Exploit Linux kernel eBPF with side-channel

Jinbum Park

jinb.park7@gmail.com

jinb-park.github.io

eBPF
(extended Berkeley Packet Filter)

- eBPF (extended Berkeley Packet Filter)
  - Linux kernel 에 존재.
  - Linux kernel 에서 Packet Filtering 을 위해서 사용됨.
  - How?? □ User 에서 일종의 스크립트 프로그램과 같은 eBPF Program 을 작성하여 Linux Kernel 로 전달. Linux Kernel 에서 각 패킷에 대해 이 프로그램을 실행시켜서 필터링 함.
  - 즉, User 에서 작성한 프로그램이 Linux kernel 에 삽입되어, Packet 처리될 때마다 Linux kernel 내부에서 실행되는 것.
  - 이를 위해 eBPF Program 을 위한 Instruction set 이 따로 있음.

Restrictions of eBPF program

- User 에서 Kernel Memory 를 Read / Write 하는 프로그램 작성하여 실행시키면 안되므로, eBPF Program 동작시에는 제한된 Memory 에만 접근 가능함.

- eBPF Prgram 에서 접근가능한 메모리는 아래와 같음.

  (1) 현재 처리중인 Packet 정보

  (2) Map (eBPF Program 에서 사용하기 위해 User 가 정의한 데이터)

  (3) Stack (Stack frame pointer 접근 가능)

- User 는 eBFP Program Input 을 주거나 Output 을 받는 용도로, eBPF Map 정의할 수 있음.

Verifier of eBPF Program (in Linux kernel)

(1) Program Loading-Time

- Loop 없는지 검사.
- Load/Store 명령어가 허용된 메모리만 접근하는지 검사.
- 프로그램의 모든 명령어를 처음부터 끝까지 simulation 해봄.
  (in-order-execution)
- …..

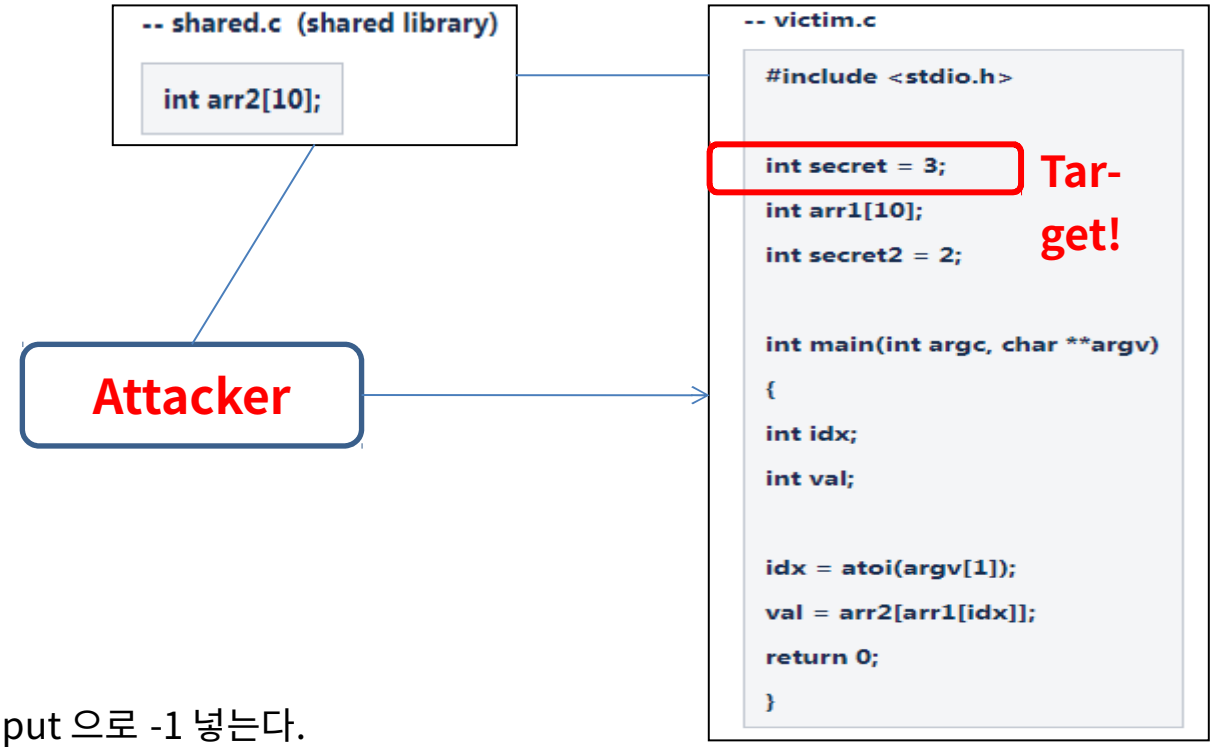
(2) Program Run-Time

- eBPF 프로그램 실행 전/후 register, stack 상태 검사.
- array out-of-bounds 검사 추가.
  e.g) val = arr[idx]; if (idx >= 0 && idx < size) val = arr[idx];
- ……

Spectre variant 1 (CVE-2017-5753)

---- bounds check bypass

---- explore a vulnerable sample code



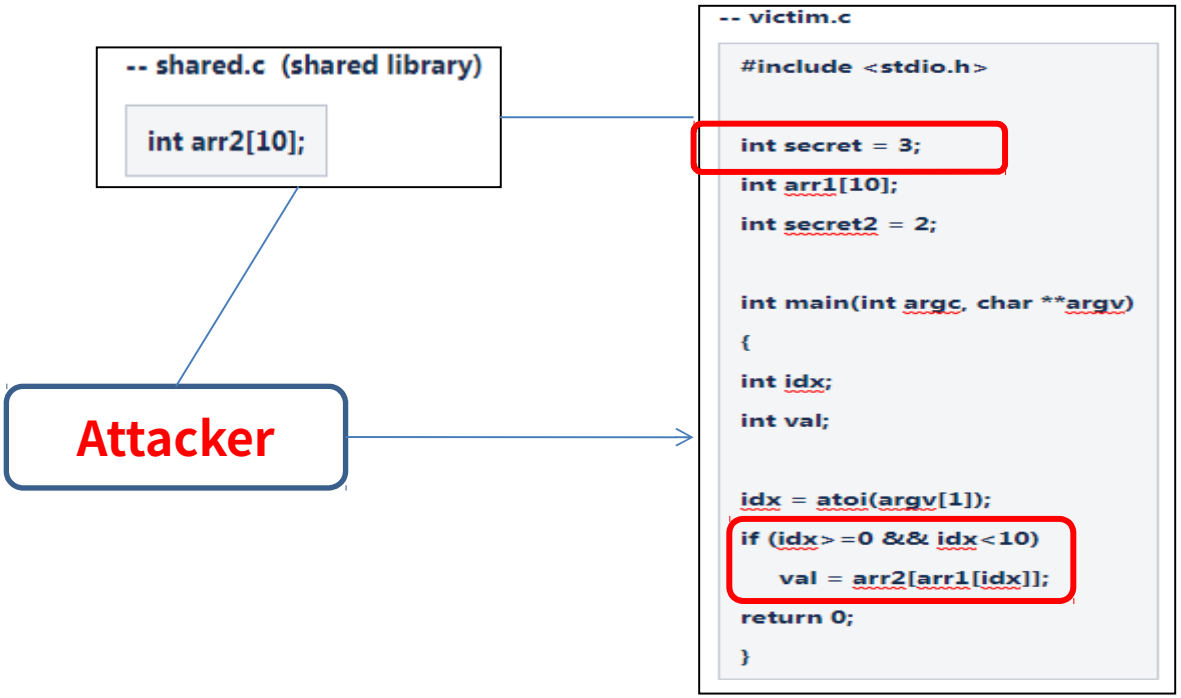
- **공격 순서**

(1) Attacker 는 Victim 에 input 으로 -1 넣는다.

(2) arr2[arr1[-1]] 실행됨 ☒ arr2[secret] □ arr2[3] 접근.
arr2[3] 이 접근되었으므로, arr2[3] 이 CPU cache 에 올라감.

(3) Attacker 는 arr2 의 모든 메모리를 접근.
이 때, arr2[3] 은 cache 에 있으므로 빨리 접근됨. 다른 인덱스는 느리게 접근됨.

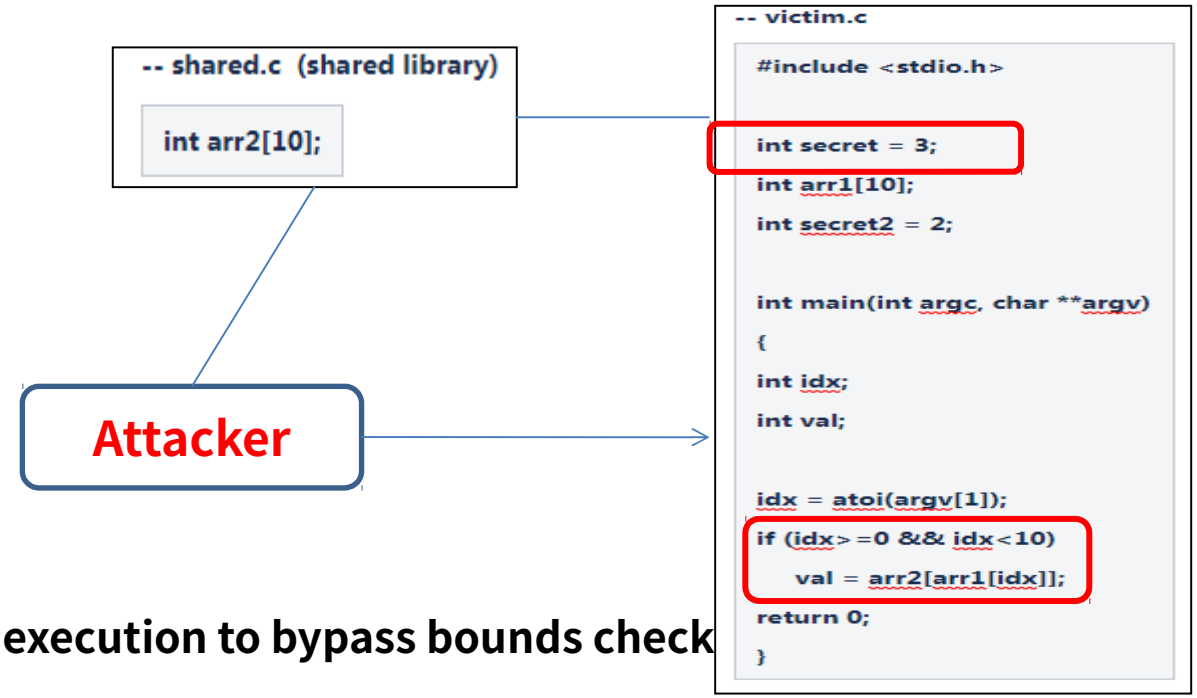
(4) Attacker 는 secret 값을 “3” 으로 추론 가능.



- Victim 에서 위와 같이 bound check 코드를 추가한다면?? 이전에 말한 공격은 실패함…

- 위와 같이 안전한 bound check 가 추가되어 있더라도, 공격할 수 있는 방법은??

  ⇨ Spectre variant1 을 이용!! 이를 활용하면 out-of-bound index 로 if문 실행 가능!!



- **Exploit speculative execution to bypass bounds check**

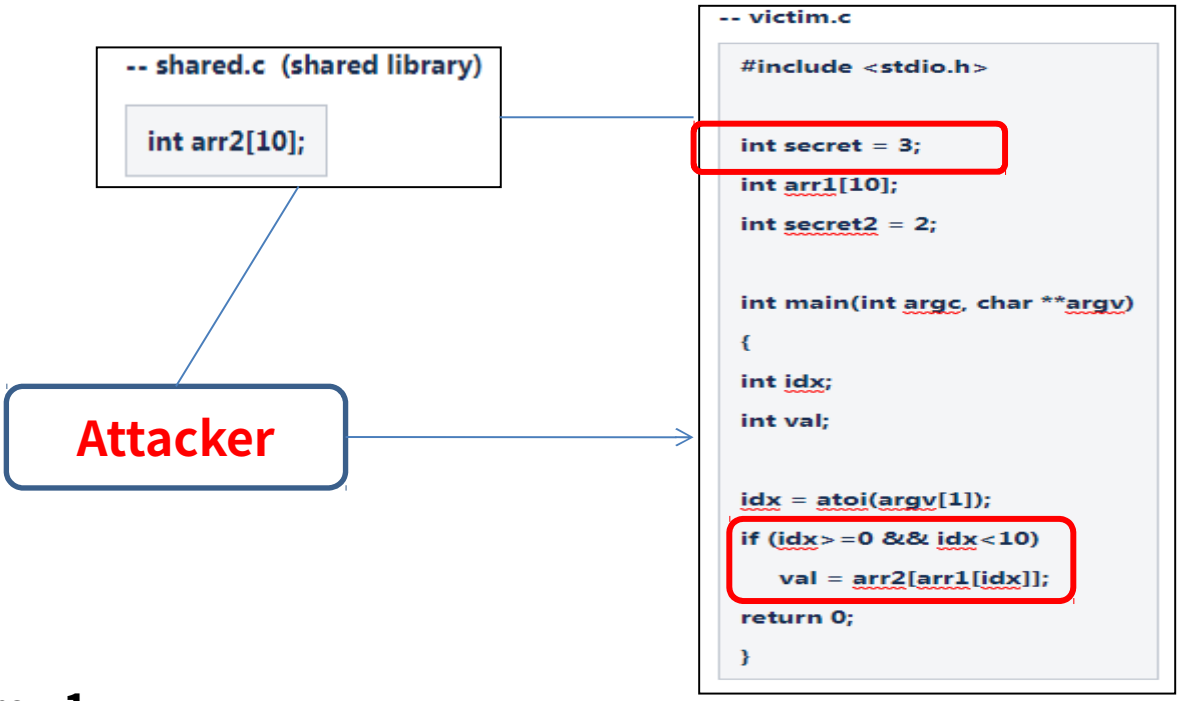
- CPU branch predictor 는 conditional branch 에 대한 예측 실행을 제공한다. 이를 이용하면.

(1) idx == 0 을 넣어서 100번 실행.

(2) if 문 실행 시. CPU 는 여기서는 if 문 분기가 훨씬 많이 일어났으므로, if 문으로 들어갈 거라고 예측함.

(3) idx == -1 넣고 실행. Branch predictor 에 의해 if 문 실행됨.

