Compte-rendu : Laboratoire d'exploitation de la faille Spectre

Auteurs :

Said Faissoil
Belhachemi M'hamed
Daunar Benjamin

Tuteur de stage et superviseur : Godard Emmanuel
Etablissement / formation : Aix Marseille Luminy - M1 Informatique FSI 2020/2021

Sommaire

Introduction

Le 4 janvier 2018, deux vulnérabilités affectant plusieurs familles de processeurs et pouvant conduire à des fuites d'informations ont été rendues publiques. Intitulées Spectre et Meltdown, ces deux vulnérabilités ont reçu les identifiants CVE-2017-5715, CVE-2017-5753 pour Spectre et CVE-2017-5754 pour Meltdown. Une nouvelle attaque du même type a été annoncée en 2019, il s'agit de ZombieLoad, CVE-2018-12130 et CVE-2019-11135; ainsi qu'une preuve d'insuffisance des mitigations en 2020 : CVE-2020-0549.

Les processeurs modernes intègrent plusieurs fonctionnalités visant à améliorer leurs performances. Parmi celles-ci, l'exécution dite out-of-order permet d'exécuter les instructions d'un programme en fonction de la disponibilité des ressources de calculs et plus nécessairement de façon séquentielle. Une faiblesse de ce mécanisme peut cependant conduire à l'exécution d'une instruction sans que le niveau de privilèges requis par celle-ci ne soit correctement vérifié. Bien que le résultat de l'exécution d'une telle instruction ne soit pas validé par la suite, il peut être possible de récupérer l'information en utilisant une attaque par canaux cachés. Un certain nombre de mises à jours ont été effectuées afin d'essayer de remédier à la présence de ces failles, malgré les pertes de performance que cela pouvait engendrer.

Plus récemment, en Mars 2021, un rapport de l'équipe de recherche en cyber-sécurité de Google a pu mettre en évidence une nouvelle manière d'exploiter la faille Spectre, Aussi nous nous intéresserons à cette nouvelle preuve de concept. Pour cela, il nous faudra dans un premier temps comprendre ce qu'il y a caché derrière les processeurs de dernière génération, afin de comprendre les mécanismes sur lesquels reposent la faille, pour pouvoir par la suite exécuter en milieu normalisé des attaques exploitant la faille Spectre.

I. Hypothese

a. CPU

De quelle manière est constitué un processeur ?

Pour comprendre comment fonctionne Spectre, il est essentiel de revenir sur les différentes compositions d'un processeur.

1. Composition CPU

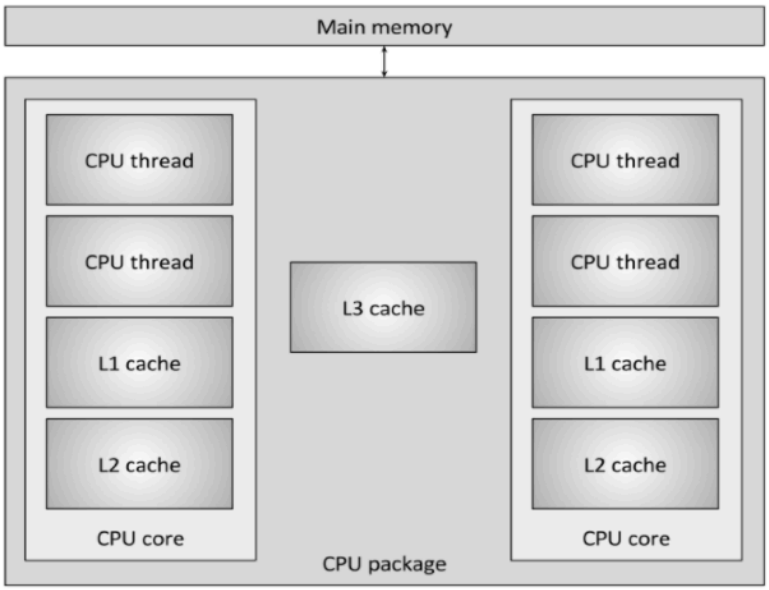


Figure 1 : Composition d'un processeur[1]

- Un **Coeur** (Core) est un ensemble de circuits capables d'exécuter des programmes, un processeur single-core (contenant 1 seul coeur) ne peut exécuter qu'une seule instruction à la fois; Les processeurs modernes utilisés dans des ordinateurs contiennent pour la plupart 2 ou 4 coeurs, les processeurs utilisés pour les serveurs peuvent contenir jusqu'à 28 coeurs.
- Un Coeur peut contenir jusqu'a 2 **threads**, ces derniers sont des composants virtuels contenant différents registres et ont la capacité d'exécuter un flux de code machine.
- Chaque Coeur contient un ensemble de niveaux de **cache** (3 principalement), le cache L1 (niveau 1) est la mémoire la plus rapide et dans un système informatique mais elle a une petite capacité de stockage, L2 quant à lui est plus lent que L1 mais dispose d'un plus grand espace de stockage.
- Le cache L3 est partagé entre tous les coeurs d'un processeur.

