CPU_v.01.01

31기 육은서

기술 배경 및 소개

취약점 뉴스 스크랩하다가 한 번 공부해봐야겠다 생각한 주제이다.

CPU에는 대표적으로 두 가지 보안 취약점이 있다. 오늘은 그중 하나인 멜트다운에 대해 소개하려고 한다.

기술 탐구 및 분석

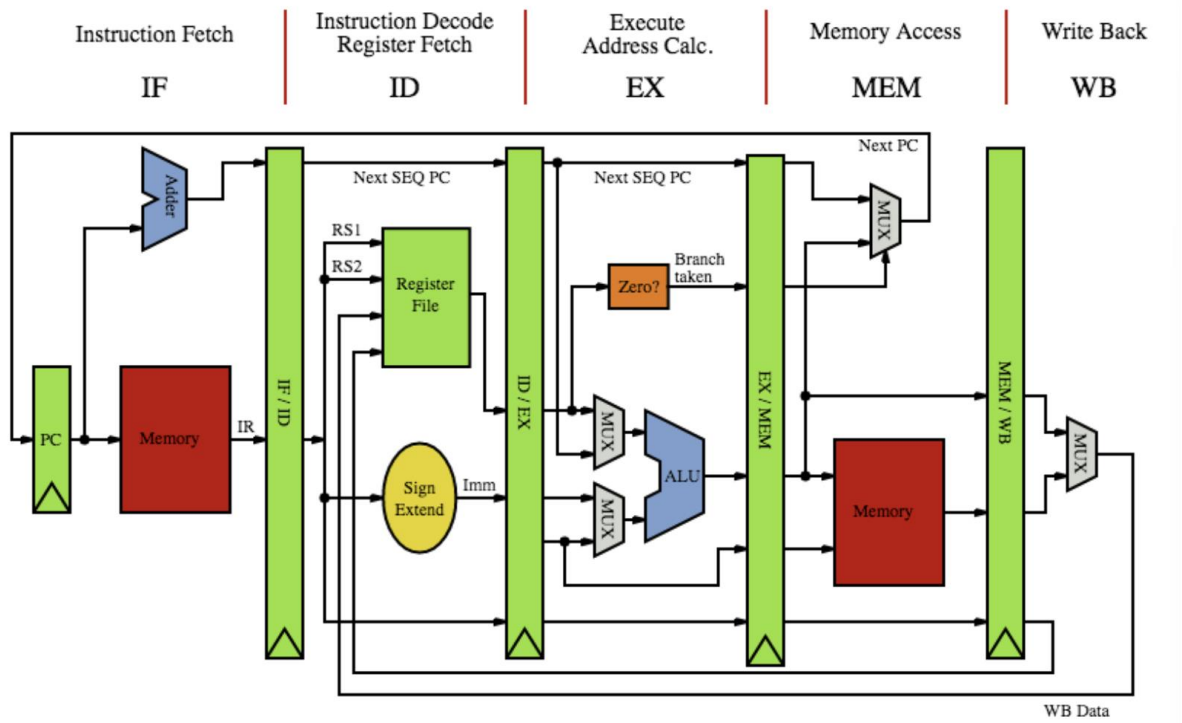
1. parksb님의 깃허브 그대로 읽어보기

멜트다운은 2018년 초에 보고되었다. 인텔 프로세서에서 발견된 취약점이다. 참고자료를 보면 parksb님의 깃허브를 중심으로 부족한 부분을 채워서 작성하였다. 그대로 인용해보자면, '비순차적 명령 실행의 맹점을 이용해 접근할 수도, 접근해서도 안되는 데이터에 접근하고, 캐시와 같은 사이드 채널을 통해 정보를 알아낸다'라고 써져있다.

- 비순차적 명령 실행은 무엇인가? 0
- 접근할 수도, 접근해서도 안되는 데이터? CPU에 이런 데이터가 무엇인가? 0

일단 계속해서 읽었다. Background에서는 프로세서가 명령을 처리하는 과정에 대해 짚고 넘어간다.