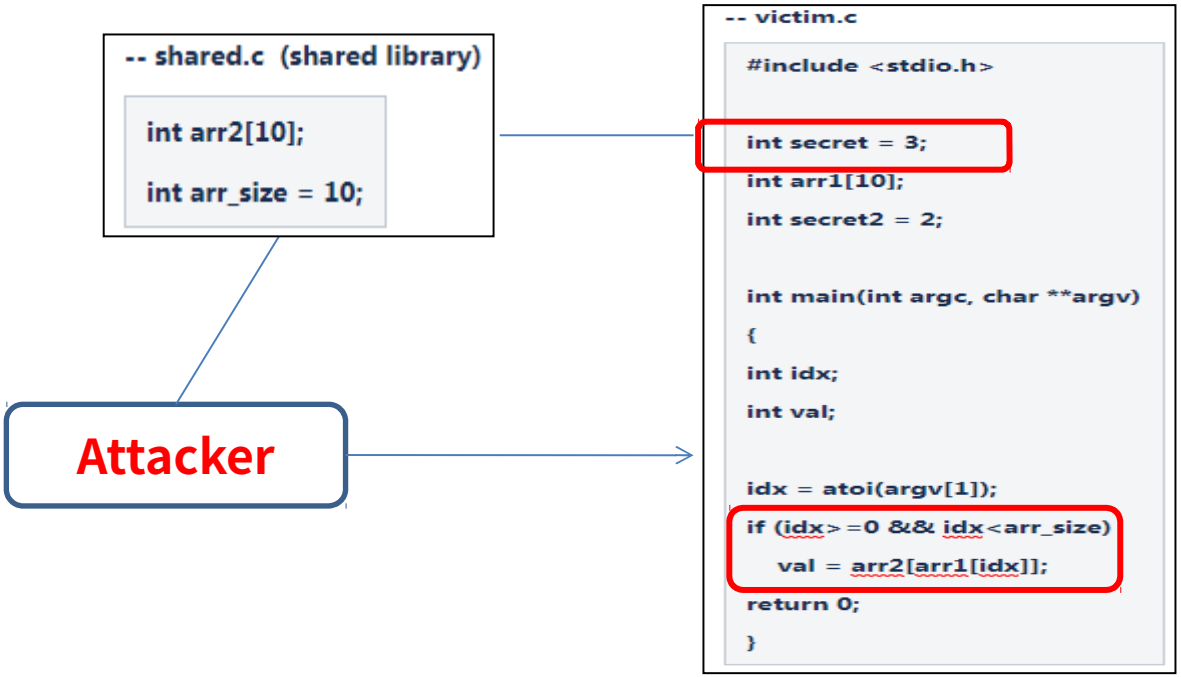
(4) CPU 는 나중에 예측이 잘못되었다는 것을 인지함. If 문 실행 취소 후, 원래 else 문 실행. 이 때 실행결과가 register 에는 남지 않지만, cache 에는 남아있음.



- **Key insight of spectre v1**

(1) branch prediction 을 이용하여, 검사를 피해 out-of-bounds 접근 가능함.

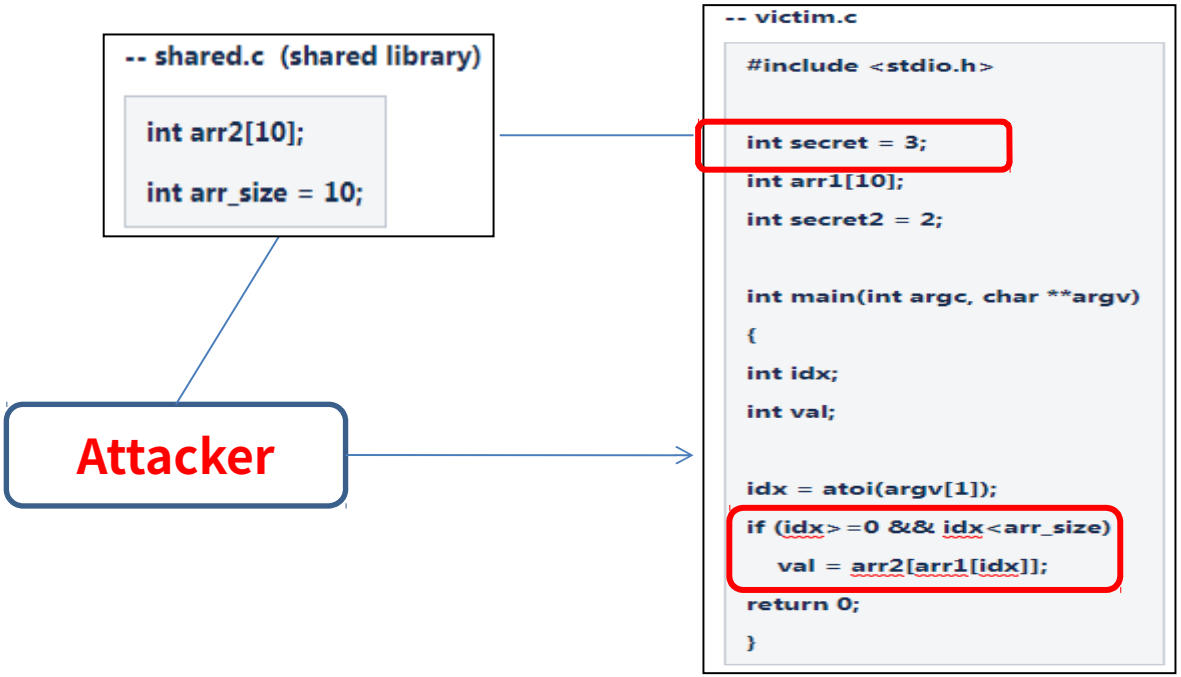
(2) CPU 에서 예측 잘못됨을 인지 후, 실행 취소하더라도, 실행 중에 접근한 memory 가 cache 에 남음.
즉, cache 상태는 취소되지 않음.



- 만약 if 문이 너무 빨리 실행된다면, 공격은 실패한다!!

왜??? If 문이 빨리 실행된다는 것은, true-false 판단이 빨리 끝나는 것. 이 판단이 끝나면, CPU 는 예측 실패 인지하고 실행 취소 가능. 즉, 실행 취소가 매우 빨리 되므로, 공격 코드가 수행되기도 전에 실행 취소되는 것!

- 만약 위 코드와 같이 if 문 내부에, 별도의 메모리 접근 (arr_size) 이 있다고 하면,
  arr_size 를 반드시 cache 에서 없애서,, if 문이 느리게 실행되도록 해야만 한다! 그래야 예측 실행 가능.



(1) idx == 0 을 넣어서 100번 실행.

(2) arr_size 캐쉬에서 없앰.

☒ 공격자가 접근 가능한 메모리는 flush 명령어를 통해 cache 에서 없앨 수 있음. (CPU specific)

(3) idx == -1 넣고 실행. Branch prediction 에 의해 if 문 실행됨.

(4) arr2[arr1[-1]] □ arr2[secret] □ arr2[3] 접근. arr2[3] 메모리가 cache 에 남음.

(5) CPU 는 예측실행 실패한 것을 인지하고, 예측실행한 명령어들 취소. (cache 는 취소 안됨)

(6) 공격자는 arr2 모든 메모리 접근. arr2[3] 만 cache 에 있으므로 빨리 접근됨. 따라서 secret == 3 을 추론

.

- Spectre v1 공격에 대한 전제 조건 (가장 큰)

  (1) arr2[arr1[idx]]; □ 이러한 형태의 코드 패턴이 있어야 함.

  (2) 위 코드 패턴에서 “idx” 를 공격자가 입력할 수 있어야 함.

- Linux kernel 에 위 두 조건 만족하는 코드 존재??

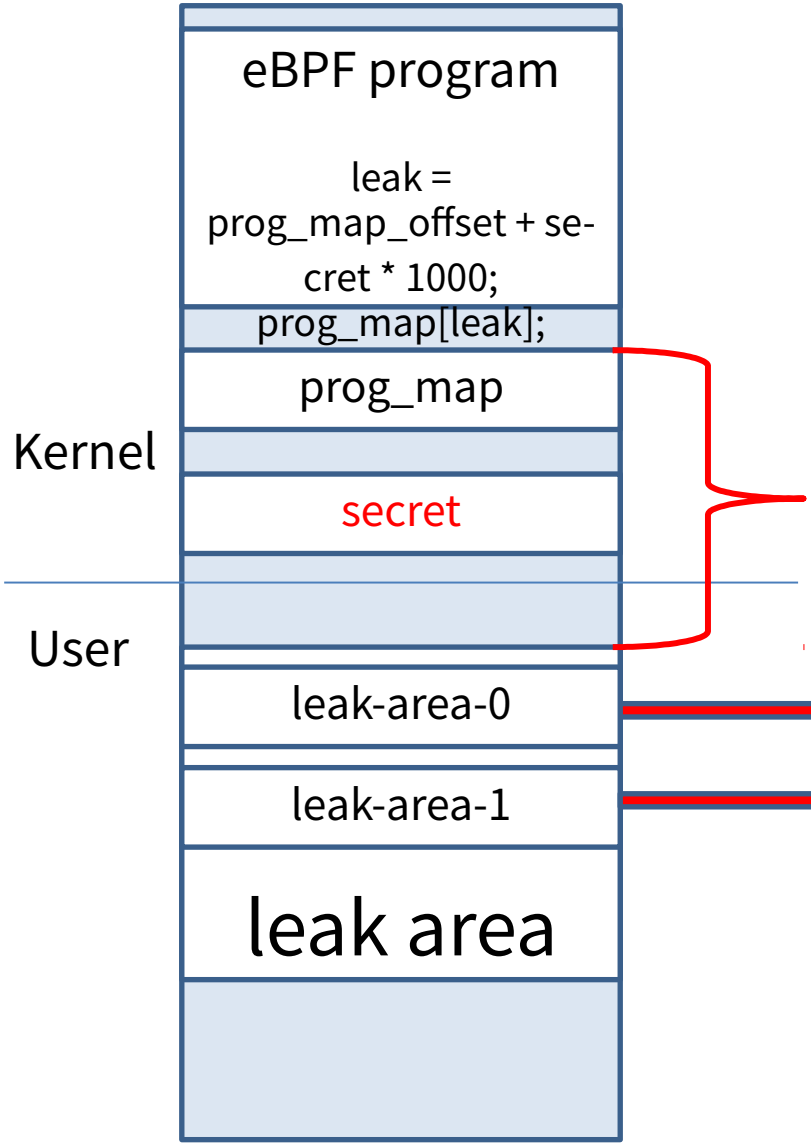
  - 없음. 그럼 어떻게 Spectre v1 을 실제 공격에 활용하나??

  - 취약한 코드 패턴을 직접 만들어서 실행!
    Spectre v1 에 취약한 코드 패턴을 가진 eBPF 프로그램 만들어서,
    Linux kernel 에서 이를 실행!!
    Linux kernel memory 를 알아내는데 활용!!

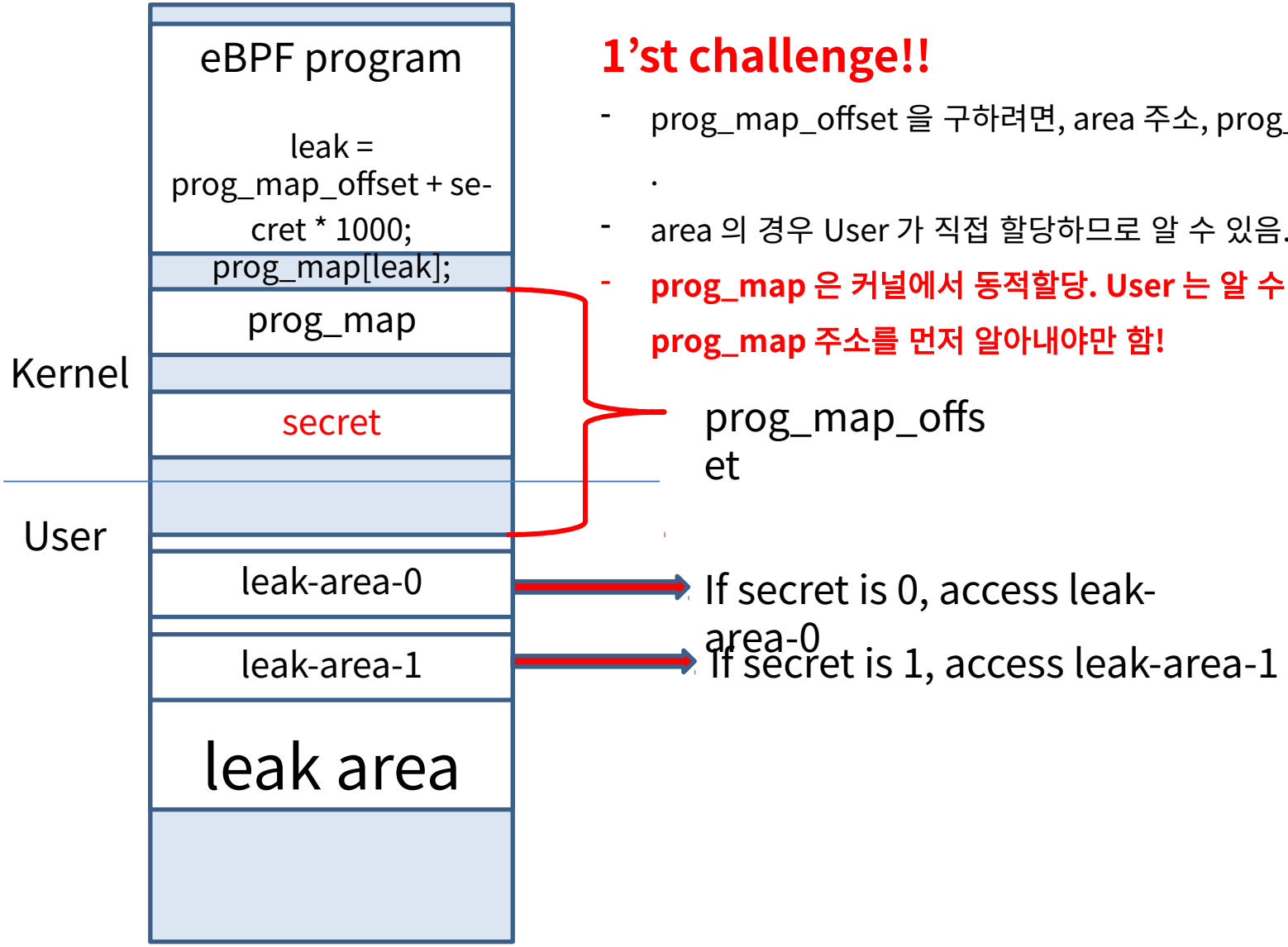
Exploit eBPF with Spectre variant 1

---- Read Linux kernel memory from unprivileged user

---- Explore exploit code from Google project zero



- 공격자는 secret 을 알아내는 것이 목표.
- prog_map array out-of-bound access 를 spectre v1 을 통해 수행.
- secret 이 0이면 area-0 을 access, secret 이 1 이면 area-1 을 access. 공격자는 area-0, 1 중 어딜 접근했는지를 보고 secret 값을 추론.

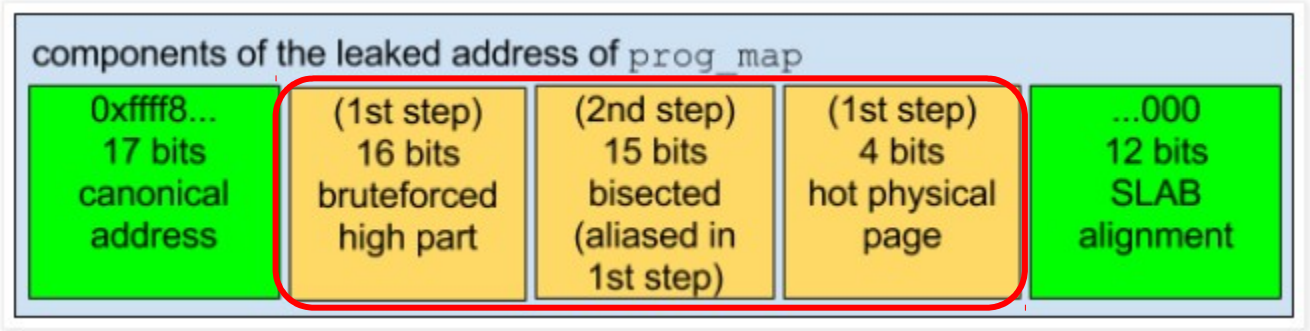


1’st challenge!!

