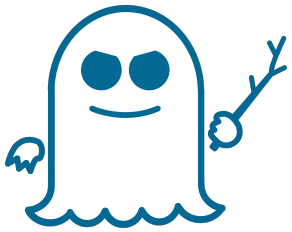
Spectre/Meltdown

© Juliano Mourão Vieira

4 de novembro de 2023



Sumário

1 O que os hackers querem? 1

2 Realizando a leitura 2

3 Extraindo um bit de informação 3
3.1 Marcando a memória com o valor lido 3
3.2 Detectando onde foi o acesso à memória 3

4 Na prática 4

5 Mitigações e correções 5

6 Links para mais informações 6

---

1 O que os hackers querem?

Há muito o que discutir sobre diferentes métodos de ataque a sistemas e como podemos nos proteger, mas um dos objetivos possíveis de um "agente malicioso" é a extração de informações privilegiadas, como senhas, por exemplo.

Uma das maneiras triviais de fazer isto é ler a memória de outro programa ou processo e examiná-la, procurando informações de interesse. Ou seja, se pudermos fazer uma leitura como:

```
dado <= RAM[alvo]
```

poderemos roubar informações. Mas é claro que sistemas operacionais modernos impedem isso. Um programa pode apenas ler os dados que o sistema atribuiu a ele mesmo (a sua região de memória), e se ele tentar acessar

um endereço alvo de outro programa, o hardware vai detectar isso [1] e sinalizar para o sistema, que irá derrubar o processo problemático com um *Segmentation Fault* ou *Access Violation*.

Portanto, se quisermos roubar dados, precisamos contornar essas proteções. Este capítulo explica um *exploit* publicado em 2017 que mostra uma maneira avançada de fazer isso.

O ataque tem dois passos: primeiro, executar a leitura sem gerar erro no sistema; depois, conseguir extrair informações da leitura feita. Nada trivial, e exige conhecimento de arquitetura de computadores para ter sucesso.

2 Realizando a leitura

A ideia, meio insana, é usar *execução especulativa* para fazer a leitura sem tropeçar no sistema. Então fazemos:

```
se (condição)
    dado <= RAM[alvo]
```

de forma que há um branch condicional logo antes da nossa leitura de dados alheios. É possível *treinar* o preditor de branches para garantir que, na próxima passagem por esta condicional, ele vá chutar que nossa "instrução especial" dentro do if será executada. [2] Uma vez que isto esteja feito, garantimos que a leitura vai ocorrer, mas que não será finalizada ("comitada"), seguindo os passos:

```
treina preditor
condição <= falso

se (condição)
    dado <= RAM[alvo]
```

A tecnicalidade aqui é a seguinte: a operação em si de leitura do endereço inválido *irá ocorrer,* mas o erro *não será detectado,* pois acessos indevidos na memória só são efetivamente registrados no commit da instrução ao final do pipeline. Nossa leitura é feita, o dado lido fica pronto dentro do processador esperando o commit, mas quando finalmente nosso branch condicional é resolvido, percebemos a previsão errada e todo o pipeline é limpo com um flush, sem a gravação do resultado e sem gerar o erro.

A questão agora é como extrair alguma informação destas operações.

[1] A detecção é feita ou com uma MPU (*Memory Protection Unit*) ou com uma MMU (*Memory Management Unit*, é uma MPU de luxo que inclui, p.ex., memória virtual); são circuitarias que interfaceiam o microprocessador com a memória principal.

[2] Podemos, por exemplo, setar temporariamente (a) a condição de teste para verdadeira e (b) o alvo para um endereço válido; então repetimos várias vezes este if.

3 Extraindo um bit de informação

O segundo momento de inspiração deste *exploit* é que *a leitura da posição proibida de memória deixa rastros!* A leitura causa um acesso na memória cache, mesmo que o valor dela não seja gravado ao final...