2. Mémoire Cache

Les processeurs exécutent un nombre « astronomique » de calculs, et dans la plupart du temps lorsque des informations doivent êtres utilisées pour des opérations futurs, celles-ci sont stockés dans le cache (en premier lieu L1, puis L2…jusqu'à la RAM).

Les caches contiennent précisément des blocs de mémoire qui sont des copies de la même information stocké dans la RAM, ces derniers utilisent un adressage physique ou bien virtuel; L1 utilise dans la plus part du temps une indexation virtuelle.

Pour bien comprendre le fonctionnement de Spectre que nous détaillerons par la suite, il nous faut connaitre les 2 notions suivantes :

Cache hit : lorsque le processeur souhaite récupérer une donnée et que celle ci est trouvée.

Cache miss : lorsque le processeur cherche une donnée dans la mémoire (cache L1 puis L2….RAM) et que celle-ci n'est pas trouvée.

3. Branchement et prediction de branchement

Pour pouvoir améliorer la vitesse de calcul, il s'est avéré judicieux de découvrir des algorithmes de **prédiction** afin de permettre un gain de temps certain lors de calculs sur machines.

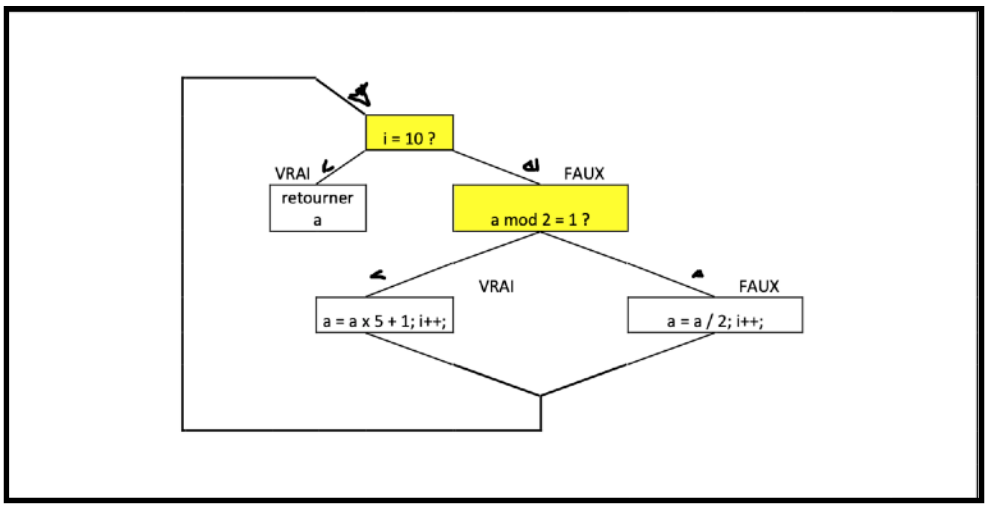
3.1. Branchement

Un branchement est un nœud dans **le graphe d'exécution** d'un algorithme. Il se traduit dans le code par des **opérations de comparaison** contenue aussi bien dans des instructions conditionnelles, que dans des boucles.

Prenons un exemple : l'algorithme suivant :

```
entier a = 1;
Pour i allant de 0 à 9
        Si a est impair
                a = a x 5 +
                1;
        Sinon
                a = a / 2;
Fin pour
retourner a
```

Cet algorithme possède l'arbre d'exécution suivant :



En particulier, il contient 2 branchements :

- (1) : i = 10 ?
- (2) : a est-il impair ?

On peut constater que (1) sera visité 11 fois, et (2) 10 fois. Aussi nous verrons combien d'erreurs de prédiction peuvent faire de simples algorithmes de prédiction de branchement.

3.2. Prédiction de branchement

A partie du graphe précédent (cf. 1.2.1), nous pouvons appliquer deux algorithmes de prédiction dits à 1 bit et à 2 bits, afin de constater que la prédiction est plutôt fiable.

- **Algorithme de prédiction à 1 bit**

Ici, il s'agit d'utiliser un bit comme bit mémoire pour se rappeler du comportement de l'algorithme lors de la dernière exécution. Aussi, ce bit mémoire prend la valeur 1 lorsqu'on prédit un résultat vrai, 0 lorsqu'on prédit un résultat faux. Si on se trompe, on inverse la valeur du bit mémoire.

Supposons que la valeur initiale du bit mémoire soit 0. On a alors le tableau de prédiction suivant pour le branchement (1) :

	valeurs de i	prédiction Bit mémoire	Résultat réel	
1ère exécution	0	0	0	
	1	0	0	
	2	0	0	
	3	0	0	
	4	0	0	
	5	0	0	
	6	0	0	
	7	0	0	
	8	0	0	
	9	0	0	
	10	0	1	
2nde exécution	0	1	0	
	1	0	0	
	2	0	0	
	3	0	0	
	4	0	0	
	5	0	0	
	6	0	0	
	7	0	0	
	8	0	0	
	9	0	0	
	10	0	1	

On constate qu'en 2 exécutions, le branchement (1) est visité 22 fois, or seules 3 erreurs de prédiction ont été commises.