- prog_map_offset 을 구하려면, area 주소, prog_map 주소 알아야 함 .
- area 의 경우 User 가 직접 할당하므로 알 수 있음.
- **prog_map 은 커널에서 동적할당. User 는 알 수 없음!!**

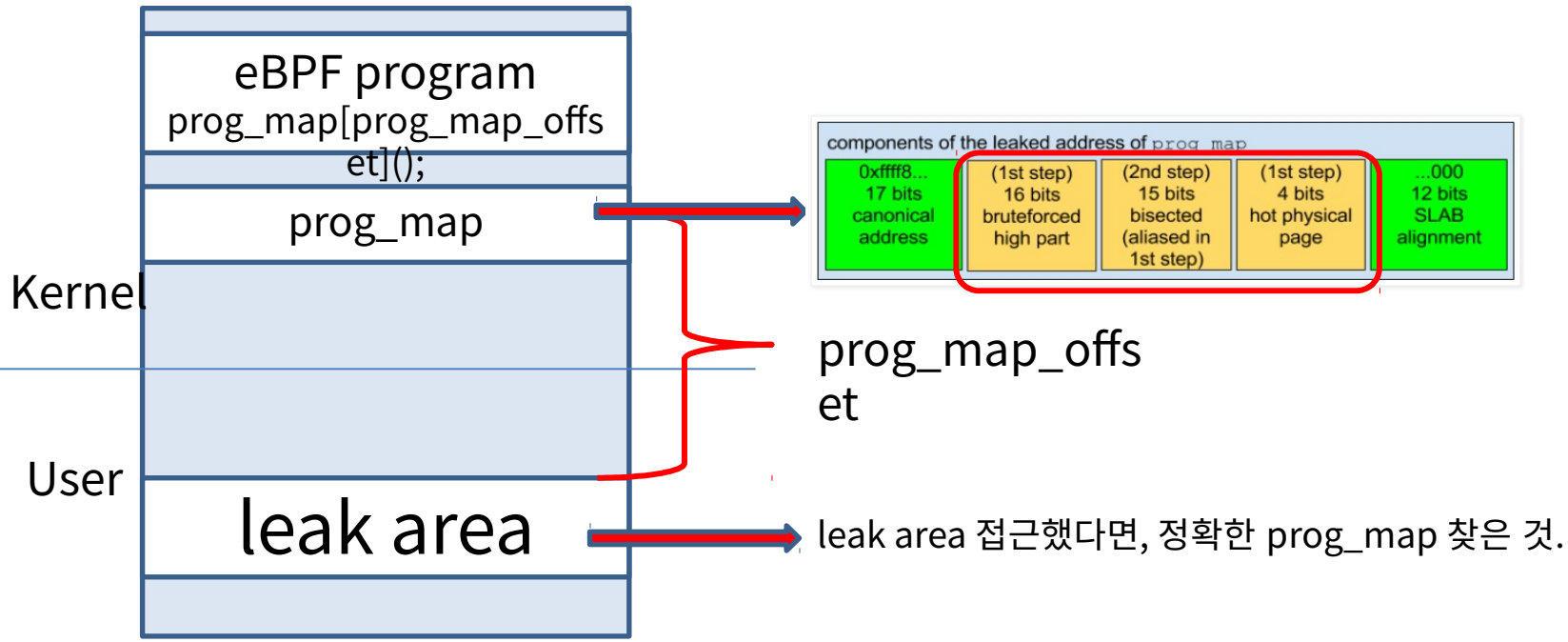
  prog_map 주소를 먼저 알아내야만 함!

Components of address of prog_map



- prog_map 의 주소를 추론해야 함. prog_map 은 64-bit 커널 주소.
- 최상위 17bit 는 Linux kernel 주소 체계에 의해 고정됨.
- 최하위 12bit 는 prog_map 이 page-aligned 되므로, 역시 0으로 고정됨.
- 나머지 35bit 를 알아내면, 정확한 prog_map 주소를 알 수 있음.

What is the simplest way to predict?



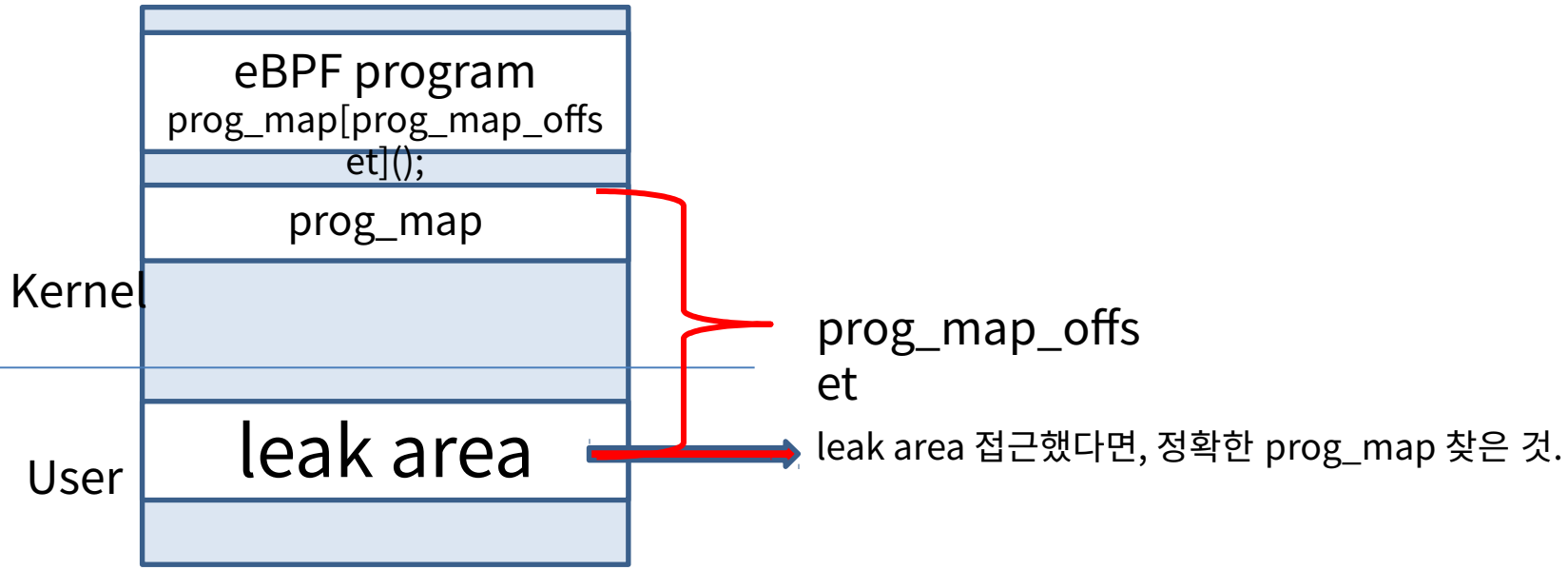
- 35-bit brute force 수행. 모든 후보 prog_map 주소에 대해 아래 과정을 반복.

(1) 후보 prog_map 주소, area 주소를 기반으로 prog_map_offset 계산.

(2) prog_map_offset 을 index 로 하여 eBPF program 실행. Spectre v1 활용해야 함. eBPF verifier 에 의해 검사되므로, 예측실행 활용해야만 함.

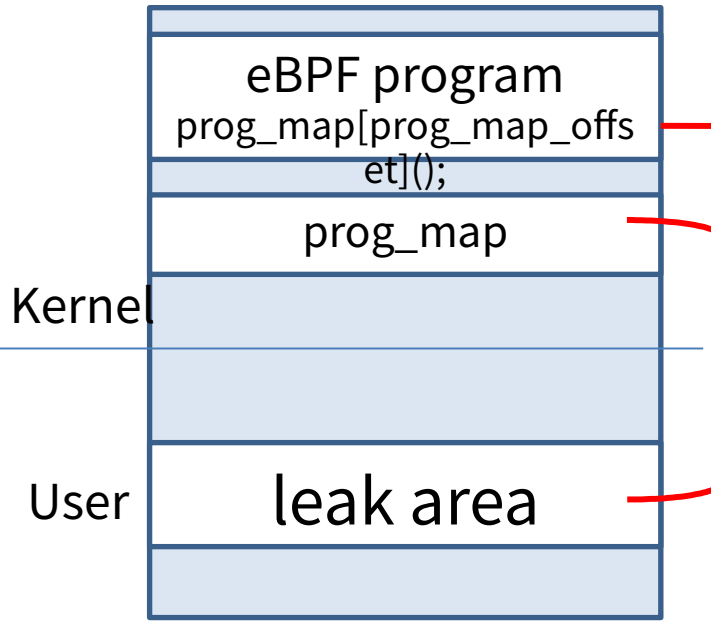
(3) area 접근 확인. 접근했다면, 정확한 prog_map 찾은 것!

- Problems of the simplest way



- 35-bit bruteforce. 느림. 좀 더 빠르게 prog_map 주소를 찾는 방법은?? (Appendix-1 참고)

prog_map 주소 찾는 eBPF program 분석



Runtime 에 실제 실행되는 코드

```
if (prog_map_offset>=0
&&prog_map_offset<prog_map.size)
    prog_map[prog_map_offset]();
```

prog_map_offs
et

- **prog_map 주소에 대한 추론 순서**

(1) 후보 prog_map 주소 선정. prog_map_offset 선정.

(2) leak area 에 대해 cache flush (cache 에서 없앰)

User memory 이므로 cache flush 명령어 활용하여 없앨 수 있음.

(3) prog_map.size 를 cache 에서 없앰. (if 문 느리게 실행되기 위해)

 kernel memory 이므로 User 에서 cache flush 불가능!!

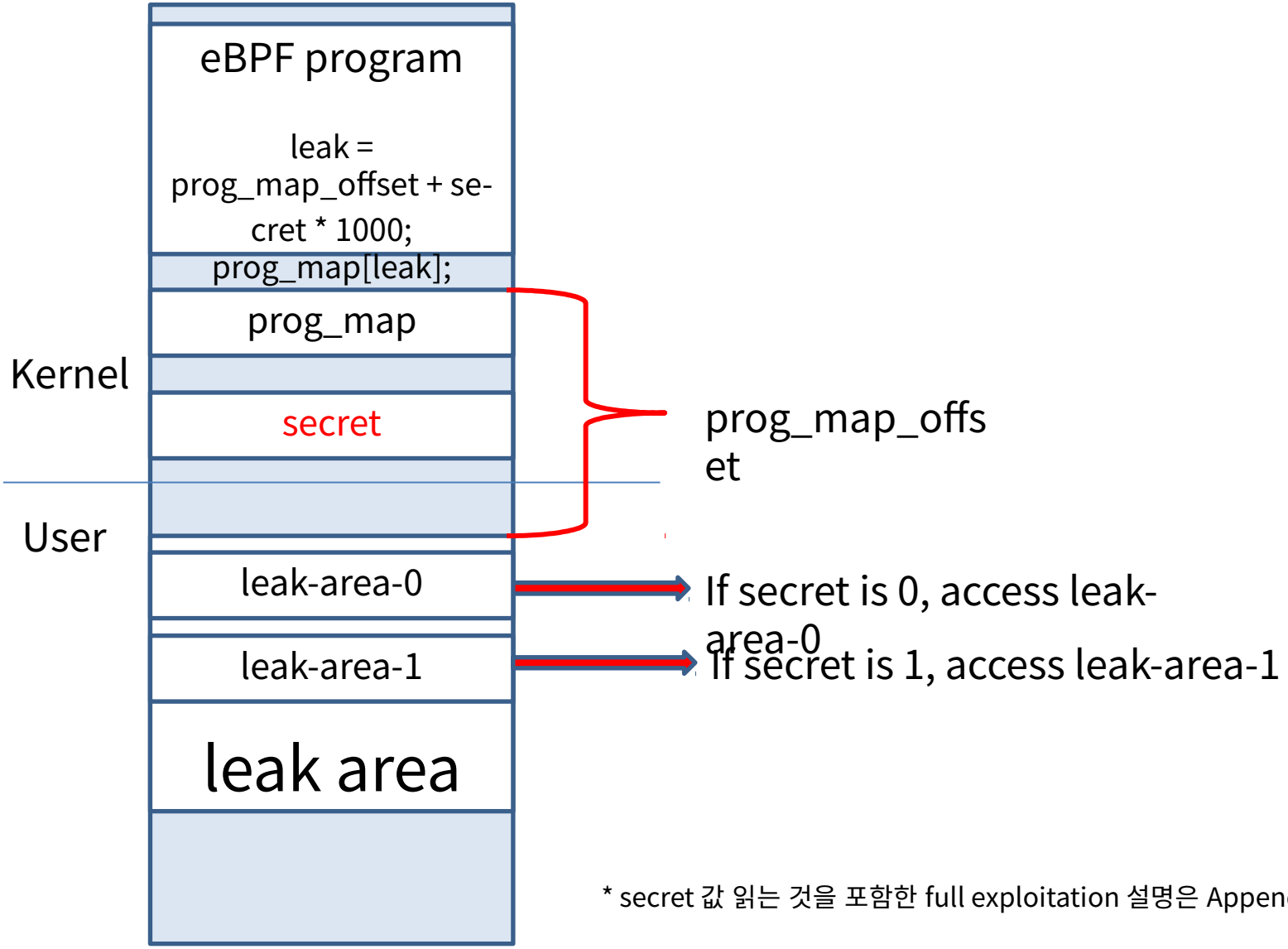
(4) eBPF 프로그램 실행.

(5) leak area 접근 여부 판단. 접근했으면 prog_map 찾은 것.

New Challenge!!
Cache line bouncing 활용!!
(Appendix-2 참고)

드디어 정확한 prog_map 주소 알아냄…

다음 step 은, 이를 활용하여 Linux kernel memory leak 수행하는 것.