하드웨어 레벨에서 프로세서의 데이터패스(Datapath)를 도식화하면 아래 그림처럼 된다:

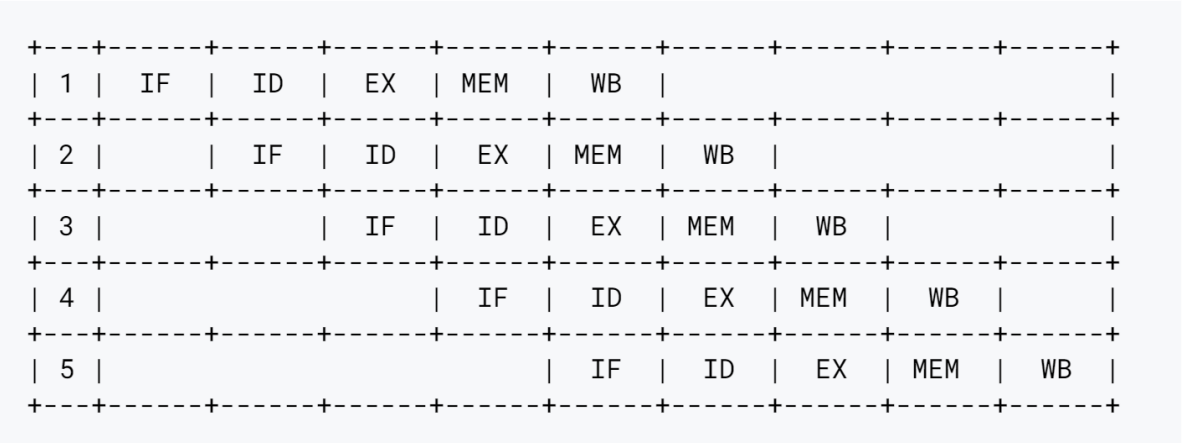


맨 위에 IF, ID, EX, MEM, WB로 과정을 나누었다.

1. IF: 바이너리 형식의 명령을 메모리에서 CPU로 가져온다..
2. ID: 명령을 디코딩하여 판단 후 레지스터에 입력한다.
3. EX: 조건문이라면 zero?인지만 확인 후 넘기고 그렇지 않다면 ALU를 거친다.
4. MEM: 여기서는 메모리에서 데이터를 가져오거나 입력한다. 이런 과정이 필요 없다면 아래 과정으로..
5. WB: 메모리와 교류한 명령(혹은 아닌 명령)은 이부분에서 레지스터로 넘긴다.

이때 한 명령이 어느 한 곳에 있을 때 다른 곳이 쉬고 있으면 비효율적이다(무조건 일해!!) 명령들이 각각의 단계에 존재하여 병렬적으로 명령을 실행하도록 만든 것이 파이프라인이다.

명령 처리에 5단계가 존재하니까 한 사이클에서 5개의 명령을 동시에 실행할 수 있다.



가운데에서 진행되는 단계가 5개로 최대이다. 한 번에 5개 명령을 처리한다니 아주 효율적이라고 생각했지만 그렇지 않은 경우가 있다고 한다.