- **Algorithme de prédiction à 2 bits**

Cette fois-ci, 2 bits mémoires XY sont utilisées.

Suivant les valeurs de X et Y, nous avons les prédictions suivantes :

- XY = 00 – Prédiction forte résultat « faux »
- XY = 01 – Prédiction faible résultat « faux »
- XY = 10 – Prédiction faible résultat « vrai »
- XY = 11 – Prédiction forte résultat « vrai »

Les valeurs des bits mémoires évoluent au cours de l'exécution suivant les règles suivantes :

- En cas d'erreur de prédiction, si X et Y ont la même valeur, inverser Y, sinon inverser les deux
- Lorsque la prédiction est bonne, si X et Y ont la même valeur, ne rien faire, sinon, inverser Y

On a alors le tableau de prédiction suivant pour le branchement (1) :

	valeurs de i	prédiction Bit mémoire	résultat	
1ère exécution	0	00	0	
	1	00	0	
	2	00	0	
	3	00	0	
	4	00	0	
	5	00	0	
	6	00	0	
	7	00	0	
	8	00	0	
	9	00	0	
	10	00	1	
2nde exécution	0	01	0	
	1	00	0	
	2	00	0	
	3	00	0	
	4	00	0	
	5	00	0	
	6	00	0	
	7	00	0	
	8	00	0	
	9	00	0	
	10	00	1	

On constate qu'il n'y a alors plus que 2 erreurs de prédiction en 2 exécutions.

Maintenant que ces bases sont posées, voyant comment nous pouvons utiliser des algorithmes de prédiction de branchement au sein d'un processeur.

4. Execution speculative

Dans le monde des microprocesseurs, où la vitesse d'exécution exprimée en gigahertz est primordiale, les différents constructeurs de microprocesseurs se battent pour savoir qui aura le plus gros chiffre. Pour obtenir ce chiffre ils passent par différentes méthodes comme la réduction de la taille des transistors, ou même l'augmentation du nombre de cœurs physique du microprocesseur.

Le problème de ces techniques c'est que les constructeurs sont limités par les lois de la physique car en dessous de 7 nanomètres les transistors chauffent très vite, ainsi que par des limites de tailles. Il n'y a aucune standardisation au niveau de la taille des microprocesseurs mais leurs tailles diffèrent peu même avec un nombre de cœurs très différent.

Puisque la puissance des microprocesseurs ne peut être augmentée à l'infinie, les constructeurs mettent en place des algorithmes pour tenter d'accélérer l'exécution d'un microprocesseur. Un de ces algorithmes est l'exécution spéculative, cet algorithme a pour but d'exécuter les instructions avant même de savoir s'il est possible d'exécuter ladite

instruction, le meilleur exemple est celui d'un branchement conditionnel. Cet algorithme permet au microprocesseur d'être plus rapide à condition de bien deviner quel branchement utiliser, pour cela l'exécution spéculative utilise la prédiction de branchement.

b. Spectre variant 1 : bounds check bypass (CVE-2017-5753)

Cette variante de Spectre que nous détaillerons par la suite, repose sur le système de prédiction de branchement ainsi que la mise en cache des données par le processeur.

Nous expliquerons dans un premier temps comment il est possible de "tromper" le processeur à l'aide de l'exécution spéculative et d'enregistrer des informations privées dans le cache du processeur, ensuite nous expliquerons comment les exploités et de les décodés.

- **Exploitation d'erreur de prédiction de branchement**

Nous avons ici un programme basic défini par une condition "if" ainsi qu'une affectation de variable à l'intérieur de celle-ci.

```
if (untrusted_index < array1_size) {
   y = array2[array1[untrusted_index] * 256] ;
}
```

Cette condition "if" effectue une vérification des limites du tableau **array1** afin de s'assurer que l'index **untrusted_index** soit inférieur à la taille du tableau **array1_size**; cette vérification est nécessaire pour s'assurer que le programme ne lit aucune information au-delà des limites du tableau.

Il faut comprendre que cela fonctionne dans les registres lors de l'exécution du programme, car la condition "if" empêche le processeur de lire la mémoire sensible en dehors d'**array1**. Et ainsi, une entrée **untrusted_index** hors limites (supérieur à **array1_size**) pourrait déclencher une exception.

Mais cela ne prends pas en compte les comportements micro-architecturaux du processeur impliquant une mauvaise prédiction de branchement lorsque **untrusted_index** est supérieur à **array1_size**, ce qui entraine alors une exécution spéculative.

On pourra ensuite utiliser la valeur chargée à partir d'**array1** comme index dans un autre tableau **array2**. Cette instruction crée des "effets secondaires" et déclenchera une mise en cache de valeur qui a été lu hors limite.

Le processeur reconnaitra une mauvaise prédiction de branchement et rejètera l'état exécuté de manière spéculative en réinitialisant l'état de son registre, la valeur secrète qui a été lu, restera elle dans le cache CPU.