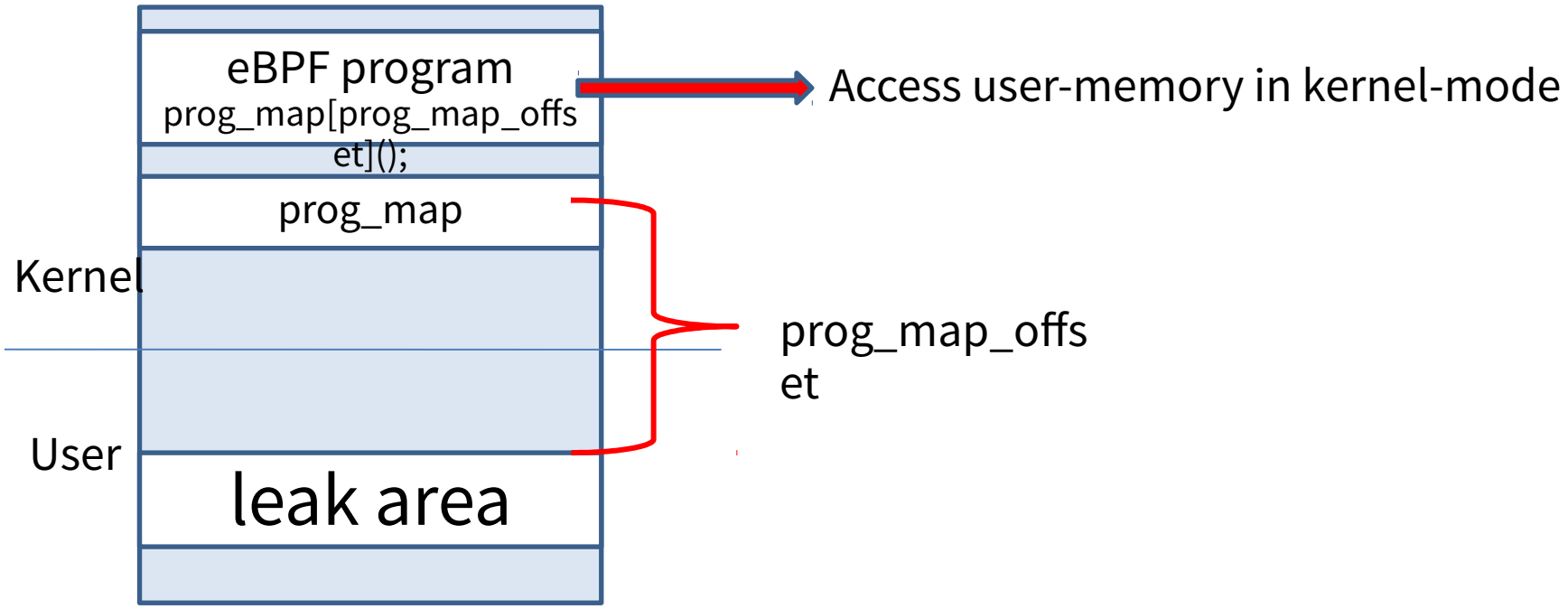
* secret 값 읽는 것을 포함한 full exploitation 설명은 Appendix-3 참고.

Demo Time!!

Exploit eBPF with Spectre variant 1

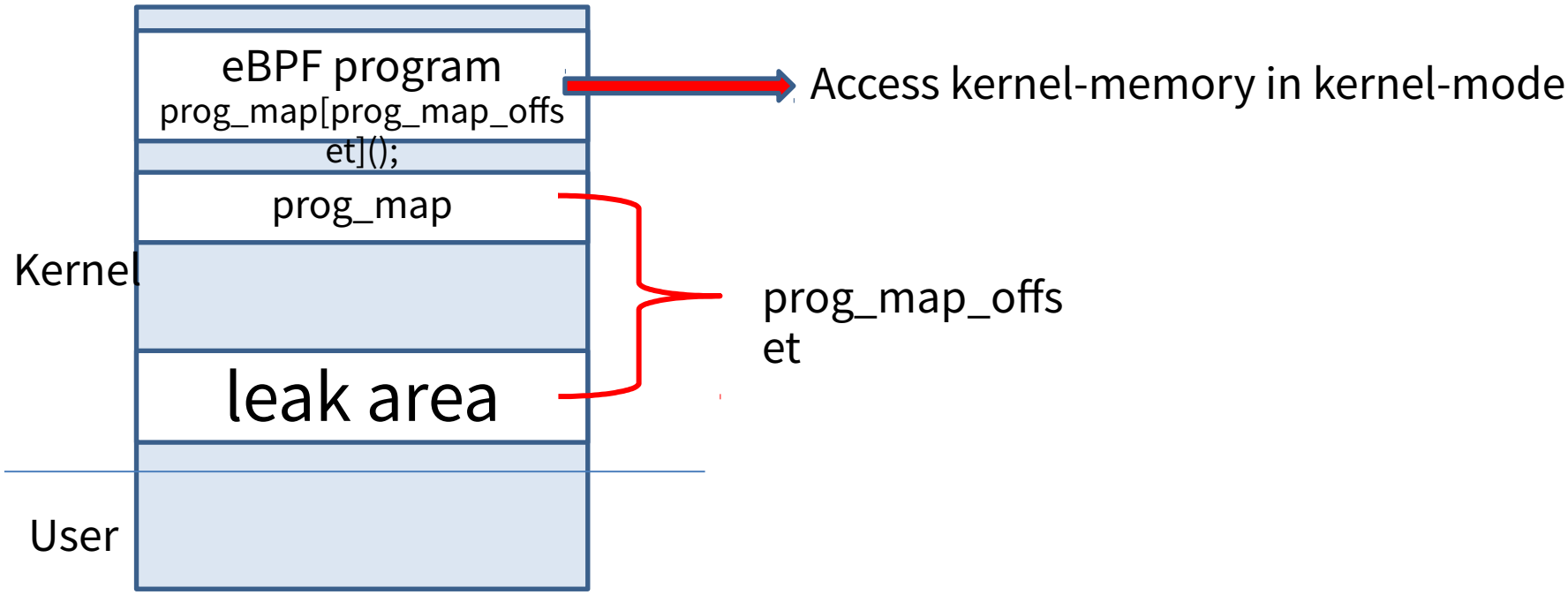
---- Limitations of exploit code from Google project zero

---- Update the exploit code to bypass SMAP

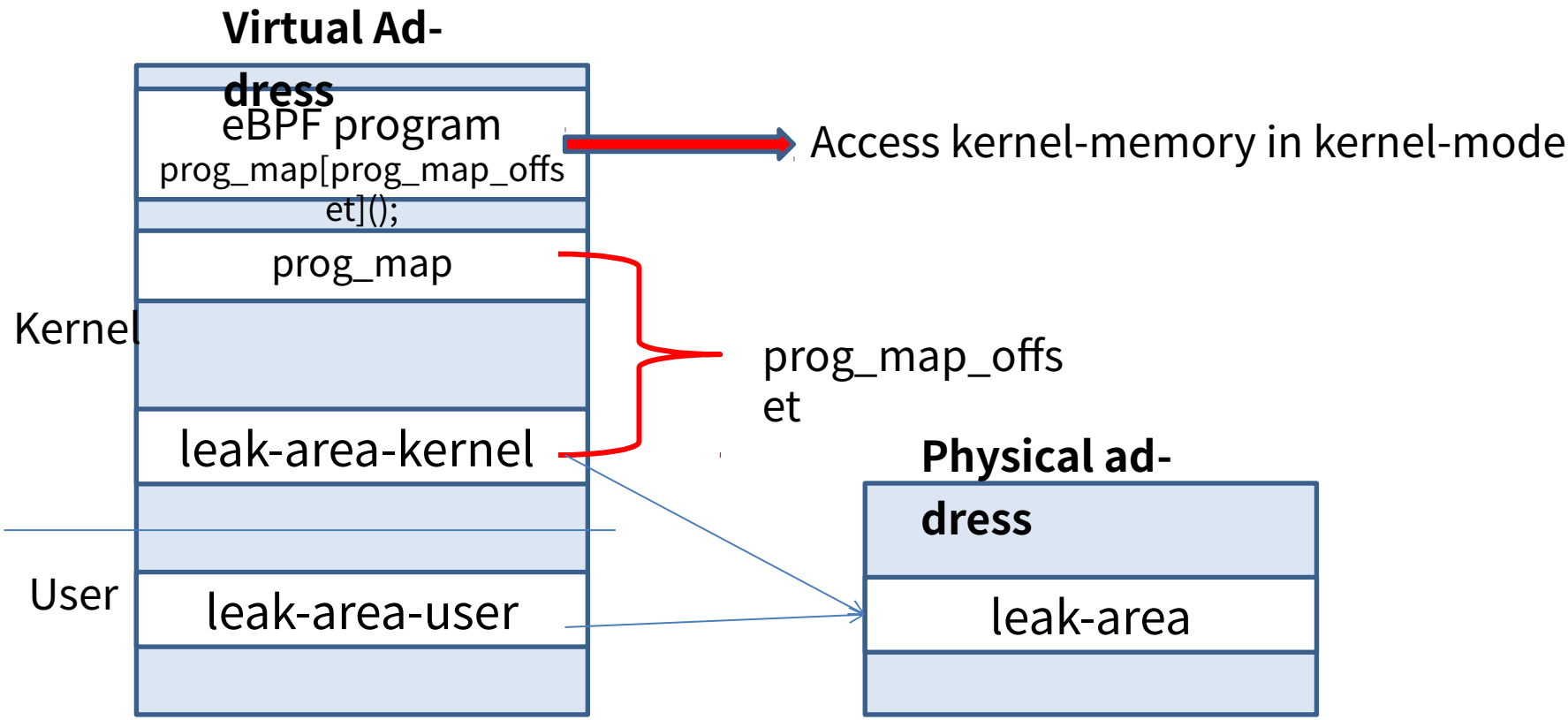


- 위와 같은 접근은 Kernel mode 에서 User memory 를 접근하는 것. 최신 CPU 에서는 Kernel mode 에서 User memory 접근 막는 기능이 포함되어 있음.
- Intel 은 SMAP, ARM 은 PAN 이라고 부름.
- 따라서, SMAP enable 되어 있는 시스템에서 이 exploit 은 동작하지 않음. 최근 대부분의 시스템은 모두 SMAP 이 enable 되어 있음.

Demo Time!!

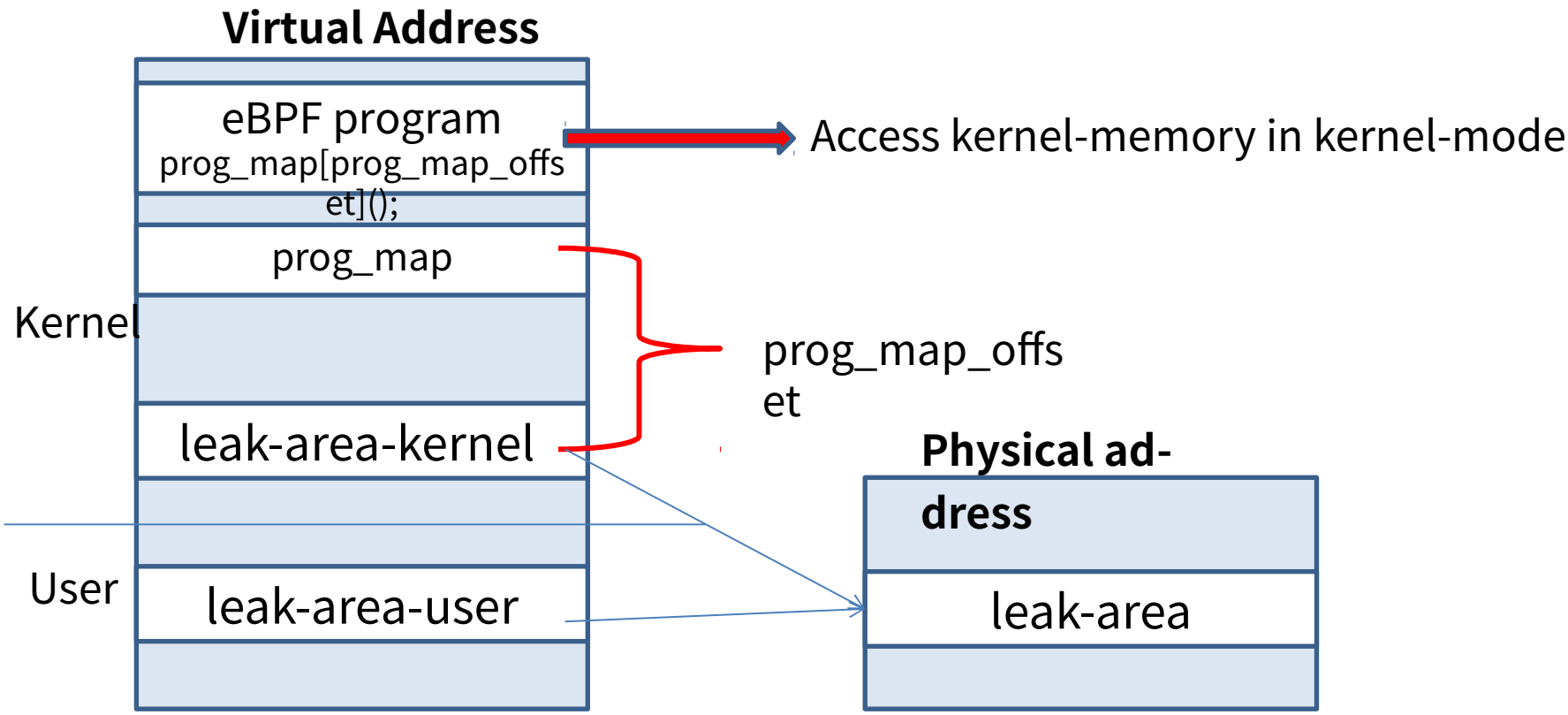


- leak area 를 kernel 로 옮기면 됨.
- 하지만 추가적인 문제 발생함. leak area 는 커널 메모리이므로, User 는 leak area 를 접근할 수 없음.
- 어떻게 User 가 커널 메모리인 leak area 를 접근할 수 있을까??



- User page 가 kernel 가상주소에도 똑같이 잡히는 사실을 악용.
  (ret2dir, 2014, USENIX Security)

- 즉, leak-area-kernel, leak-area-user 는 같은 물리주소를 가리키므로, leak-area-kernel 이 접근되어서 cache 에 올라가면,, leak-area-user 접근 시 더 빨리 읽혀진다. 같은 물리주소이므로!!



1. leak-area-user 에 대해 cache flush.
2. eBPF 프로그램 실행. Secret 값에 따라 leak-area-kernel 을 접근.
3. leak-area-user 를 접근. 접근되는 속도에 따라 secret 값을 추론.

Demo Time!!

Spectre variant 4 (CVE-2018-3639)

---- speculative store bypass

---- explore a vulnerable sample code

Spectre variant 4 와 연관된 CPU 최적화 기능

- **Memory disambiguation (== Speculative store bypass)**

  - 성능향상을 위해, store 를 실행하지 않은 상태에서, 다음 load 명령을 먼저 예측 실행하는 것.
  - 이를 악용하여 뭘 할 수 있나??

    ☒ 특정 메모리에 이전에 저장된 값을 읽을 수 있다.

Spectre variant 4 에 취약한 코드 패턴

```
(1) ptr = *stack;
(2)*ptr = idx;
(...)
(3)idx2 = *ptr2;
(4)val = arr[idx2];
```

- In-order execution flow (1 → 2 → 3 → 4)

  (1) ptr = *stack; → stack 에 있는 값을 읽음.
  (2) *ptr = idx; → (1) 이 수행이 완료되어야만, ptr 주소 확정. 그 후 store 수행 가능.
  (3) idx2 = *ptr2; → (2) 의 store 명령이 수행될 때까지 기다림.
  → ptr 주소 확정되고, (2) 실행됨. 실행완료 후, load 명령 수행.
  (4) val = arr[idx2]; → (3) 의 결과값으로 arr 접근.

- 아무런 문제 없음.

Spectre variant 4 에 취약한 코드 패턴 – 최적화 버전

```
(1) ptr = *stack;
(2)*ptr = idx;
(...)
(3)idx2 = *ptr2;
(4)val = arr[idx2];
```

- Optimized execution flow (1 → 3 → 4 → 2)

  (1) ptr = *stack; → stack 에 있는 값을 읽음.
  (2) *ptr = idx; → (1) 이 수행이 완료되어야만, ptr 주소 확정. → Speculative store bypass!!
  (3) idx2 = *ptr2; → ptr != ptr2 라면 굳이 (2) 수행완료를 기다릴 필요 없음.
  → (2) 완료 기다리지 않고, 로드 명령어 바로 수행.
  (4) val = arr[idx2]; → (3) 의 결과값으로 arr 접근.