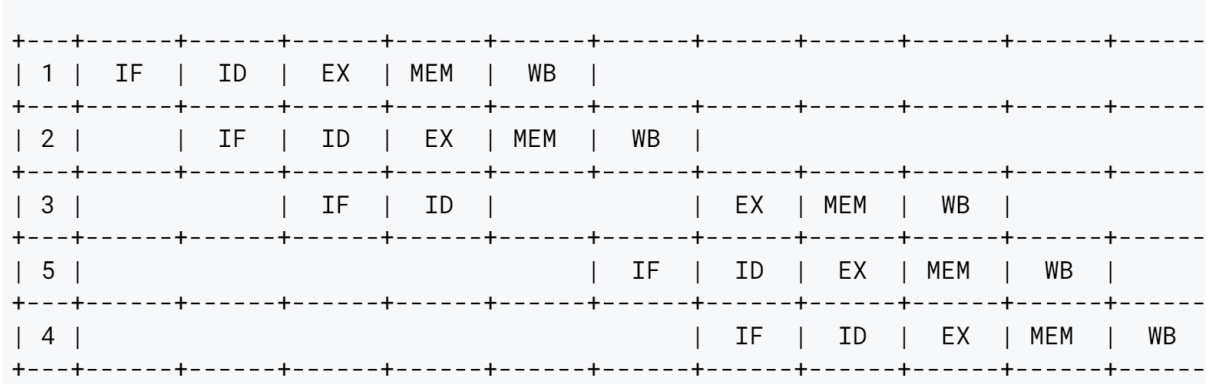
```
1:  lw $t0, 0($sp)
2:  lw $t1, 4($sp)
3:  and $s1, $t0, $t1
4:  add $s2, $t0, $s1
5:  addi $s3, $t0, 20
```

명령1,2는 메모리에서 데이터를 가져와 각각 $t0, $t1에 저장하는 명령어라고 하고, 명령3,4,5는 뒤의 두 인자의 합을 첫번째 레지스터에 저장하는 명령이다. 저장되고 사용되는 레지스터를 잘 살펴보면 의존해야 하는 명령이 있다(약간 부모 클래스 생각하면 되는 거 같다. 먼저 있어야 나도 있는..)

$t0이 쓰이는 명령: 명령3,4,5

$t1이 쓰이는 명령: 명령3

순차적으로 실행하면 명령 1,2,3,4,5 순이지만 어떤 명령은 필요한 값을 가져오는 동안 머물러 있어야 해서 지연이 발생할 것이다. 그래서 다음과 같이 처리한다고 한다.



1,2,3,5,4순으로 실행하였다. 5는 $t1과의 의존성은 없으므로 4보다 먼저 실행하여서 4에게는 $t1이 올 시간을 벌어주고, 5는 더 빨리 연산을 처리하도록 만든 것이다. 이렇게 순차적으로 처리하지 않는 명령 실행을 비순차적 명령 실행이라고 한다.

인텔 프로세서는 의존성이 없는 명령을 모아 같은 사이클에 돌린다. 이 기술을 Superscalar라고 한다.

백그라운드가 끝났다. 멜트다운 취약점 공격의 개요는 다음과 같다.

1. 실행하면 안되는 코드를 비순차적 실행
2. 캐시에 올라간 데이터 접근
3. 정보 유출!

A Toy Example(PoC)

```
Raise_exception();
//the line below is never reached
Access(probe_array[data*4096]);
```

Raise_exception 함수에서 예외를 일으키면 access 함수는 실행되지 않는다. 하지만 비순차적 명령을 실행하면 raise_exception 함수보다 access 함수가 먼저 실행된다. 여기서 access 함수에 접근하면 안되는 인자를 넣는다면..?

Data를 접근할 수 없는 메모리 공간으로 설정하고 페이지 크기인 4096을 곱하여 베이스 주소에 접근한다.

- Cpu의 페이지와 베이스 주소 0

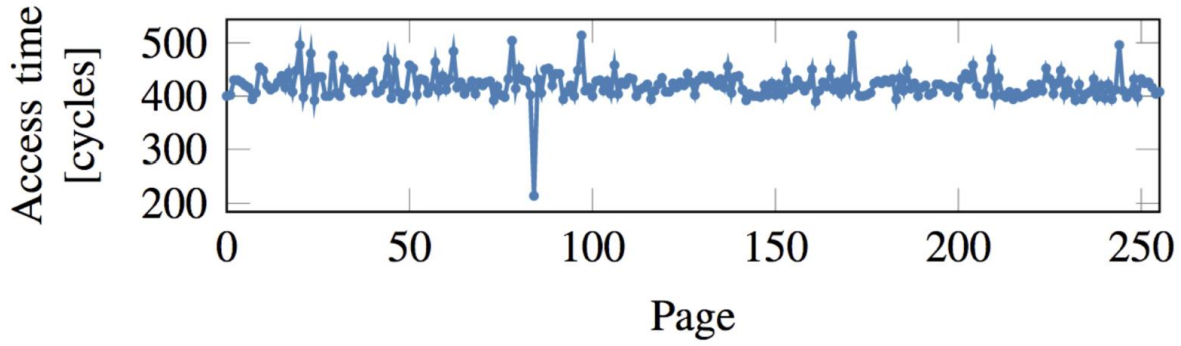
순서를 다시 한 번 짚고 넘어가면 다음과 같다.