- **Décoder l'information à l'aide d'une attaque Temporelle (Timing attack)**

Etant donné que cette partie est la suite directe de la partie précédente, on supposera pour simplifier la compréhension que nous avons lu et enregistrer dans le cache CPU un mot secret dont on cherche à connaitre la valeur, ce mot est « bonjour »

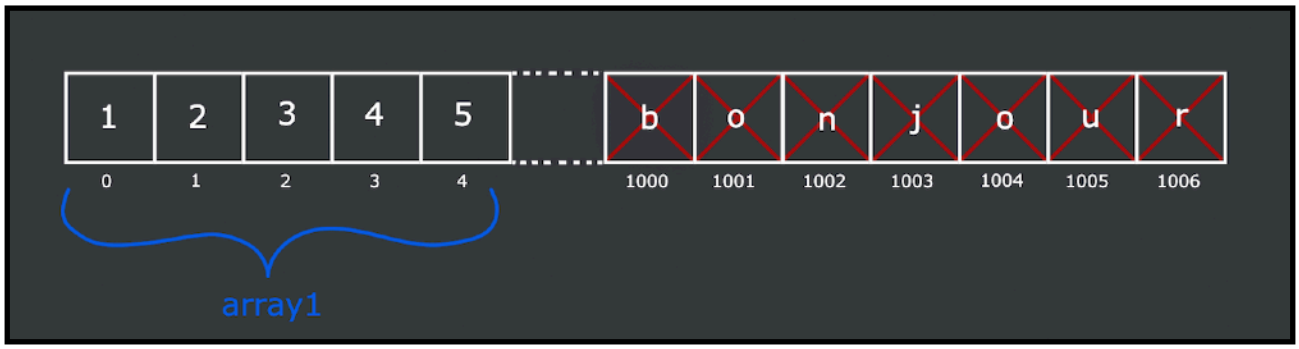


Figure 2 : exemple de représentation du cache CPU avec les informations enregistrées

Pour trouver ce mot nous procéderons de la manière suivante :

L'attaquant crée un tableau de taille 255 contenant tous les caractères du tableau ASCII, on appellera notre tableau **char** dans cet exemple.

Le programme va d'abord lire un caractère du tableau, supposons que ça soit char[97] qui vaut 'a', celui-ci sera donc mis dans le cache CPU.

Ensuite, on tente d'accéder à l'adresse de la valeur qu'on veut découvrir, ici 'b' à l'adresse 1000 en reprenant la figure 2, et on calcule le temps que mets le processeur à accéder à la valeur pointée.

On enregistre le temps écoulé, si un certains temps fixé par le programme est dépassé, cela veut donc dire que ce n'est pas la bonne valeur (cache miss). Sinon c'est un cache hit

On réitère ensuite l'opération avec tous les caractères du tableau **char**, le caractère ayant le temps le plus court à le plus de chance de correspondre à la valeur recherchée.

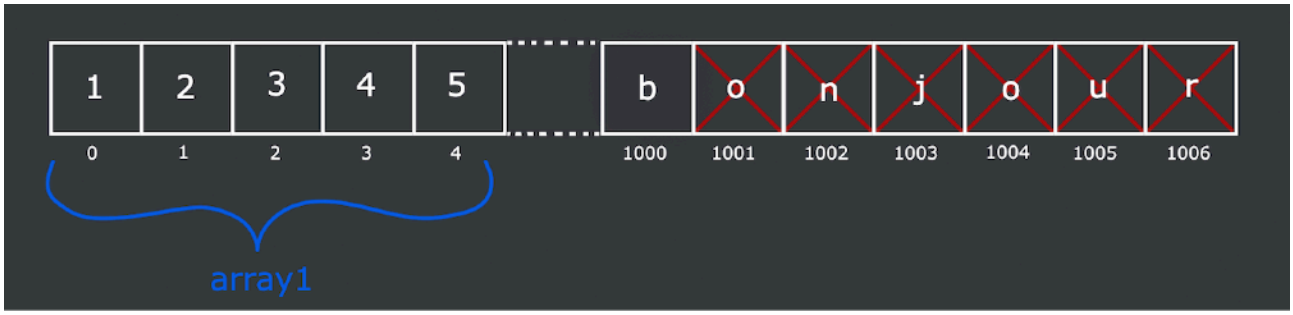


Figure 3 : exemple de représentation du cache CPU avec la valeur de l'adresse 1000 déchiffrée

Dans notre scénario, nous avons obtenu pour 'a' un temps de 2,5 ms, et pour 'b' un temps de 0,9 ms, 'b' a été enregistré comme caractère ayant le meilleur temps.

Donc la valeur de l'adresse 1000 trouvée est 'b'.

Pour trouver le reste il suffit juste de répéter l'opération, mais cette fois-ci pour l'adresse 1001 et ainsi de suite.