- 위와 같이 예측 실행했는데, ptr == ptr2 였다면?? 예측 실패한 것. CPU 는 예측 실패 인지하고, 실행 취소.

Spectre variant 4 에 취약한 코드 패턴 – 보안 상 문제되는 시나리오

```
(0)  *ptr = secret;

(1)  ptr = *stack;
(2)  *ptr = idx;
(3)  idx2 = *ptr2;
(4)  val = arr[idx2];
```

Code

```
(0) *secret_addr = secret;

(1) secret_addr = *stack;
(2) *secret_addr = idx;
(3) idx2 = *secret_addr; (idx2 = idx;)
(4) val = arr[idx];
```

In-order exe-cution flow

secret 사용 안됨.

```
(0) *secret_addr = secret;

(1) secret_addr = *stack;
(3) idx2 = *secret_addr; (idx2 = secret;)
(4) val = arr[secret];
(2) *secret_addr = idx;
```

Vulnerable execution flow

- secret_addr 에 있던 이전 값 secret 을 읽음
.
- secret 에 따라 다른 메모리 접근 발생!!

- ptr 이 secret address 라고 가정.
  secret address 에서는 secret 값이 저장되어 있는 상태.
  ptr == ptr2 인 경우.

- 즉, secret_addr 에 저장되어 있던 이전 값 secret 을 읽을 수 있다!!

Spectre variant 4 exploit 을 위한 추가 요구사항

```
(1) ptr = *stack;
(2)*ptr = idx;
(...)
(3)idx2 = *ptr2;
(4)val = arr[idx2];
```

- 위 코드에서, (1), (2) 가 충분히 빨리 실행된다면?? 예측 실행에 의해 (3), (4) 가 실행되지 않음.
- 따라서, stack 이 변수에 대한 메모리를 cache 에서 미리 제거해 줘야 함. 이를 통해 (1), (2) 를 느리게 동작하도록 만들어야만 (3), (4) 가 예측 실행 될 수 있음.

Exploit eBPF with Spectre variant 4

---- Read Linux kernel memory from unprivileged user

Exploit code from Google

- https://www.exploit-db.com/exploits/44695/
- Root 권한, kernel code 수정 필요. real world exploit 이라고 보기 어려움.
- 따라서 Google 의 exploit code 전략을 따르지 않고, 다른 전략의 exploit code 작성.

  (Variant4 Gadget 의 기본적인 형태는 Google exploit code 를 참고)
- 최종 목표는, User 권한에서 추가 가정 없이 Kernel memory read 하는 것.

Review

(1) Spectre variant 4 를 이용하여, 특정 메모리에 이전에 저장되었던 값을 읽을 수 있음.

(2) eBPF Program 에서 접근할 수 있는 Memory 는 제한되어 있음.

- 일반 Kernel memory 접근 불가능.
- eBPF Program 에서 사용하는 Kernel stack memory 는 접근 가능.
- eBPF Program 에서 Kernel stack memory 접근하기 위해서는 무조건 한번 write 를 해야만 함. (Uninitialized stack 공격을 막기 위해)

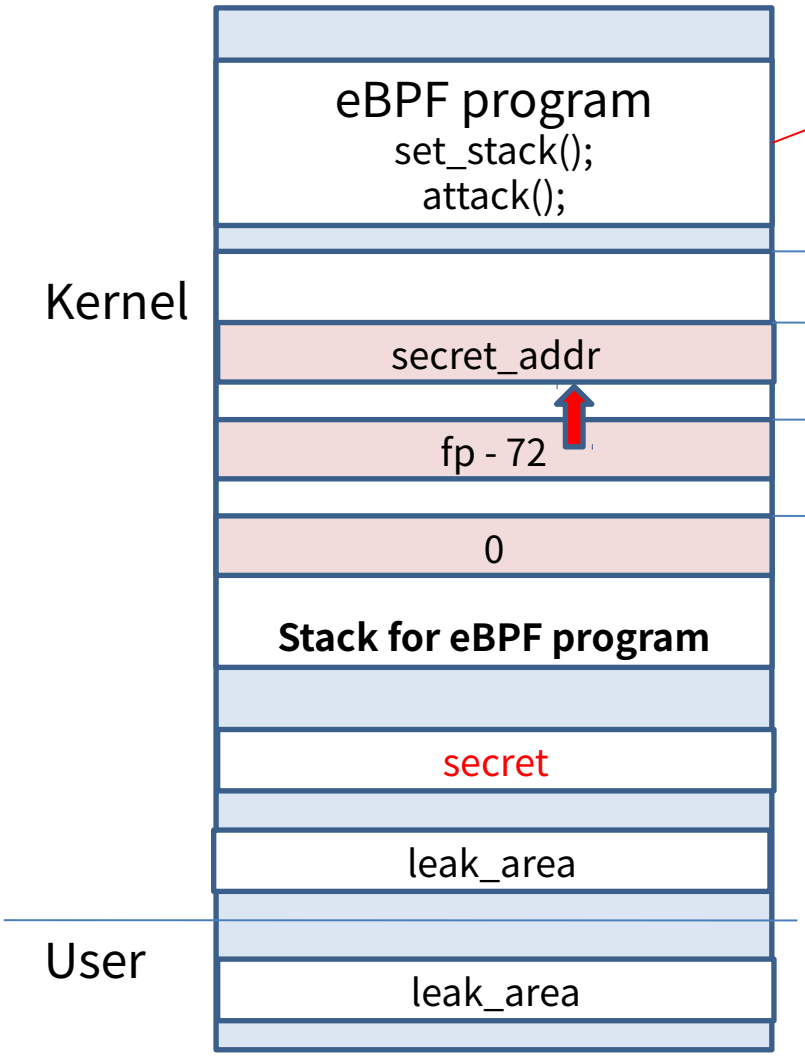
High level strategy

- Spectre variant 4 (특정 메모리에 이전에 저장된 값 읽는 것 가능) 활용하여, eBPF Program 의 제약 중 하나인, “일반 Kernel memory 접근 불가능” 을 우회하는 것이 목표.

- 공격자가 접근할 메모리를 secret address 라고 명칭. (일반 Kernel memory). 이 메모리는 곧바로 접근 불가능.

  - secret address 를 stack 에 씀. (stack 에 이전에 저장되었던 값이 되도록)
  - Variant4 공격 수행.
  - 원래 읽을 수 없던 secret address 를 load 가능. secret 값 추출 할 수 있음.
  - secret 값에 따라 다른 leak area 접근.

Memory layout



공격 프로그램은 크게 2단계로 구성됨.

(1) 공격을 위한 stack 준비
(2) 공격

fp

fp-72

fp-144

fp-216

secret_addr

: secret 의 주소.

leak_area

: secret 에 따라 다르게 접근할 data.

공격을 위한 eBPF program 슈도코드

```
/* Stack and data */
*(fp-72) = secret_addr;
*(fp-144) = fp-72;
*(fp-216) = 0;
```

- 공격 프로그램 실행 전 Stack 준비

```
(1) ptr = *(fp-144);
(2) *ptr = fp-216;
(3) ptr2 = *(fp-72);
(4) secret = *ptr2;
(5) if (secret == 0)
    exit;
(6) *leak_area;
```

- 공격 프로그램 슈도코드

```
(1) ptr = fp-72;
(2) *ptr = fp-216; ==> overwrite secret_addr!
(3) ptr2 = fp-216;
(4) secret = *ptr2; ==> secret = 0;
(5) if (secret == 0)
    exit;
(6) *leak_area;
```

In-order execution flow

```
(1) ptr = fp-72;
(3) ptr2 = secret_addr;
(4) secret = *secret_addr;
(5) if (secret == 0)
    exit;
(6) *leak_area;
```

Vulnerable execution flow

Exploit 성공하기 위해선 fp-144 메모리가 cache 에서 없어야 함

```
(1) ptr = *(fp-144);
(2) *ptr = fp-216;
(3) ptr2 = *(fp-72);
(4) secret = *ptr2;
(5) if (secret == 0)
    exit;
(6) *leak_area;
```

이 명령이 느리게 실행완료 되어야,
(3)~(6) 명령어가 예측실행 가능하다.

- "fp-144" 메모리를 cache 에서 제거해야만 함. 어떻게? (Flush kernel stack memory from user)
  - Kernel stack memory 이기 때문에 공격자는 cache flush 불가능.
  - Kernel stack memory 이기 때문에 공격자는 cache line bouncing 불가능.
  - Cache eviction 등 추가 시도해봤지만 불가능했음.
- Google exploit code 에서는 이를 위해 Kernel code 수정. Kernel 에 fp-144 를 flush 하는 코드를 추가.

Code gadget 을 2가지 영역으로 나누어 문제 다시 분석

```
(1) ptr = *(fp-144);
(2) *ptr = fp-216;
(3) ptr2 = *(fp-72);
(4) secret = *ptr2;
(5) if (secret == 0)
    exit;
(6) *leak_area;
```

→ Code gadget-1

→ Code gadget-2

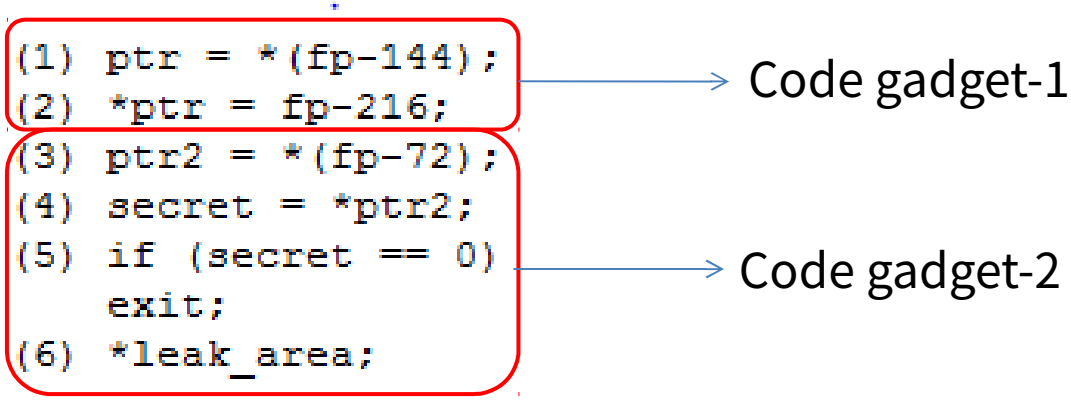
- **상황-1) Code gadget-1 이 Code gadget-2 보다 먼저 실행된다면??**

  □ fp-144 (kernel stack) 를 반드시 cache flush 해야 함.
  이를 통해 Code gadget-1 의 실행 속도 늦춰야 함.

- **상황-2) Code gadget-2 가 Code-gadget-1 보다 먼저 실행된다면??**

  □ fp-144 (kernel stack) 을 cache flush 해줄 필요 없음!!
  ☒ 이러한 상황을 만들어내고 exploit 해보자!!

- 어떤 상황에서 Code gadget-2 가 Code gadget-1 보다 먼저 실행될 수 있을까?



- **Spectre variant1 에 대한 security patch 방법은 2가지임.**

  1) Sanitize array index
  2) Add a barrier (lfence)

- 만약, “2) Add a barrier” 방법으로 patch 가 되어있는 리눅스 커널이라면,
  Code gadget-2 가 Code-gadget-1 보다 빨리 실행되는 경우가 존재함!! (드물게 발생)

  Why?? → Appendix-4 참고.

- Ubuntu kernel 4.4.0-[128~] 버전들에서 공격 가능함을 확인. Ubuntu security team 에 레포팅 한 상태.
  아직 패치되지는 않음. 패치 관련되서 확답 받은 후 공격코드 공개할 예정.

Demo Time!!

Find and Exploit
real Spectre gadgets in Linux kernel

Review

- Linux kernel 에 진짜 존재하는 Spectre gadgets 을 찾아보고, exploit 해보자!!
- Spectre gadgets 의 조건

  1) 취약한 코드 패턴 가짐 (Variant1, Variant4).
  2) User process 에서 1) 패턴에 malicious input 전달 가능.

How to easily find exploitable Spectre gadgets?

- 조건 2개에 대해서,

  1) 취약한 코드 패턴 가짐 (Variant1, Variant4). → Linux kernel 의 기존 취약점 유형과 연결해보자!!
  2) User process 에서 1) 패턴에 malicious input 전달 가능. → 기존에 CVE 가 할당된 코드!!

- 즉, Variant1, Variant4 와 유사한 SW 취약점을 가진 CVE 를 다시 방문해보는 것!!
  Security patch 된 CVE 에 대해서 다시 공격해보는 것!!

Revisit CVE to find Variant1 gadget

- 조건 2개에 대해서,

  1) Variant1 코드 패턴. → out-of-bound array access 취약점.
  2) User process 에서 1) 패턴에 malicious input 전달 가능. → out-of-bound array access 에 대한 기존 CVE

- 즉, out-of-bound array access 에 대한 기존 CVE 를 찾고, CVE 패치된 버전에 대해서 Variant1 공격을 수행.

CVE-2010-3437 is a great example

```
static struct pktcdvd_device *pkt_find_dev_from_minor(int dev_minor)
{
    if (dev_minor >= MAX_WRITERS)
        return NULL;
    return pkt_devs[dev_minor];
}
```

Not patched.

```
static struct pktcdvd_device *pkt_find_dev_from_minor(unsigned int dev_minor)
{
    if (dev_minor >= MAX_WRITERS)
        return NULL;
    return pkt_devs[dev_minor];
}
```

pkt_devs 주소 leak 가능!!
KASLR 우회 가능!!

Patched.
But, still exploitable by Variant1.

Revisit CVE to find Variant1 gadget

- 이와 같은 전략으로, 총 3개의 real gadget 찾았고, Linux kernel patch 까지 반영 완료함.

  - 시도해본 exploitable Variant1 gadget 수 : 3 개 (더 찾을 순 있음..)
  - exploit 성공 (KASLR 우회) : 2개
  - Linux kernel patch 반영 : 2개 (1개는 응답 없음)

Exploit the Variant1 gadget!!