1. 비순차적 실행으로 인해 access 함수 실행
2. Raise_exception에서 예외 발생

2번이 진행되어도 1번 과정에 의해 캐시에 data*4096이 저장된다.

- 캐시가 뭐지? 0

이후에는 사이드 채널 공격이 이루어진다.



- 블로그 설명이 모호한 부분이 있어서 더 자세한 설명 찾아야 할 거 같다. 0

실제 공격의 핵심 코드는 다음과 같다(x86 어셈블리 명령)

```
; rcx = kernel address, rbx = probe array (주석이다)
xor rax, rax
retry:
mov al, byte [rcx]
shl rax, 0xc
jz retry
mov rbx, qword [ rbx + rax ]-
```

Retry문 아래 코드부터 보기

1. rcx를 로드하며 al로 표현된 rax 레지스터에 저장 (al과 rax는 호환된다..)

2. 유저 모드에서는 커널 메모리에 접근할 수 없다. 주석에서 설명했듯이 커널에 있는 값을 요구하므로, 예외 발생.

3. 비순차적 명령 실행으로 인해 다음 명령 shl rax, 0xc 명령 실행하면 rax값에 0xc(4096 바이트를 나타냄)을 곱한다.

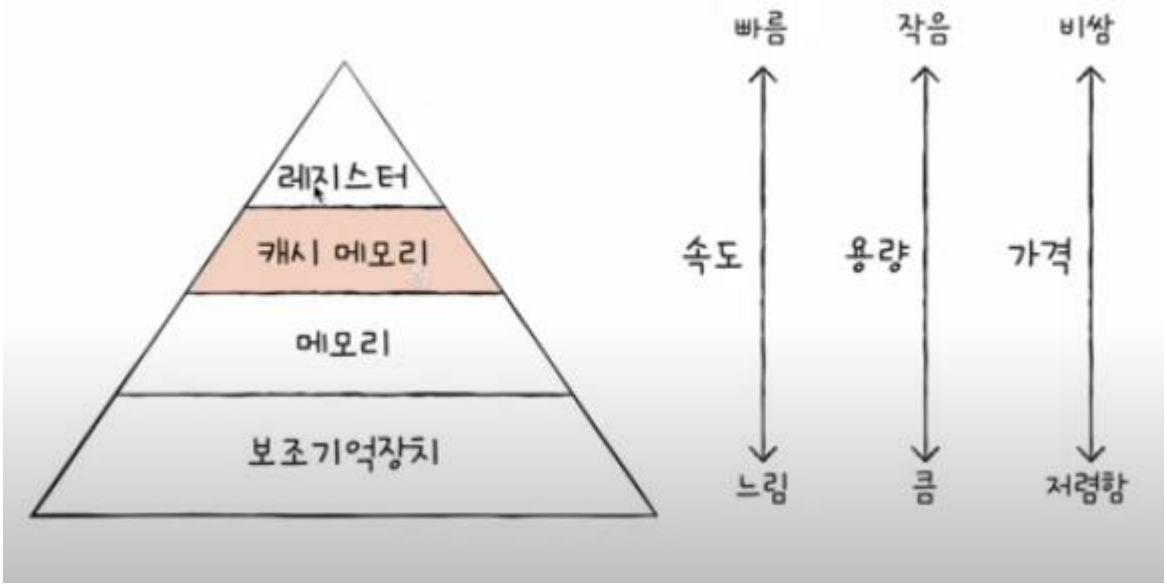
4. jz retry를 건너뛰고 mov rbx, qword [rbx+rax]를 실행한다. 이러면 베이스 주소에 4096 바이트가 곱해진 rax값을 더한다. Rbx=rbx+rax*0xc가 되는 셈.