À noter bien sur que la création de tableau de caractères n'est pas la seule manière possible pour performer une attaque temporelle.

c. Spectre variant 1 : bounds check bypass (browser attack)

De manière assez naturelle, on remarque que le mécanisme de l'attaque est le même que celui utilisé par le variant 1, à une différence majeure près : l'attaquant accède à la machine de la victime depuis une autre machine !

- **La preuve de concept Google**

Le 12 Mars 2021, la Google Security Research Team publiait sur son blog une preuve de concept alarmante. Il s'agit de la première exploitation de la faille Spectre à distance. Selon eux, l'utilisation d'un navigateur pouvant exécuter du code JavaScript sur un ordinateur ayant un processeur utilisant l'exécution spéculative peut permettre à un attaquant d'exécuter du code malveillant en vue d'influencer le Branch Target Buffer afin d'introduire dans le cache des données que l'on souhaite lire. Ils ont attaché une vidéo démonstrative de l'attaque dans laquelle la machine victime utilise Chromium 88 comme navigateur Web, sous Linux, avec un processeur Intel i7 – 6500U.

L'intérêt d'utiliser Chromium 88 réside dans le fait que ce navigateur web permet l'exécution de code JavaScript. Le processeur lui, a pour propriété d'utiliser l'exécution spéculative.

Dans leur vidéo démonstrative, on peut observer cependant que les données qu'ils récupèrent sur l'ordinateur cible en cas de succès sont des données qu'ils connaissent pour les avoir « fixées », aussi ils savent déjà ce qu'ils cherchent.
Une analyse du temps d'exécution du processeur permet de vérifier que l'information cherchée est dans le cache. Il ne leur reste plus qu'à en déduire les informations contenues dans ce cache.

La particularité de cette démonstration repose sur le fait que la victime se connecte à la machine malveillante, via son navigateur web, et lance l'exécution de l'exploitation de son propre chef. Par ailleurs, les données récupérées ici sont connues à l'avance.

d. Spectre variant 2 : branch target injection (CVE-2017-5715)

[1] L'idée de base de l'attaque est de cibler le code de la victime qui contient un branchement indirect dont l'adresse est chargée de mémoire et de vider la partie du cache contenant l'adresse. Puis, lorsque le processeur atteint le branchement indirect, il ne connaît pas la vraie destination du saut, et il ne sera donc pas en mesure de calculer la vraie destination tant qu'il n'aura pas fini de recharger dans les données précédemment supprimées, cette opération prend quelques centaines de cycles. Par conséquent, il existe une fenêtre de temps d'à peu près 100 cycles dans lesquels le processeur exécutera spéculativement des instructions basées sur la prédiction des branches.

```
void bhb_update(uint58_t *bhb_state, unsigned long src, unsigned long dst) {
 *bhb_state <<= 2;
 *bhb_state ^= (dst & 0x3f);
 *bhb_state ^= (src & 0xc0) >> 6;
 *bhb_state ^= (src & 0xc00) >> (10 - 2);
 *bhb_state ^= (src & 0xc000) >> (14 - 4);
 *bhb_state ^= (src & 0x30) << (6 - 4);
 *bhb_state ^= (src & 0x300) << (8 - 8);
 *bhb_state ^= (src & 0x3000) >> (12 - 10);
 *bhb_state ^= (src & 0x30000) >> (16 - 12);
 *bhb_state ^= (src & 0xc0000) >> (18 - 14);
}
```

Figure 4 : La fonction de mise à jour du "branch history buffer"[2]

La raison pour laquelle la prédiction de branchement peut être abusée vient de sa fonction de mise à jour « bhb_update » fig4. Les adresses des branchements sont stockées dans une variable de taille 58 octets et chaque adresse prend 2 octets. Il peut donc y avoir qu'un maximum de 29 branchements, en polluant le « branch history buffer » il est possible de pouvoir tromper le processeur.

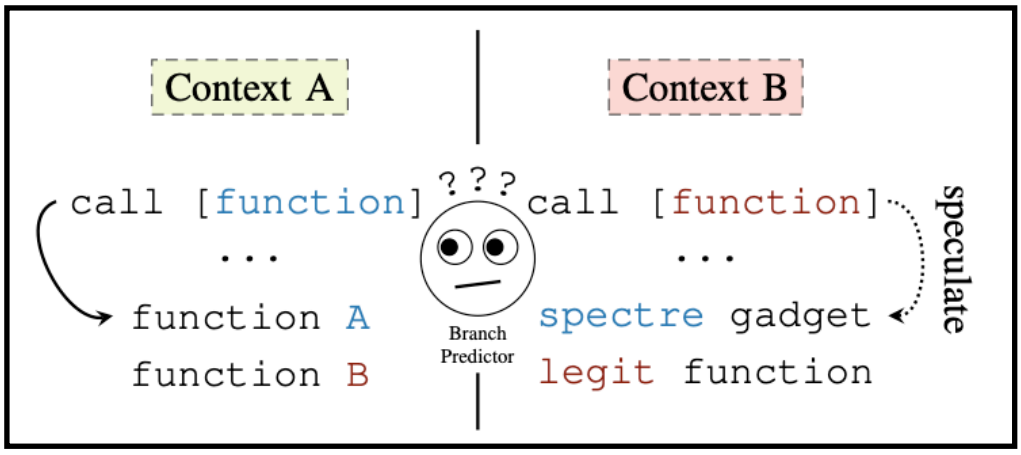


Figure 5 : Le prédicateur de branche pollué dans le contexte A par l'attaquant se retrouve à devoir spéculer dans le contexte B en se basant sur les résultat su contexte A.[2]