3.1 Marcando a memória com o valor lido

Vamos inicialmente criar um vetor de 128 bytes, preenchendo duas linhas da memória cache (cada linha, atualmente, ocupa 64 bytes de memória; é possível alinhar o vetor com as linhas, usando o compilador).

Depois vamos isolar o bit menos significativo do valor lido (o LSB, fazendo um AND com 1), *ainda especulativamente,* e vamos acessar o nosso vetor especulativamente também. Temos então:

```
char vec[128]

treina preditor
condição <= falso

se (condição)
    dado <= RAM[alvo]
    bit <= dado & 1
    "leia" vec[64*bit]
```

Note que tanto a escrita em `dado` quanto em `bit` serão descartadas pelo microprocessador. O acesso ao vetor será feito em `vec[0]` ou `vec[64]`, de acordo com o valor do bit LSB gravado no endereço `alvo` da memória RAM. Embora a operação em `vec` não seja gravada pelo processador, *a linha de memória acessada será copiada para a cache,* e isso pode ser explorado.

3.2 Detectando onde foi o acesso à memória

Esta parte é a mais fácil: vamos inicialmente limpar a cache, garantindo que qualquer dado lido subsequentemente cause uma cópia de uma "nova" linha da memória principal para a cache, o que é feito com alguma forma de cache flush. [3]

Dentro da execução especulativa a ser descartada, iremos ler uma das duas linhas da cache, dependendo do bit que já extraímos. A linha lida ainda estará presente na cache depois do pipeline flush; a outra linha estará ausente. O que faremos então é ler um dado qualquer da 1ª linha e ler um dado qualquer da 2ª linha, mas *vamos cronometrar estes acessos* (há

[3] Há uma instrução `clflush` no x86-64, por exemplo. E é possível posicionar cuidadosamente as linhas de cache utilizadas pelo vetor de 128 bytes, com alguma escovação minuciosa de bits.

contadores internos no processador que podem ser utilizados). Uma das linhas terá acesso rápido, pois já estava na cache; a outra terá acesso mais lento, pois precisa ser copiada da memória principal.

Aquela que for mais rápida indicará o bit lido especulativamente com este nosso esquema. Este tipo de ataque se chama *side channel attack,* pois utiliza algum "canal secundário" de informações lidas indiretamente, que não é o primário (execução das instruções e seus efeitos desejados ou a construção de um algoritmo); neste nosso caso, são tempos de acesso à memória.

```
char vec[128]

treina preditor
condição <= falso
flush cache

se (condição)
    dado <= RAM[alvo]
    bit <= dado & 1
    "leia" vec[64*bit]

t0 <= cronometra_leitura(vec[0])
t1 <= cronometra_leitura(vec[64])

se (t0>1)
    BIT LIDO FOI 0!
senão
    BIT LIDO FOI 1!
```

E está feito. Este é o Meltdown; o Spectre é um pouco diferente, seguindo ideias similares.

4 Na prática

É bastante simples generalizar nosso esquema de roubar apenas um bit; se fizermos aquele AND com o número 2 ao invés de 1, obteremos o bit seguinte; usando 4, 8, 16... conseguimos uma palavra completa de 32 bits. (Os artigos, palestras e vídeos do Youtube começam com mais de um bit, o que confunde um pouco nas tecnicalidades de fixação de endereços do vetor correspondentes às cache lines e etc.). Se repetirmos o procedimento mais vezes, podemos ler grandes blocos de memória, embora esta operação seja bastante lenta.

Algumas demonstrações foram feitas à época, com um processo roubando informações (strings) de outro, inclusive um Javascript em uma aba

do Firefox roubando dados de outra aba. Portanto, trata-se de uma vulnerabilidade com potencial verdadeiro de aplicação, não apenas algo de interesse acadêmico, embora nunca tenha sido observado "solto na natureza," como dizem, ou seja, utilizado em um ataque real.