5. 예외가 발생했기 때문에 위 과정이 취소가 되지만, rax번째 베이스 주소는 L1 캐시에 올라간다.

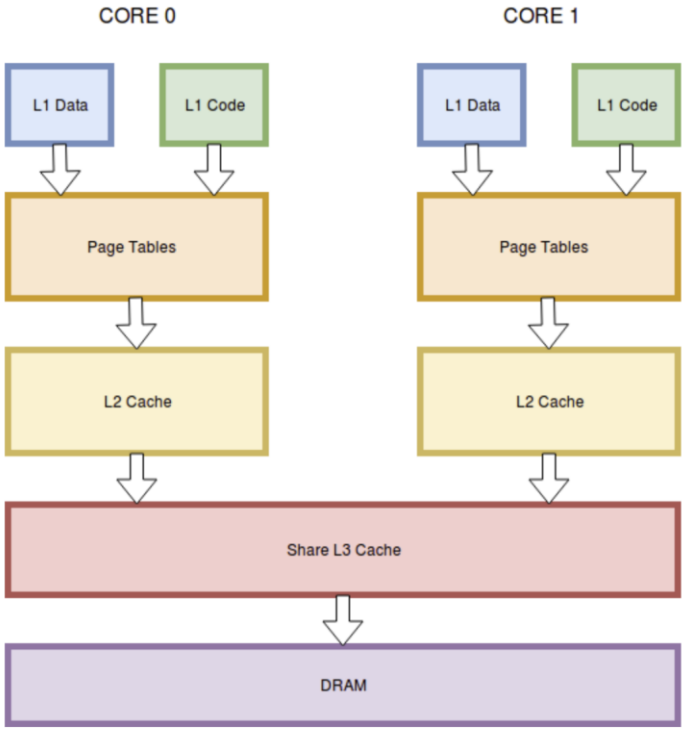
6. 이제 페이지들을 모두 훑으면서 접근 시간을 측정한다. rax번째 페이지라면 캐싱이 되어있는 페이지이기 때문에 유독 시간이 짧을 것이다.

2. 캐시 메모리와 페이징 기법

CPU가 메모리에 있는 내용을 로드하는 과정을 알아야 한다.



레지스터는 CPU 안에 있는 작은 메모리이다. 메모리와 레지스터 사이를 보면 캐시 메모리라는 것이 존재한다. 캐시는 임시 저장소로 생각하면 된다. 병목 현상을 줄이기 위한 중간 계층이다. CPU에서 메모리의 데이터를 로드할 때 virtual address를 physical address로 변환해야 한다.