Tout comme le variant 1 la preuve de concept (qui sera vu après) du variant 2 utilisera la prédiction de branchement, l'attaque chronométrée, l'exécution spéculative et l' « out-of-bounds ». Cependant il n'est pas question ici de tendre un piège au processeur en l'attendant au pied d'une condition mais plutôt de forcer le processeur à exécuter de manière spéculative une mauvaise fonction. fig5.

II. Execution

a. Configuration de la VM

- **Configuration des VM pour les variants 1 & 2 de Spectre**

Les machines virtuelles utilisées pour spectre v1 et v2 sont des machines virtuelles sous debian 11 dans lesquelles les mitigations contre spectre ont été désactivées et les sources immédiatement disponibles.
Un fichier « README.md » est mis à disposition expliquant la démarche a suivre pour lancer les machines virtuelles.

- **Configuration de la VM pour le variant 1 : attaque par navigation (browser attack)**

Nous avons commencé par créer une **bionic-salt** box en suivant la méthode décrite par l'exemple présent dans le répertoire **Labsec/exemple** du dépôt distant que nous utilisions. L'idée étant par la suite d'utiliser une configuration réseau afin de tester autrement qu'en local.

Nous avons par ailleurs configuré un dossier de partage comme étant **src/** (répertoire contenant le code de l'attaque) sur la machine hôte,

La machine cible

Nous avons donc désactivé les mitigations et installé chromium 88 sur la machine virtuelle. Nous avons préparé la configuration automatique des paramètres réseaux, mais n'avons pas encore placé les fichiers **config.sls** et **top.sls** de sorte à exécuter l'attaque « localement » dans un premier temps.

La machine de l'attaquant

Nous avons donc configuré la machine virtuelle afin qu'elle puisse héberger le site **leaky.page** dont le code source est disponible sur un dépôt **Git** dont le lien est disponible sur la même page. Pour cela, nous avons, à l'aide du script présent dans le répertoire **rules/leaky.page.config** de notre travail, installé apache et créé le répertoire **/var/www/html/** qui nous permettra de mettre notre site « en ligne ».
Nous avons par ailleurs désactivé les mitigations sur la machines de l'attaquant afin de s'assurer un minimum de barrières à l'exécution.

b. Programme Spectre : bounds check bypass

Nous allons dans cette partie tenter d'expliquer comment fonctionne la variante 1 de Spectre en détaillant les fonctions du programme **spectre.c**[4], le code qu'on présenteras dans cette partie seras factoriser, le but ici est de récupérer la valeur secrète qui est : « The Magic Words are Squeamish Ossifrage »

À noter que la variante 1 de spectre n'existe pas seulement en langage c mais aussi en javascript, elle sera en parie en partie II.c.

- **Le code victime**

```
21 char * secret = "The Magic Words are Squeamish Ossifrage.";
22 void victim_function(size_t x) {
23 if (x < array1_size) {
24    temp &= array2[array1[x] * 512];
25  }
26 }
```

On commence tout d'abord par enregistrer la variable secret en mémoire.

Nous allons supposer un scénario comme précédemment dans la partie I.b., dans celui-ci la variable **secret** est stockée dans l'adresse 0x0020; Etant donné que le texte secret est en

ASCII, chaque caractère correspond à un octet en mémoire. Notre secret occupe donc la mémoire de l'adresse 0x0020 à 0x0059 avec :
0x0020 —> 'T'
0x0021 —> 'h'
0x0022 —> 'e'
……
0x0059 —> '.'

Nous supposerons également que notre tableau **array1** se trouve à l'adresse 0x0001
La fonction **victim_function()** fonctionne de manière assez simple, celle-ci sera appelée à plus reprises dans la fonction **readMemoryByte()** que nous présenterons par la suite, avec un argument **x** plus petit qu'**array1_size** et donc respectant la condition "if".

Le processeur entrainé à une condition toujours vraie, il sera donc possible d'injecter une variable **x** supérieur a **array1_size** (out of bound).

Dans cet exemple et dans le reste de l'explication nous chercherons la première lettre du secret c'est à dire 'T'; On connait son adresse (0x0020), rappelons également que l'adresse d'**array1** est 0x0001.

Afin de connaitre la valeur **x** à fournir pour accéder à cette adresse mémoire le programme réalisera le calcule suivant : 0x0020 - 0x0001 = 0x0019, soit 25 en décimal.

Lorsque notre code atteindra à l'aide de l'exécution spéculative l'instruction **array1[25]**, la valeur récupérée sera l'octet représentant 'T'.