- **See the code!!**

Revisit CVE to find Variant4 gadget

- 조건 2개에 대해서,

  1) Variant4 코드 패턴. → Uninitialized stack 취약점.
  2) User process 에서 1) 패턴에 malicious input 전달 가능. → Uninitialized stack 에 대한 기존 CVE

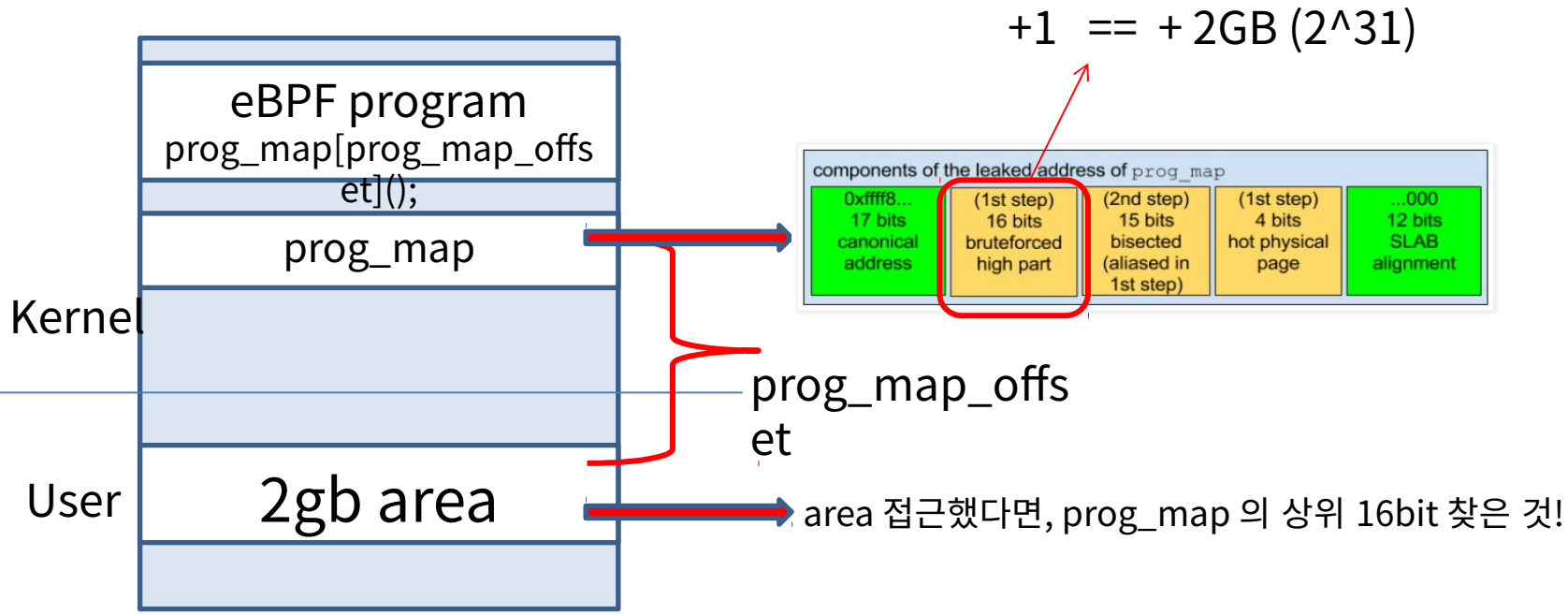
- 즉, Uninitialized stack 에 대한 기존 CVE 를 찾고, CVE 패치된 버전에 대해서 Variant4 공격을 수행.

Thank you

Appendix-1

-- How to optimize bruteforce on Variant1 exploitation?

Predict 16bit bruteforced high part



- 첫번째 16bit 를 brute-force 방식으로 추론.
- 하지만, 여기서도 문제가 있음.

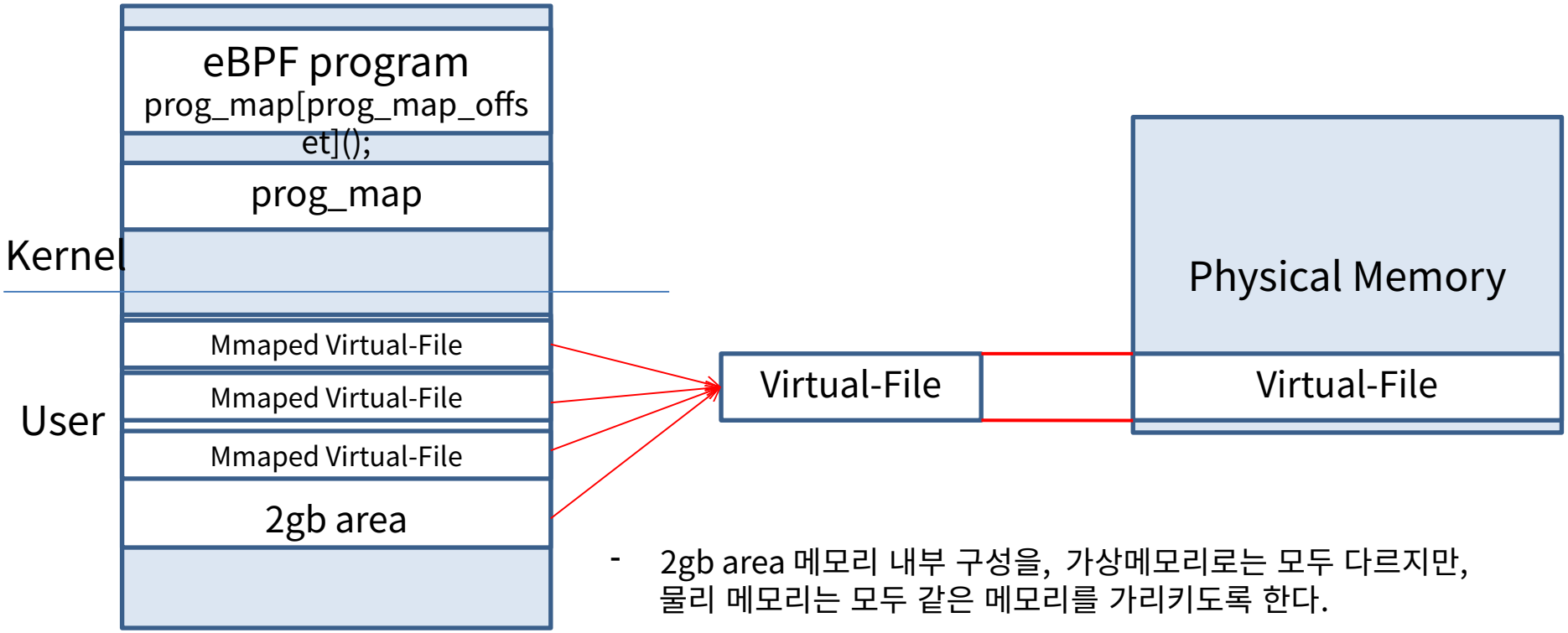
(1) 뒤쪽의 추론하지 않은 19bit 는 랜덤한 값임.

따라서 2gb area 를 access 했다 하더라도, 그 내부에 정확히 어디를 접근했는지 알 수 없음.

(2) 최악의 경우, 2gb 메모리를 모두 탐색해야만 2gb area 에 재대로 접근했음을 알 수 있음.

그러면 bruteforce space 는 2^16 이 아닌, 2^16 * (2^31 / 2^19) == 2^35. 결국 2^35 가 됨. 성능 향상 안됨.

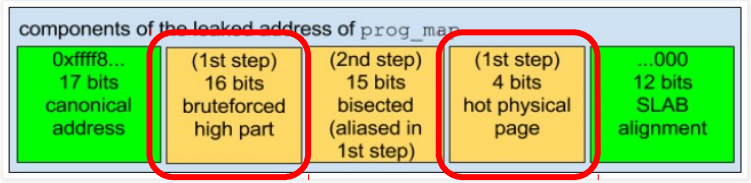
- Predict 16bit bruteforced high part with some trick



- 2gb area 메모리 내부 구성을, 가상메모리로는 모두 다르지만, 물리 메모리는 모두 같은 메모리를 가리키도록 한다.

- 그렇게 되면, 2gb area 중 가상메모리 어디를 access 하더라도, 같은 물리메모리이므로, 같은 cache 에 저장된다.
  따라서 공격자는 2gb area 중 몇 부분만 접근해봐도, access 여부 확인할 수 있다!!

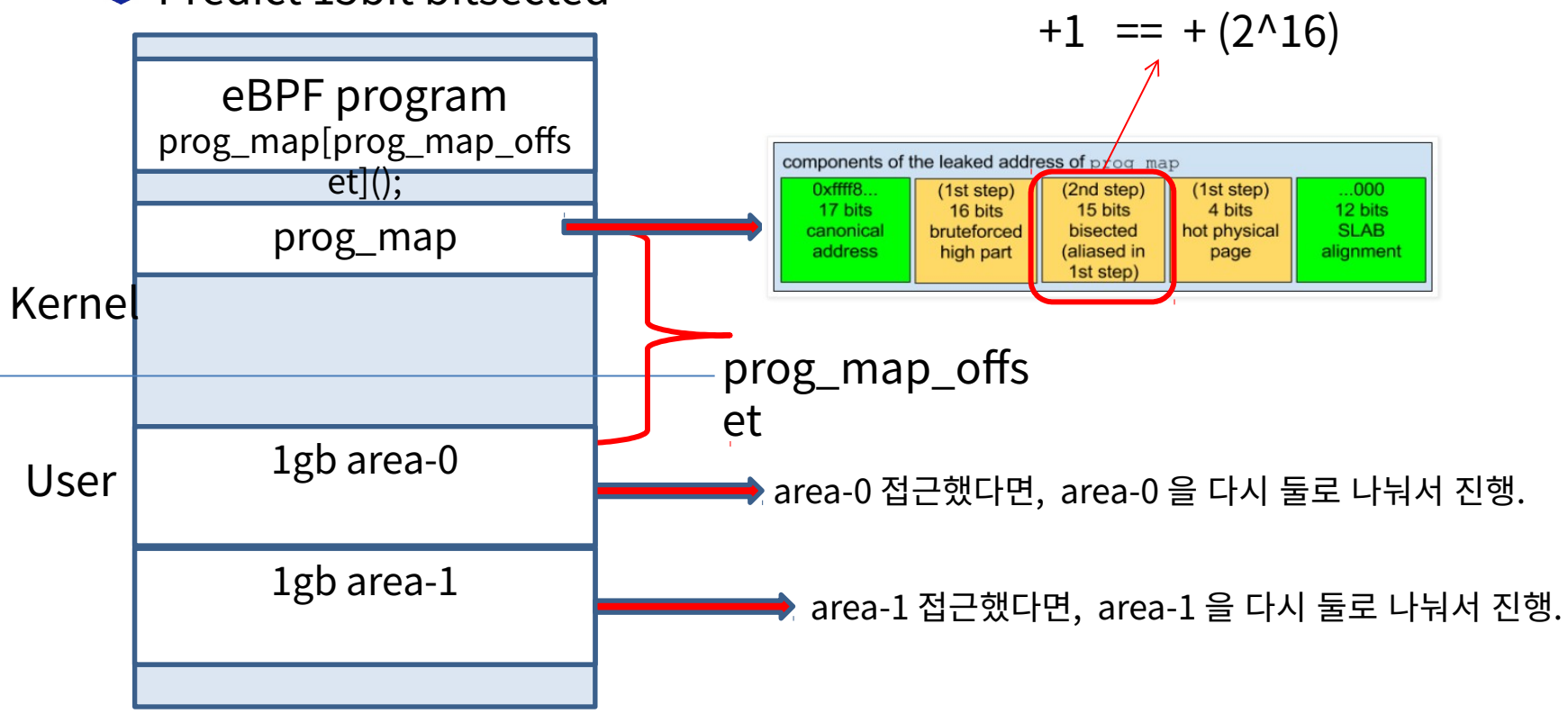
- Predict 4 bit hot physical page



Linux memory 관련 제약 때문에 가장 작은 size 는 2 ^ 16 Bytes.
즉, 2^4 pages == 16 개 page 로 구성.

4 bits hot physical page 주소 예측하려면, 이 16개 page 중 어디를 접근했는지 확인하면 됨.

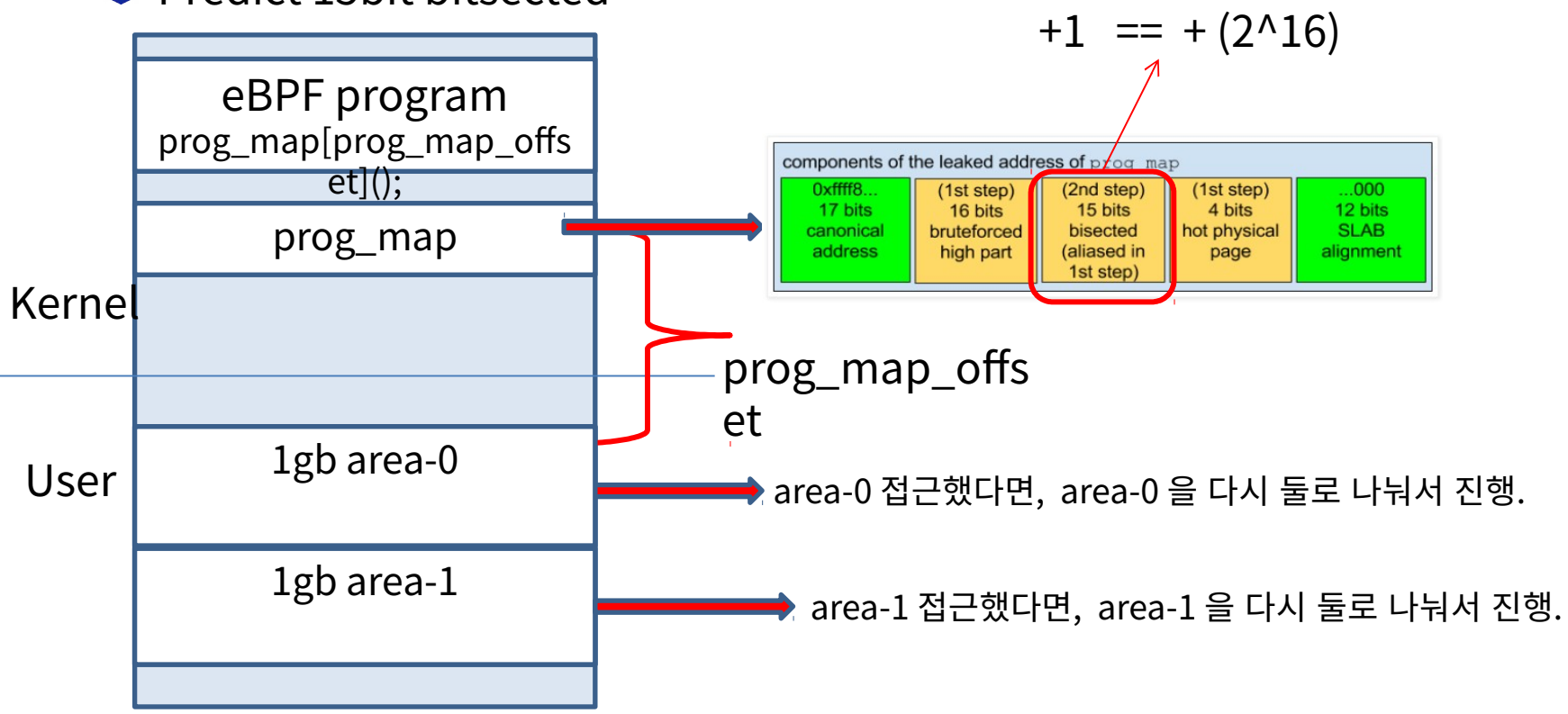
Predict 15bit bitsected



- quick search 알고리즘 형태로 접근.