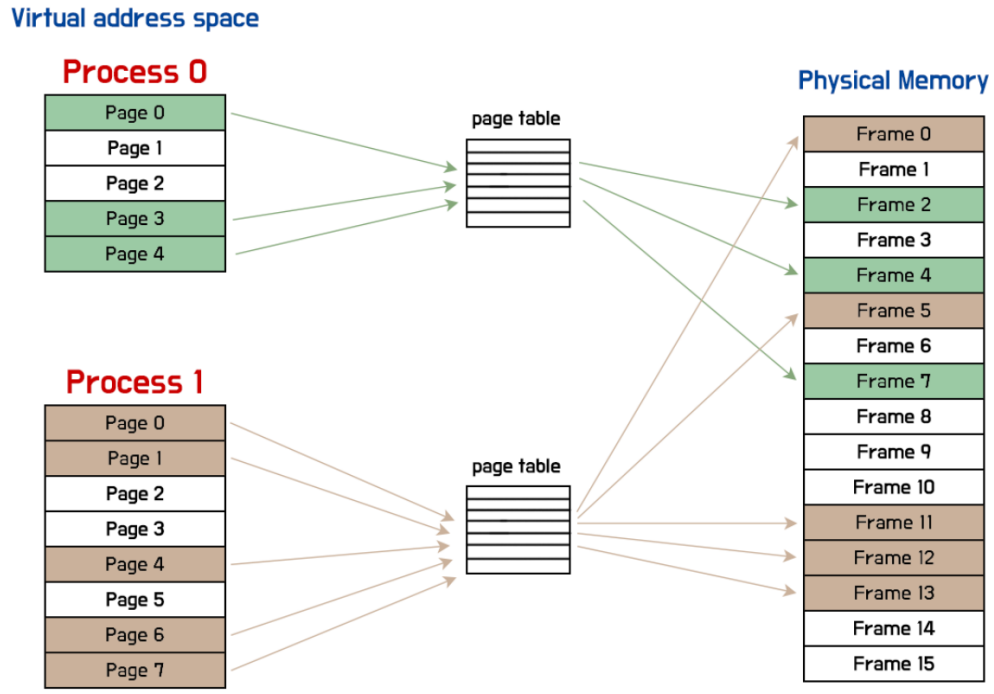


▲CPU Core의 하위시스템을 단순화한 그림

데이터는 가장 먼저 L1 캐시를 거친다. 만약 L1 캐시에서 필요한 데이터를 찾을 수 없다면 DRAM에 요청한다. 요청 과정을 표현한 것이 위 그림인데, 중간에 page tables를 거치는 걸 알 수 있다. 이부분에서 virtual address를 physical address로 변환하는데 그중 한 단계가 권한 검사이다. 보호비트를 활용하여 접근할 수 있는 권한인지 확인한다. 아마 이부분에서 예외로 처리되는 거

같다(권한 탈락)

이러면 이제 베이스 주소는 무슨 베이스 주소인가, 페이지에 대한 내용만 남았다.



위 그림처럼 virtual address에서 프로세스의 메모리를 자르는 단위를 페이지라고 한다. 페이지 테이블을 거친 physical memory에서도 같은 단위로 쪼갠다. 1프로세스 1페이지 테이블이 원칙이다. 그리고 이 페이지 테이블에는 각 페이지의 base address를 저장한다. 페이지와 대응하는 physical address를 말한다. 해당 환경에서 probe array의 크기는 256 페이지 길이만큼인데, 각 페이지는 크기가 4096 바이트이다. 길이가 256인 페이지에서 타겟 페이지의 데이터를 캐싱하고, 0부터 255 페이지까지 실행하여 짧은 시간을 찾는다(3번에서 자세히..)

3. 사이드 채널 공격에 대하여

조금 더 자세히 알아보면 좋을 거 같아서 논문 참고했다(영어가 싫어서 미뤘다) Step 3은 "The attacker uses Flush+Reload to determine the accessed cache line and hence the secret stored at the chosen memory location"이라고 소개한다. 해당 논문에서 사용한 부채널 공격, Flush+Reload 공격에 대해 알아보자.

아까 256 페이지를 하나하나 다시 접근한다고 했다. 만약 공격자가 접근한 페이지가 미리 캐싱된 타겟 페이지였을 때 Flush+Reload 공격을 할 수 있다. 타겟과 공격자가 같이 있다면, 타겟의 메모리를 clflush로 캐시에서 비워버릴 수 있다. 이후 다시 타겟을 로드하는데(reload), 타겟이 flush 했던 페이지에 접근한다면 시간이 적게 걸릴 것이다.

4. 왜 비순차적 실행을 하는가?

위에서 참고한 블로그 글이 조금 비약적이라 생각해서 자료를 더 찾았다. '잘못된 추측 실행(Abusing Speculative Execution)'에 대한 내용을 넘긴 거 같다.