La représentation décimale de 'T' étant 84; le processeur résout donc l'instruction suivante dans le script : **array2[X*512]** avec **X**= 84 et enregistre le résultat dans son cache (L1).

À l'instant ou le processeur fera état de cette erreur, il réinitialisera son registre mais la mise en cache de l'information nous permettra de récupérer à nouveau la valeur avec l'instruction **array2[X*512]**.

- **Code d'analyse**

```
33 #define CACHE_HIT_THRESHOLD 80
34
35 void readMemoryByte(size_t malicious_x, uint8_t value[2], int score[2]){
36
37 for (tries = 999; tries > 0; tries--){
38  for (i = 0; i < 256; i++)
39    _mm_clflush( & array2[i * 512]);
40  training_x = tries % array1_size;
```

```
41  for (j = 29; j >= 0; j--) {
42    _mm_clflush( & array1_size);
43    x = ((j % 6) - 1) & ~0xFFFF;
44    x = (x | (x >> 16));
45    x = training_x ^ (x & (malicious_x ^ training_x));
46    victim_function(x);
47  }
```

Dans cette, fonction qui sera appelée dans la fonction **main()**, on peut remarquer à la ligne 37, une boucle "for" avec 999 itérations; Celle-ci nous permettra comme nous l'avons expliquer précédemment, "d'habituer" notre processeur en affectant une variable **x** respectant la condition "if" du code victime à plusieurs reprises . La variable **malicious_x** permet d'injecter à ce dernier une variable non fiable permettant la lecture d'informations secrètes.

Les instructions des lignes 39 et 42 permettent quant à elles de supprimer/vider **array1_size** et **array2** du cache à chaque itération et donc chaque appel de la fonction **victim_function()**. Donc le processeur sera obliger de chercher leurs valeurs dans la RAM, ce qui pousse donc le processeur à faire une prédiction de branche considérant la condition étant vraie.

La deuxième partie du code est celle nous permets de trouver la valeur recherchée à l'aide d'une attaque temporelle (Timing attack).

```
51  for (i = 0; i < 256; i++) {
52    mix_i = ((i * 167) + 13) & 255;
53    addr = & array2[mix_i * 512];
54    time1 = __rdtscp( & junk);
55    junk = * addr;
56    time2 = __rdtscp( & junk) - time1;
57    if (time2 <= CACHE_HIT_THRESHOLD && mix_i != array1[tries %array1_size])
58      results[mix_i]++;
60  }
```

Comme on peut le voir à partir de la ligne 51, on affecte une valeur dépendante allant entre 0 et 255 à la variable **mix_i**, cette valeur dépends de la variable **i** conditionnée elle même dans une boucle for allant de 0 à 255.

Il nous suffit de réaliser à chaque itération de boucle l'affectation **array2[mix_i*512]** et de calculer le temps (à l'aide de la fonctions **__rdtscp**) que le processeur mets à nous fournir l'information.

Ensuite le programme effectue quelques comparaisons et vérifications, si le temps écoulé pour une valeur testée dépasse **CACHE_HIT_THRESHOLD**, cela correspond à un cache miss et donc la valeur testée ne correspond pas à celle recherchée. Dans le cas contraire on incrémente l'index correspondant à la valeur testée dans le tableau **result**.

Le programme enregistre ensuite les deux valeurs contenant le temps le plus court; Étant donné que dans notre exemple, la valeur recherchée est 'T', dans ce cas la ce sera le résultat avec le meilleur temps qui sera **result[84]**.

- **Main**

Dans la partie **main**, on se contente d'abords d'affecter l'adresse du secret à la variable **malicious_x**, le programme lance ensuite une boucle "while" qui lancera à son tour la fonction **readMemeoryByte()** et le meilleur résultat obtenu, sera affiché.
La variable **malicious_x** sera incrémenté à chaque itération afin de passer à l'adresse suivante et chercher sa valeur ainsi de suite.

c. Programme Spectre : bounds check bypass (browser attack)

Une premier lancement des VM nous permet de mettre en évidence que la création de box a fonctionné. Toutefois une fois en connecté en **ssh** aux machines virtuelles, un problème se pose, apache ne s'installe pas sur la machine de l'attaquant, aussi on se retrouve à effectuer toute la configuration manuellement.

Une fois les configurations effectuées, on modifie le Vagrantfile de la machine cible afin de pouvoir l'exécuter avec une interface utilisateur, de sorte à pouvoir se connecter à la **leaky.page** de l'attaquant. Toutefois, nous nous retrouvons face à une demande d'authentification dont les paramètres ne sont pas ceux introduits dans le Vagrantfile.