(1) 추론할 15bits 에 대해 2가지 메모리로 나눔.
(2) 후보 prog_map 정해서 eBPF 프로그램 실행.
(3) area-0 에 access 했다면, area-0 을 다시 2가지 메모리로 나눔. 계속 진행.
(access 여부 판단 시, 물리메모리 trick 활용)

Predict 15bit bitsected



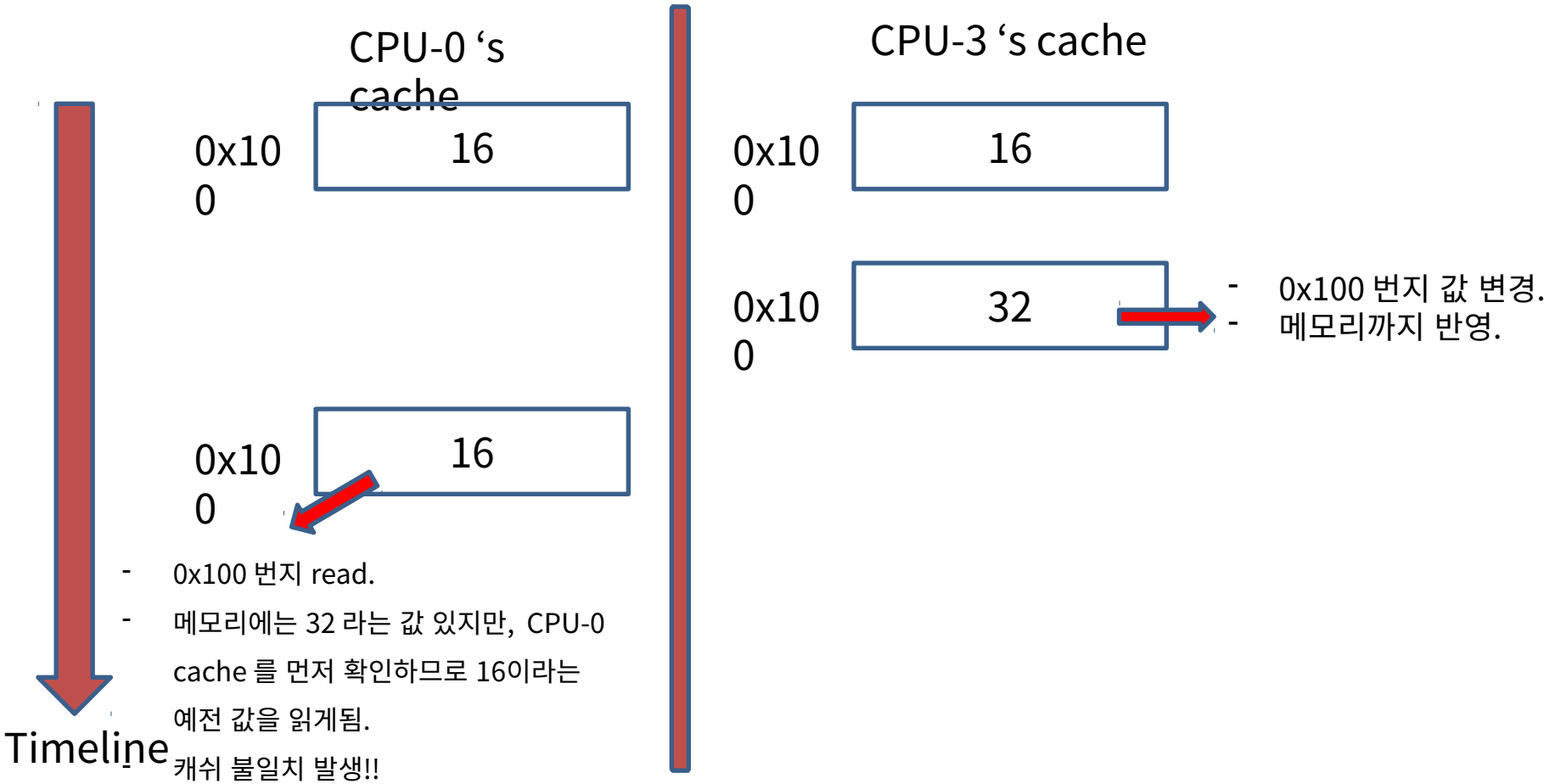
[Quiz] quick search 형태로 할거면,,
왜 처음에 전체 35bit 부터 quick search 알고리즘 적용해서 하지 않았을까???

- 가운데 15bit 를 둘로 나누기 위해선 1GB 영역 2개 필요. 1GB 는 Linux 에서 할당 가능한 크기.
- 상위 16bit 를 둘로 나누려면?? TB 수준의 영역 2개 필요. 하지만, Linux 에서 최대 할당 가능한 크기는 4GB. 즉, Linux memory 관리 제약 때문에!!

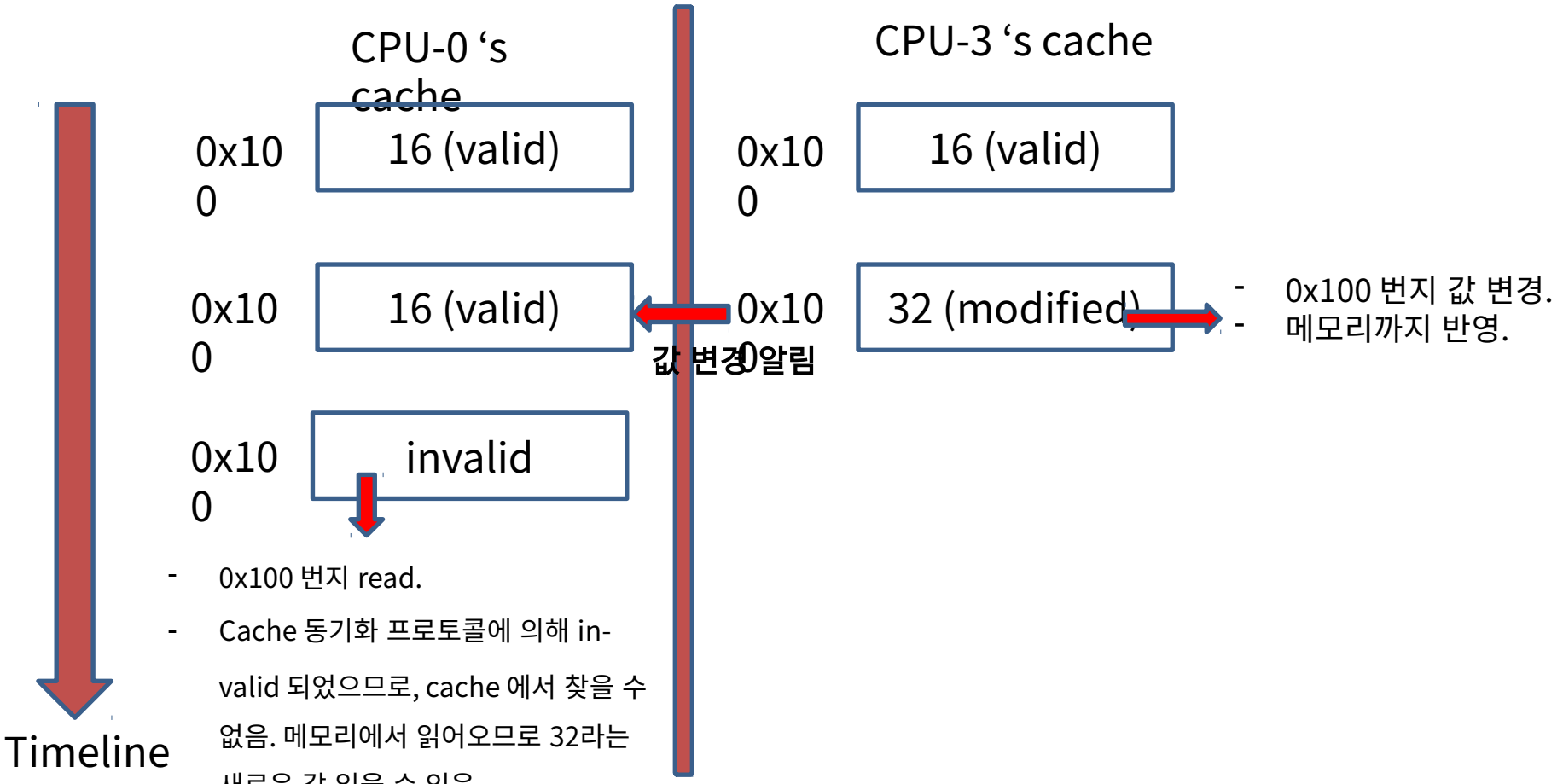
Appendix-2

-- Cache line bouncing to flush cache on kernel memory

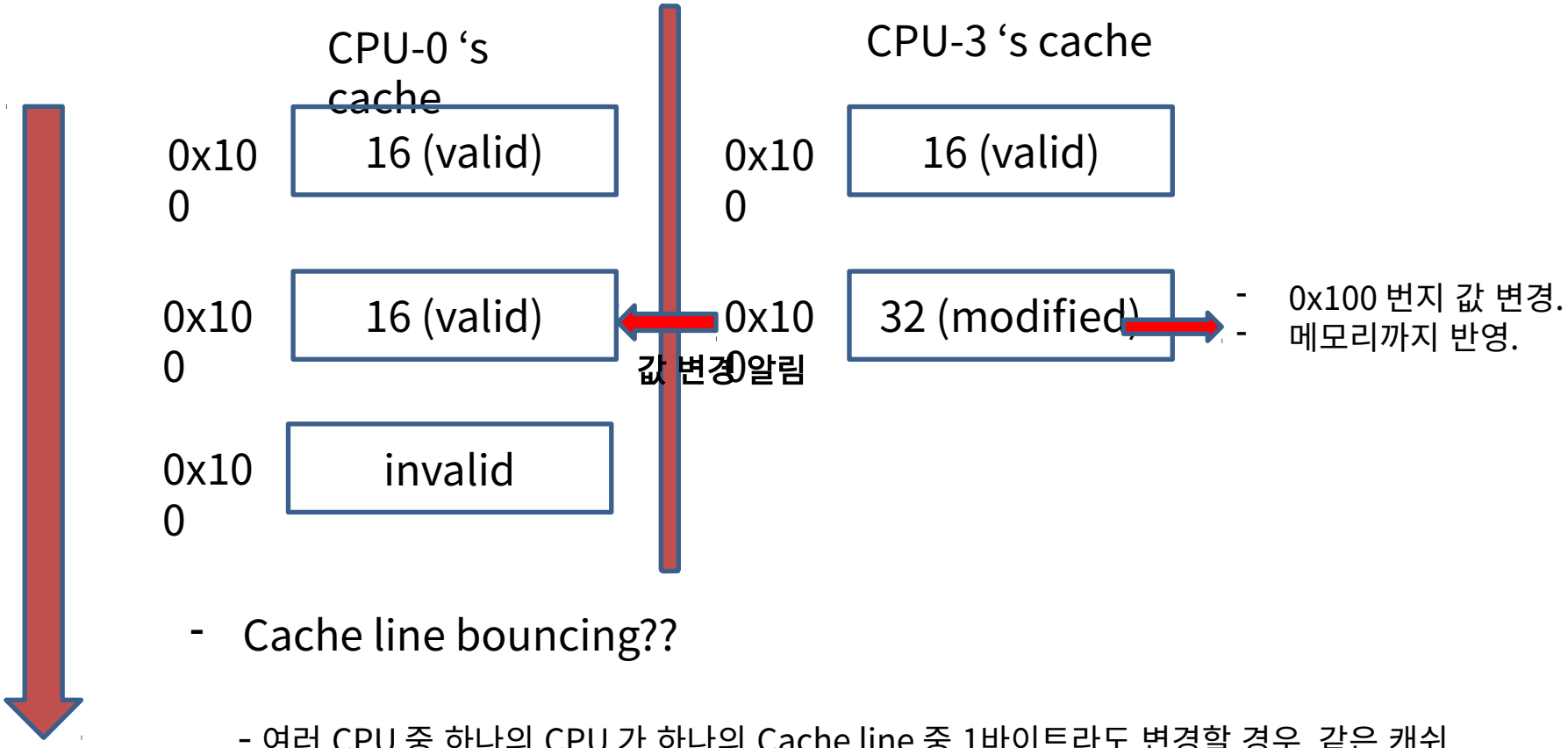
Cache incoherence



Cache coherence



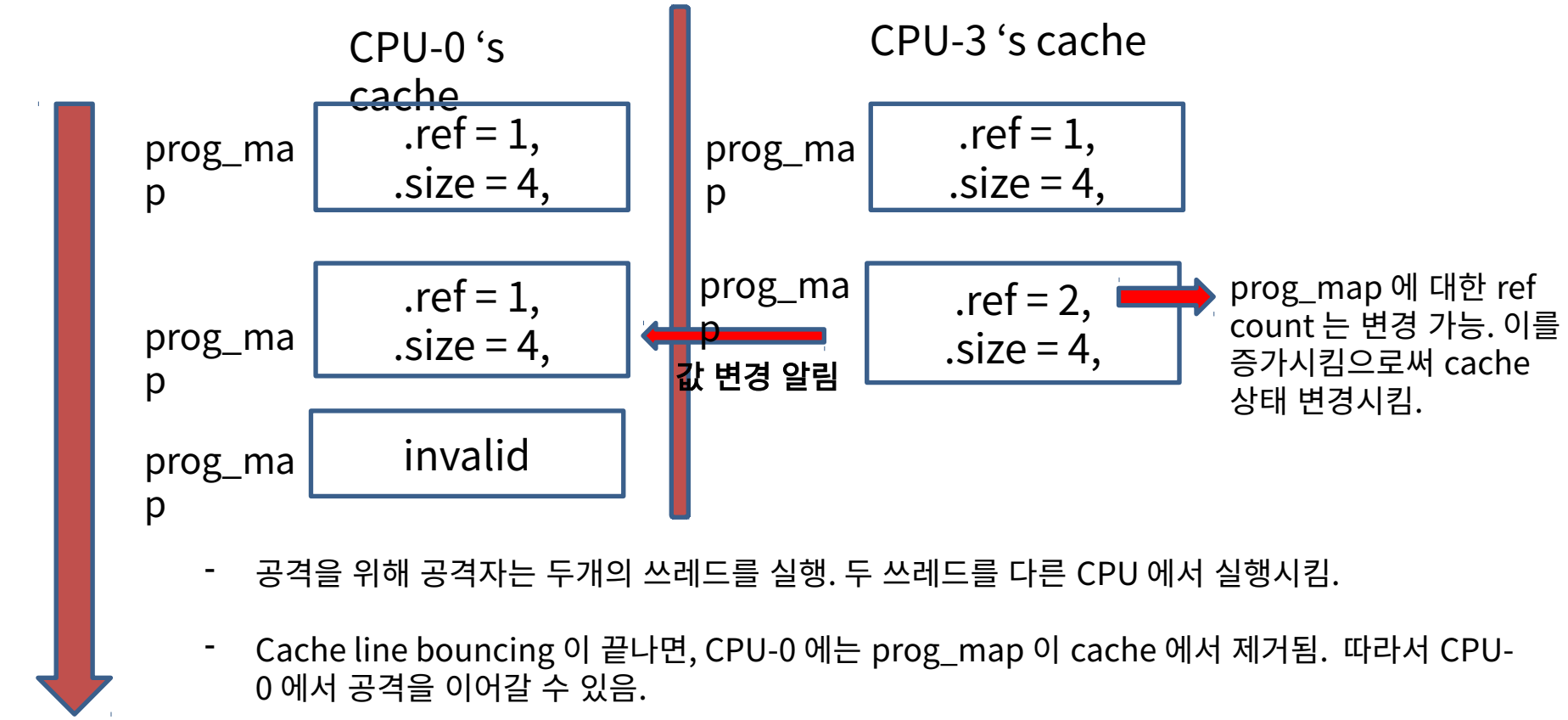
Cache line bouncing



- Cache line bouncing??