추측 실행은 아까 의존성 있는 명령과 관련이 있다. 다음 명령이 의존적이라면 CPU는 결과를 추측하고(분기 예측) 맞는지 판단한다. 맞다면 계속해서 수행, 맞지 않다면 파이프라인을 비우고 재실행한다.

예외 처리에 대해 자세히 봐야한다. 명령을 실행(시작)한다. Kernel address는 레지스터에 commit 되지 않는다. 이러면 예외가 발생한다. 이때 commit이 되지 않는다는 건 user에게 보여주지 않는다는 뜻으로, 이미 실행되었기 때문에 데이터를 로드했다고 볼 수 있다(로드한 데이터를 user에게 보여주지 않음)

```
Raise_exception();
//the line below is never reached
Access(probe_array[data*4096]);
```

정확한 이해를 위해 코드를 다시 가져왔다. Raise_exception()을 1번 함수, access를 2번 함수라고 한다.

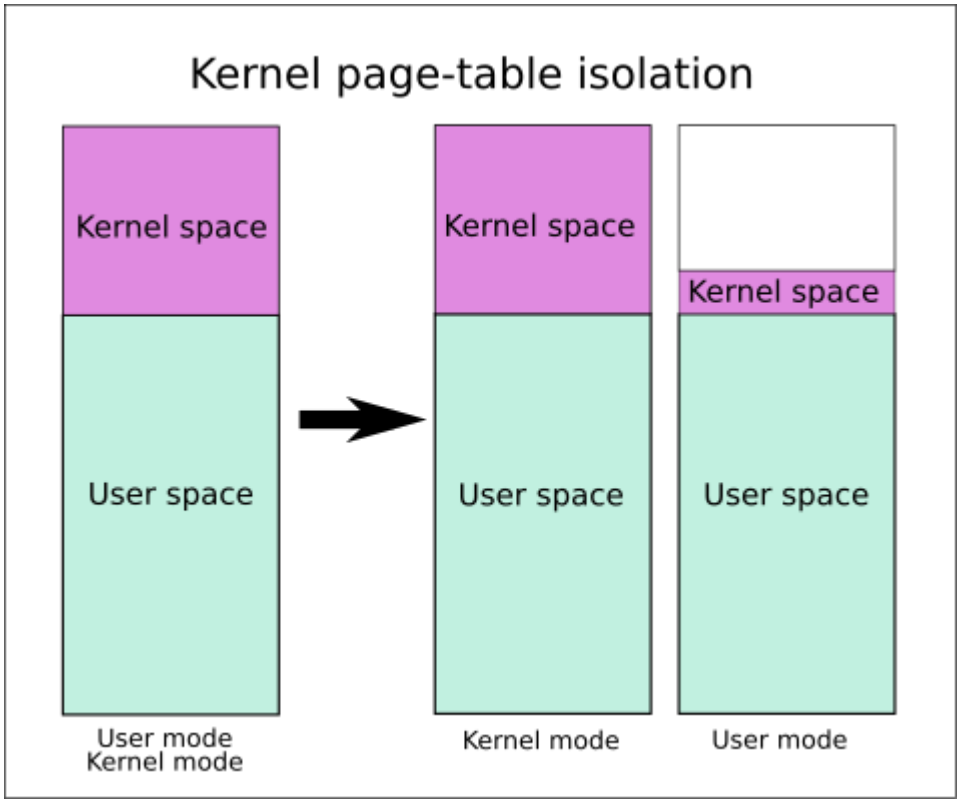
1. 1번 함수에서는 절대 예외가 나온다. -> 실행하면 안되는 명령
2. 추측 실행으로 인해 access는 실행되었다.
3. 실행된 명령에 의한 결과값은 L1 캐시에 저장된다. -> 이부분을 드러내면 정보 유출!

5. 접근할 수도, 접근해서도 안되는 데이터?

