www.usbkill.com/`, December 2016.

[3] Zhaohui Wang Angelos Stavrou.
Exploiting smart-phone usb connectivity for fun and profit.
In *Black Hat DC*, January 2011.
`https://media.blackhat.com/bh-dc-11/Stavrou-Wang/BlackHat_DC_2011_Stavrou_Zhaohui_USB_exploits-wp.pdf`.

[4] K. Chen.
Reversing and exploiting an apple firmware update.
In *Black Hat*, July 2009.
`http://www.blackhat.com/presentations/bh-usa-09/CHEN/BHUSA09-Chen-RevAppleFirm-PAPER.pdf`.

[5] Loïc Duflot.
Bogues et piégeages des processeurs, quelle conséquence sur la sécurité ?
In *SSTIC*, June 2008.
`http://www.ssi.gouv.fr/archive/fr/sciences/fichiers/lti/sstic08-duflot.pdf`.

[6] Loïc Duflot and Yves-Alexis Perez.
Can you still trust your network card ?
In *CanSecWest*, 2010.
`http://www.ssi.gouv.fr/IMG/pdf/csw-trustnetworkcard.pdf`.

[7] Christophe Giraud.
Attaques de cryptosystèmes embarqués et contre-mesures associées.
PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, 26 octobre 2007.

References II

[8] Jeremi Gosney.
Password cracking HPC.
In *Passwords12 Security Conference*, dec 2012.
http://passwords12.at.ifi.uio.no/Jeremi_Gosney_Password_Cracking_HPC_Passwords12.pdf.

[9] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard.
KASLR is Dead : Long Live KASLR, pages 161–176.
Springer International Publishing, Cham, 2017.

[10] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten.
Lest we remember : Cold boot attacks on encryption keys.
In *Proceedings of the 17th USENIX Security Symposium*, July 2008.

[11] A. Huang.
Keeping secrets in hardware : the Microsoft XBox (TM) case study.
Technical Report AI Memo 2002-008, Massachusetts Institute of Technology, May 2002.

[12] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom.
Spectre attacks : Exploiting speculative execution.
ArXiv e-prints, January 2018.
https://meltdownattack.com/spectre.pdf.

[13] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg.
Meltdown.
ArXiv e-prints, January 2018.
https://meltdownattack.com/meltdown.pdf.

References III

[14] Charlie Miller.
Battery firmware hacking.
In *Black Hat*, August 2011.
https://media.blackhat.com/bh-us-11/Miller/BH_US_11_Miller_Battery_Firmware_Public_WP.pdf.

[15] Karsten Nohl, David Evans Starbug, and Henryk Plötz.
Reverse-Engineering a Cryptographic RFID Tag.
In *USENIX Security Symposium*, pages 185–193, July 31 2008.
San Jose, CA, USA (http://www.usenix.org/event/sec08/tech/full_papers/nohl/nohl_html/Online HTML).

[16] Dave Wichers.
Sql injection prevention cheat sheet, January 2011.
http://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet.

[17] Jeff Williams and Jim Manico.
Xss (cross site scripting) prevention cheat sheet, January 2011.
http://www.owasp.org/index.php/XSS_%28Cross_Site_Scripting%29_Prevention_Cheat_Sheet.

[18] Yuval Yarom and Katrina Falkner.
Flush+reload : A high resolution, low noise, l3 cache side-channel attack.
In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, 2014. USENIX Association.
https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.