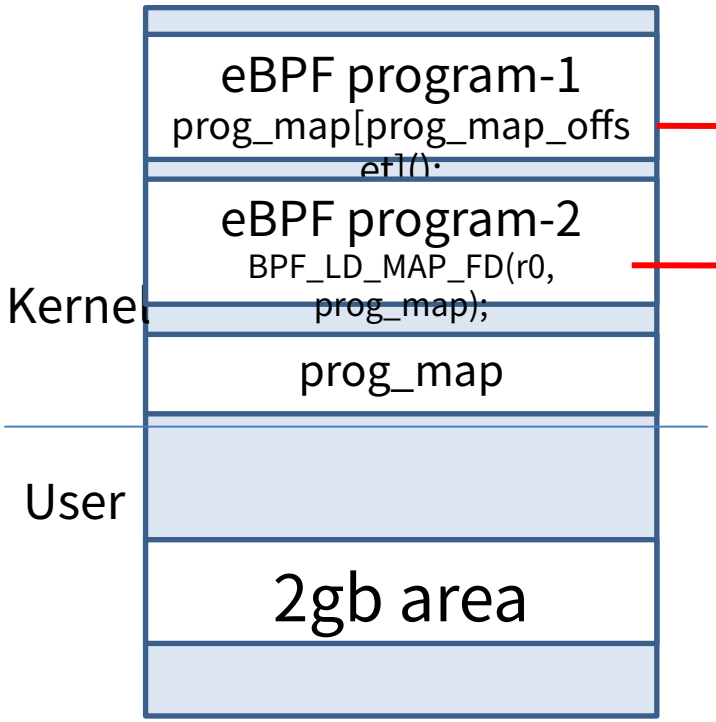
- 여러 CPU 중 하나의 CPU 가 하나의 Cache line 중 1바이트라도 변경할 경우, 같은 캐쉬 가지는 다른 CPU 에서 해당 cache 가 invalidate 되는 것. (캐쉬 동기화를 위해서)

- Exploit cache line bouncing for removing prog_map.size from cache



- 공격을 위해 공격자는 두개의 쓰레드를 실행. 두 쓰레드를 다른 CPU 에서 실행시킴.
- Cache line bouncing 이 끝나면, CPU-0 에는 prog_map 이 cache 에서 제거됨. 따라서 CPU-0 에서 공격을 이어갈 수 있음.
- Cache line bouncing 을 위해, prog_map 을 변경해야 하는데, size 는 변경 불가. refcount 수정 가능. refcount 수정하는 eBPF program 필요함.

Cache line bouncing 을 포함한 prog_map 주소 예측 과정



if (prog_map_offset>=0
&&prog_map_offset<prog_map.size)
prog_map[prog_map_offset]();

prog_map 의 refcount 증가시켜 주는 프로그램.

- **Full steps**

(1) 후보 prog_map 주소 선정. prog_map_offset 선정. (음수값)

(2) 2gb area 에 대해 cache flush.

(3) Cache line bouncing 을 이용하여 prog_map 을 캐쉬에서 제거.
이 때, eBPF program-2 활용하여 prog_map 값 변경. (CPU-3)

(4) eBPF program-1 프로그램 실행. (CPU-0)

(5) 2gb area 접근 여부 판단. □ 접근했으면 prog_map 찾은 것.

Appendix-3

-- Full exploitation on Variant1

Linux kernel memory leak 을 위한 eBPF program 슈도코드

```
data_map[] = {secret_offset, prog_map_offset, bitmask, bitshift};
if (secret_offset >= 0 && secret_offset < victim_map.size) {
    secret = victim_map[secret_offset];
    user_leak_offset = calc_prog_map_offset_with_secret(secret, prog_map_offset);

    if (user_leak_offset >= 0 && user_leak_offset < prog_map.size) {
        prog_map[ user_leak_offset ]();
    }
}
```

If문 1번

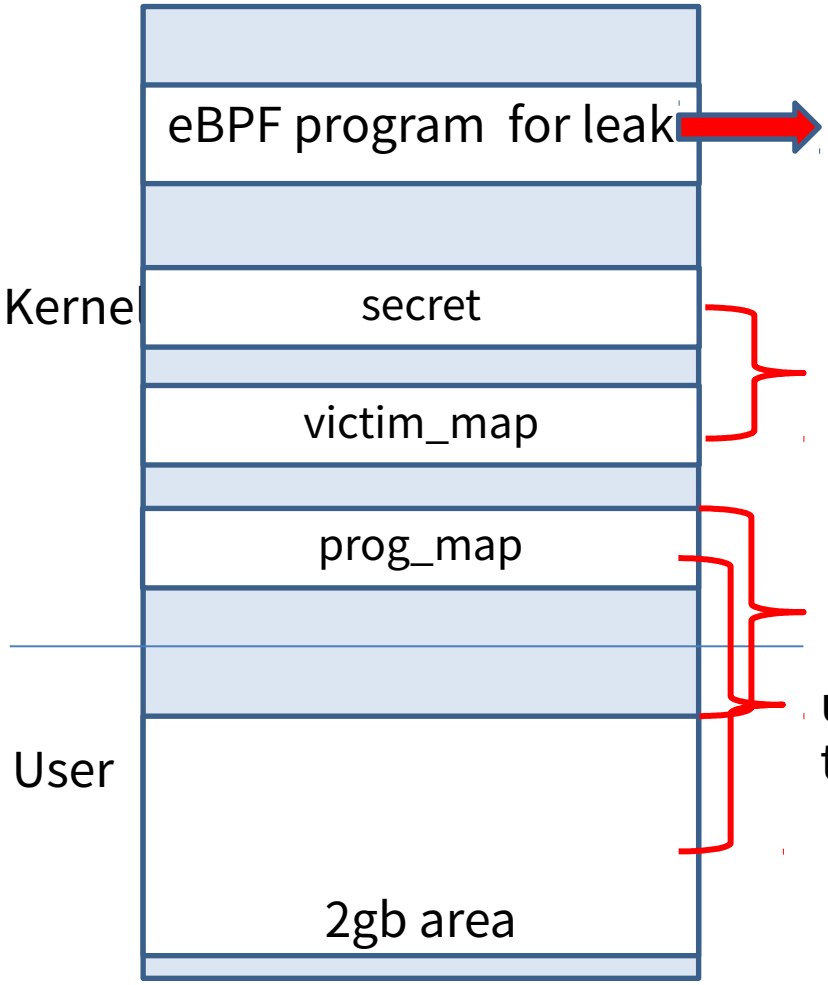
If문 2번

- data_map : 공격에 필요한 data 세팅할 map
- victim_map : kernel memory 값 (secret) 알아올 때 사용할 array map.
- prog_map : secret, prog_map_offset 에 따라 user memory (area) 접근할 용도로 사용되는 함수포인터 map.

- secret : kernel memory 값.
- prog_map_offset : 이전 단계에서 찾은 prog_map <-> area 에 대한 offset.
- user_leak_offset : secret 값에 따른 메모리 access 를 하기 위한 최종 offset.

- 위 If 문 1번, 2번 둘 모두에서 branch prediction 발생해야 함!

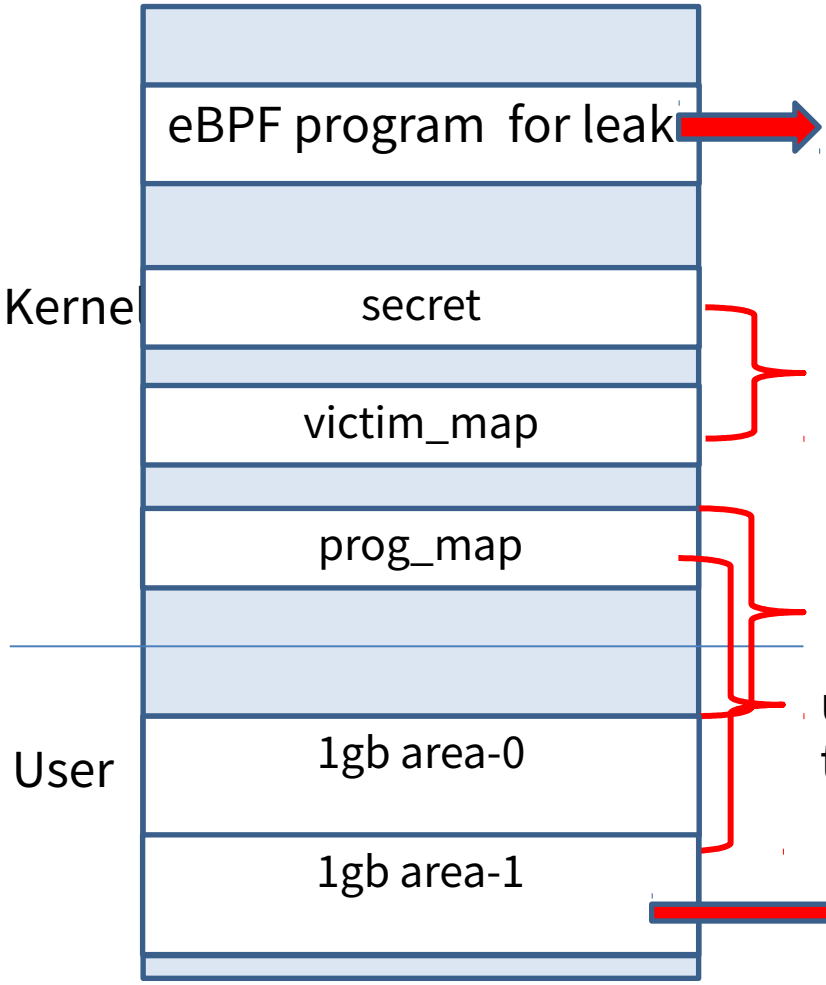
Linux kernel memory leak 을 위한 Memory Layout




```
data_map[] = {secret_offset, prog_map_offset, bitmask, bitshift};
if (secret_offset >= 0 && secret_offset < victim_map.size) {
    secret = victim_map[secret_offset];
    user_leak_offset = calc_prog_map_offset_with_secret(secret, prog_map_offset);

    if (user_leak_offset >= 0 && user_leak_offset < prog_map.size) {
        prog_map[ user_leak_offset ]();
    }
}
```

Linux kernel memory leak 을 위한 Memory Layout




```
data_map[] = {secret_offset, prog_map_offset, bitmask, bitshift};
if (secret_offset >= 0 && secret_offset < victim_map.size) {
    secret = victim_map[secret_offset];
    user_leak_offset = calc_prog_map_offset_with_secret(secret, prog_map_offset);

    if (user_leak_offset >= 0 && user_leak_offset < prog_map.size) {
        prog_map[ user_leak_offset ]();
    }
}
```

secret_offset

prog_map_offset

user_leak_offset

- secret 값을 bit 단위로 leak.
- bitmask, bitshift input 을 통해, secret byte 중 몇번째 bit 를 leak 할지 선정.
- secret == 0 이면, area-0 접근하도록 user_leak_offset 을 계산함.

Linux kernel memory leak 을 위한 전체 공격 Step

eBPF program
for memory leak

```
data_map[] = {secret_offset, prog_map_offset, bitmask, bitshift};
if (secret_offset >= 0 && secret_offset < victim_map.size) {
    secret = victim_map[secret_offset];
    user_leak_offset = calc_prog_map_offset_with_secret(secret, prog_map_offset);

    if (user_leak_offset >= 0 && user_leak_offset < prog_map.size) {
        prog_map[ user_leak_offset ]();
    }
}
```

(1) secret_offset == 0, user_leak_offset == 0 이 되도록 input 선정. 즉, if 문 training 하기 위한 정상 input 을 선정.
(2) eBPF program 을 N 번 실행. 두 if 문이 예측 실행되도록 함.
(3) user space “area” 메모리 cache flush.
(4) victim_map, prog_map 을 cache line bouncing 이용하여 cache 에서 제거.
(5) 공격자가 leak 하려는 메모리를 위한 secret_offset, bitmask, bitshift 값 선정.
(6) eBPF program 을 실행.
(7) If 문 1번에서 예측 실행 발생. 커널 메모리 값이 secret 에 저장됨.
(8) secret byte 의 1 bit 를 알아내기 위한 user_leak_offset 을 계산. (secret, prog_map_offset 을 가지고 계산)
(9) If 문 2번에서 예측 실행 발생. prog_map[user_leak_offset] 이 실행되면서, user space “area” 접근됨.
(10) 공격자는 “area-0”, “area-1” 중 어디 접근되었는지 확인. 해당 커널메모리 bit 가 0인지 1인지 판단.

 (1)~(10) 과정을 secret_offset 값 변경하면서 반복. 모든 커널 memory dump.

Appendix-4

-- Exploiting barrier

Barrier patch for fixing Variant1

```
(1) ptr = *(fp-144);
(2) *ptr = fp-216;
(3) ptr2 = *(fp-72);
(4) secret = *ptr2;
(5) if (secret == 0)
    exit;
(6) *leak_area;
```

Non-patched code

```
(1) lfence();
(2) ptr = *(fp-144);
(3) *ptr = fp-216;
(4) lfence();
(5) ptr2 = *(fp-72);
(6) lfence();
(7) secret = *ptr2;
(8) if (secret == 0)
    exit;
(9) lfence();
(10) *leak_area;
```

Barrier-patched code

- Barrier patch 의 경우, eBPF program 에서 수행되는 모든 명령어 앞에 lfence 라는 barrier 를 삽입. (Intel) lfence 란?? lfence 이전의 로드명령어가 끝나기 전에, lfence 이후의 로드명령을 예측실행시키지 말라는 것.

Step-1 : Speculative fetching & decoding

```
(1)  lfence();
(2)  ptr = *(fp-144);
(3)  *ptr = fp-216;
(4)  lfence();
(5)  ptr2 = *(fp-72);
(6)  lfence();
(7)  secret = *ptr2;
(8)  if (secret == 0)
       exit;
(9)  lfence();
(10) *leak_area;
```

lfence 실행되는 동안,
이후 로드 명령어 실행은 안되지만,
모두 실행대기상태로 만들어둘 수 있음!!

(1) 실행되는 동안, 모두 실행대기 상태가 됨.

Barrier-patched code

- Barrier patch 의 경우, eBPF program 에서 수행되는 모든 명령어 앞에 lfence 라는 barrier 를 삽입. (Intel)
  lfence 란?? lfence 이전의 로드명령어가 끝나기 전에, lfence 이후의 로드명령을 예측실행시키지 말라는 것.

Step-2 : Jumping over the barrier

```
(1) lfence();
(2) ptr = *(fp-144);
(3) *ptr = fp-216;
(4) lfence();
(5) ptr2 = *(fp-72);
(6) lfence();
(7) secret = *ptr2;
(8) if (secret == 0)
    exit;
(9) lfence();
(10) *leak_area;
```

store, weakly-ordered memory type 에 대한 연산은 대기하지 않음!!
(Not clear)

Barrier-patched code

- Code gadget-2가 모두 실행대기 상태에 있으므로, out-of-order execution 에 의해 Code gadget-1보다 먼저 실행 가능.
  하지만 그러러면 (4) lfence 를 넘어야만 함. (lfence 가 다음 명령이 먼저 실행되는 걸 막으므로)

- 예외적으로 lfence 이전의 store 명령, weakly-ordered memory 에 대한 명령의 경우, lfence 이후의 명령이 먼저 실행될 수 있음. 따라서 몇몇 예외 경우를 제외하고, 간헐적으로 Code gadget-2 가 먼저 실행 가능!

Thank you