위 데이터는 커널에 있는 모든 자원을 뜻하는 것 같다. 커널에서 진행되는 과정이 노출되면 정보 유출은 당연하고 어떤 프로그램을 실행하고 암호문, 평문의 데이터를 모두 알 수 있다.

6. 패치

리눅스에서는 KPTI를 적용하였다. Kernel Page-Table Isolate의 약자로, 페이지와 테이블을 분리하는 기술이다. 기존에는 page table에 kenel(physical) address가 매핑되어 있었다. 그래서 속도 면에서는 빨랐으나 멜트다운으로 인해 다운그레이드를 적용하고 패치해야 했다.



아주 가시적으로 잘 보여주는 그림이다. Page table에서는 user, kernel 모드에 따라 정보를 다르게 준다. 오른쪽 User mode를 보면 왼쪽 Kernel mode에 비해 kernel space 범위가 좁은 것이 보인다.

용어정리

Probe array: 유저가 접근할 수 있는 영역, 256페이지를 접근할 수 있다(아스키 코드를 의미)

분기 예측:

느낀 점

쌀..을 재제출 하면서 공부한 거 그대로 담았으면 좋겠다는 조언을 듣고 진짜 내 공부 흐름대로 담았다. 그래서 뭔가 정보전달 목적으로 보면 가독성이 매우 떨어진다고 생각하는데 이렇게 작성해도 되는건가 싶긴 하다.. 개인적으로는 글을 읽으면서 드는 생각이나 질문을 적으니까 공부에는 더 도움되고 갈래가 잡히는 거 같다(스윙로그 같은 경우에는 다르게 써야 할 거 같다)

다음 주차 내용

아직 확실하게 정하진 않았는데 스타블리드, 블루본.. 아니면 썬더볼트 포트 관련한 펌웨어 공격에 대해 알아볼 거 같다.

참고 자료

CPU 보안 취약점을 공격하는 아주 구체적인 원리 . (n.d.). https://parksb.github.io/article/31.html.

캐시가 동작하는 아주 구체적인 원리 . (n.d.). https://parksb.github.io/article/29.html.

Moritz Lipp . (n.d.). *Meltdown: Reading Kernel Memory from User Space* . n.p.: Google Project Zero.

메모리와 메모리 관리 . (n.d.). https://velog.io/@prettylee620/%EB%A9%94%EB%AA%A8%EB%A6%AC%EC%99%80-%EB%A9%94%EB%AA%A8%EB%A6%AC-%EA%B4%80%EB%A6%AC.

[보안 Issue] Meltdown(멜트다운) 취약점을 파헤쳐보자. . (n.d.). https://sata.kr/entry/%EB%B3%B4%EC%95%88-Issue-Meltdown%EB%A9%9C%ED%8A%B8%EB%8B%A4%EC%9A%B4-%EC%B7%A8%EC%95%BD%EC%A0%90%EC%9D%84-%ED%8C%8C%ED%97%A4%EC%B3%90%EB%B3%B4%EC%9E%90-1.

추측 실행(Speculation Execution), 분기 예측(Branch Prediction) . (n.d.). https://rmagur1203.tistory.com/17.

[특별기고] CPU 취약점 종합보고서 ①: 취약점 기본원리 . (n.d.). https://library.gabia.com/contents/infrahosting/5412/.

[OS] CH3-3.1 Virtual Memory . (n.d.). https://velog.io/@woo0_hooo/OS-CH3-3-Virtual-Memory.

[운영체제] 페이징(Paging)이란? 페이지 테이블이란? . (n.d.). https://code-lab1.tistory.com/55.

[특별기고] CPU취약점 종합보고서②: MELTDOWN과 SPECTRE VARIANT 1 . (n.d.). https://library.gabia.com/contents/infrahosting/5425/.

[OS/운영체제] 페이징 (Paging) - (1). . (n.d.). https://4legs-study.tistory.com/48.

성능과 바꾼 보안 Part. 1: Meltdown . (n.d.). https://hashmm.com/post/perf-and-security-meltdown-part1/index.html.