Obviamente não é algo tão simples de ser explorado – o hacker ou precisa ter uma ideia do que pode ser roubado e onde se encontra na memória global ou então precisa de bastante tempo para copiar uma montanha de informações e procurar algo útil lá. Mas a história nos ensina a não duvidar da persistência destes "atores malignos," especialmente quando há grana envolvida.

Uma coisa que chama bastante atenção é que o programa mal-intencionado pode rodar com direitos de um usuário normal (não precisa fingir ser administrador) e pode acessar dados do kernel, sem ter os direitos de supervisor. Esta característica é crítica.

Também dê uma olhada nas informações do meu PC comprado em 2017, via Linux (cortei fora coisas inúteis):

```
juliano@pop-os:~$ cat /proc/cpuinfo
processor : 0
...
model name : Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
...
bugs : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass
 l1tf mds swapgs itlb_multihit srbds mmio_stale_data
 retbleed gds
...
cache_alignment : 64
```

Praticamente todos os processadores com memória cache e execução especulativa produzidos antes de 2018 são vulneráveis a ataques como o Spectre ou o Meltdown e suas variantes, que inauguraram em toda uma classe de vulnerabilidades.

5 Mitigações e correções

Os detalhes de implementação omitidos anteriormente são mais complexos do que parecem. Em especial, o atacante não consegue garantir determinados resultados nos passos, ou então precisa de muita sorte (ou seja, um número de repetições que é inviável, de tão grande).

Mas é importante notar que esta vulnerabilidade *não possui conserto generalizado.* Ela é um bug das arquiteturas superescalares especulativas modernas, um artefato arquitetural, e *está presente em todos os processadores de alto desempenho.* Isso inclui Intel e AMD com seu x86-64, mas também

inclui parte dos ARM mais avançados, os IBM POWER e até, potencialmente, os RISC-V. Algumas variantes possuem conserto em processadores contemporâneos, outras não.

As mitigações, ou seja, procedimentos de reforço de segurança, buscam dificultar ou impedir vias de acesso utilizadas nos passos dos *exploits*, através de mudanças no kernel do sistema operacional e/ou na ISA do processador. Por exemplo, o pessoal do Kernel do Linux nos informa que "Currently the only known real-world BHB attack vector is via unprivileged eBPF" [4] para a variante 2 do Spectre, que usa o *Branch History Buffer* para mexer na predição (ao invés de, p.ex., o *Branch Target Buffer*). Como o único acesso conhecido para esta manipulação é usar o *Extended Berkeley Packet Filter* – eBPF – no modo usuário (*unpriviledged*), normalmente o kernel desabilita este modo de utilização. Isto, é claro, não impede que novas formas de manipulação sejam descobertas.

Argh.

Mas certas soluções são convincentes e não muito complexas: por exemplo, algumas limitações no acesso à memória usando instruções de barreiras (*fences,* que obrigam uma ordem definida nas instruções de leitura e escrita) são suficientes para impedir algumas variantes.

6 Links para mais informações

Se eu não recomendasse a Wikipédia em inglês, não seria eu. Mas também existe o site oficial do Spectre/Meltdown, feito pelos próprios pesquisadores, que tem bastante informação.

Tem muitos vídeos no Youtube também, mas geralmente eles gastam vinte minutos explicando branch prediction e funcionamento das caches, o que é um pé no saco. Mas tem alguns muito bons e outros bem profundos.

Por fim, sugiro dar uma pesquisada em *Side Channel Attacks,* incluindo o Zenbleed da AMD, descoberto com uma técnica interessante; o Rowhammer; e alguns assustadores que quebram a segurança de sistemas embarcados medindo variações na corrente consumida pelo processador e cronometrando a execução de comparações (tipo descobrindo a senha de dois cofres eletrônicos).

[4] Essa sopa de letrinhas mostra o quanto é difícil entender as entranhas destes problemas; a citação é de `https://docs.kernel.org/admin-guide/hw-vuln/spectre.html`. Este documento tem vários exemplos de utilização do Spectre e suas mitigações.