Aussi les résultats obtenus sont ceux obtenus à partir d'un **Windows 10 – CPU Intel i7 10700F**. Dans un premier temps le test de la leaky.page ne parvient pas à accéder au cache. Cependant, nous persistons à répéter l'exécution. Au bout de la 4ème tentative, il y a bien une fuite de mémoire. Forcé de constater qu'il s'agit d'une attaque dure à mettre en place vu l'incertitude de la réussite

d. Programme Spectre : branch target injection

Ce POC [3] est très proche du variant 1, comme expliqué un peu plus tôt, les deux utilisent quasiment les mêmes procédés, on ne se concentrera ici que sur les parties non vues

précédemment, à noter que ce POC est simplifié (même zone mémoire / attaquant et victime ont les mêmes droits...) et que le langage de programmation choisi est le C.
Pour commencer, nous avons besoin d'un pointeur que nous appelons **target** et qui va pointer sur deux fonctions soit

Fonction gadget

```
int gadget(char *addr){
    return channel[*addr * GAP]; // GAP = 1024
}
```

Ou soit

Fonction safe_target

```
int safe_target(){
    return 42; //valeur a
}
```

Dans un premier temps l'attaquant doit polluer le « Branch History Buffer » de la victime pour cela target doit pointer vers la fonction gadget, ensuite l'attaquant appelle la fonction victime 50 fois et c'est cette même fonction qui utilisera le pointeur target avec comme paramètre un caractère quelconque.

Ensuite l'attaquant doit retirer du cache son tableau **channel**, et le pointeur target pour augmenter les chances de mauvaise prédiction de branchement du processeur.

Puis il doit faire pointer target sur la fonction **safe_taget()**, lancer la fonction victime avec cette fois ci l'adresse contenant le secret en paramètre, en espérant que lorsque la fonction victime appelle le pointeur **target** le *BHB* se trompe et exécute la fonction gadget au lieu de la fonction **safe_target()**, ce qui résultera à la mise en cache d'un caractère contenu dans le secret.
Pour finir l'attaquant lance une attaque chronométrée sur le cache pour en extraire le caractère.

III. Resultat

Le résultat observé correspond à la récupération du secret.

Conclusion

Pour performer une attaque exploitant Spectre, l'attaquant devra faire cela sur une machine dont il n'as pas l'accès physique, et sachant que l'emplacement des informations en mémoire est aléatoire, il ne pourra pas analyser une zone spécifique car cela ne servirais à rien. Il devra donc, soit lire aléatoirement des portions entières de la mémoire ou bien lancer une attaque de force brute et donc tenter de lire une bonne partie de la mémoire en prenant le risque de ne pas obtenir autant d'informations que voulu, en plus de perdre une quantité de temps conséquente.

En supposant ensuite que l'attaque ait été un succès; l'attaquant devra ensuite récupérer les données collectées. Et une grande quantité d'information est difficilement transférable sans que la victime se doute de quelque chose.

De ce fait, Spectre reste une faille difficile à exploiter, et il n'y'a eu jusqu'à maintenant aucun cas d'utilisation réele.

Bibliotheque

Gitlab, création et configuration des VM variants 1 et 2
https://etulab.univ-amu.fr/labsec/2021/spectre_VM_builder

Gitlab, création et configuration des VM variant 1 browser attack
https://ident.univ-amu.fr/cas/login?service=https%3A%2F%2Fetulab.univ-amu.fr%2Fusers%2Fauth%2Fcas3%2Fcallback%3Furl

Comment fonctionne le cache du processeur et que sont L1, L2 et L3?
https://commentgeek.com/comment-fonctionne-cache/

Wikipedia, micro-processeur
https://fr.wikipedia.org/wiki/Microprocesseur_multi-cœur

Spectre mitigations in MSVC
https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/

Youtube, Spectre attack explained like you're five
https://www.youtube.com/watch?v=q3-xCvzBjGs

Youtube, How does Spectre work?
https://www.youtube.com/watch?v=2KrodnG9ujM

1. Sécurité : Explication et analyse de la vulnérabilité Spectre
https://blog.engineering.publicissapient.fr/2018/02/23/securite-explication-et-analyse-de-la-vulnerabilite-spectre/

2. Project Zero: Reading privileged memory with a side-channel, Project Zero,
https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with- side.html

3. Spectre Attacks, Paul Kocher, Jann Horn […]
https://spectreattack.com/spectre.pdf

4. Depot Github du programme Spectre variant 1 par jedict1
https://gist.github.com/jedisct1/3bbb6e50b768968c30629bf734ea49c6

Spectre V2 POC, Anton Cao
https://github.com/Anton-Cao/spectrev2-POC

Bulletin d'alerte du CERT-FR
https://cert.ssi.gouv.fr/alerte/CERTFR-2018-ALE-001/

Zombieload
https://zombieloadattack.com/zombieload.pdf

Nouvelle exploitation
https://www.lesnumeriques.com/appli-logiciel/une-faille-spectre-exploitable-dans-google-chrome-et- tous-les-navigateurs-chromium-n161567.html

démo et lien vers git
https://leaky.page/

Google Security Blog
https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html

VirtualBox documentation
https://www.virtualbox.org/manual/

virtualbox forum configure processor
https://forums.virtualbox.org/viewtopic.php?t=84423

VM configuration exemple hashcorp
https://www.vagrantup.com/docs/providers/virtualbox/configuration

Git Attaque Spectre JS
https://github.com/google/security-research-pocs/tree/master/spectre.js