Meltdown and Spectre Samples

Written in Assembly

U. Plonus

March 8, 2018

Contents

1 Introduction

1.1 Overview

TBD

1.2 Nasm

TBD

5 ⟨*preamble* 5⟩≡ (7a)

```
bits 64
      global _start
```

2 Cache Access Timing

2.1 Basics

TBD

2.2 Detect Cache Access Time

TBD

7a ⟨*cachetiming.asm* 7a⟩≡

```
⟨preamble 5⟩

⟨cachetiming-uninitialized-data 7b⟩

section .text
_start:
⟨exitProgram 11b⟩

⟨xorshift-prng 12a⟩

⟨utilities 11a⟩
```

TBD

7b ⟨*cachetiming-uninitialized-data* 7b⟩≡ (7a)

```
section .bss
      measures: resq 2048
      padding:  resb 4096
      align 4096
      data:     times 257 resb 4096
```

2.3 Read Array via Cache Access Time

TBD

3 Signals

3.1 Detecting Signals

TBD

3.2 Handling Signals

TBD

4 Utilities

4.1 Introduction

TBD

11a ⟨*utilities* 11a⟩≡ (7a)
  ⟨*print* 12b⟩

4.2 Exit Program

TBD

11b ⟨*exitProgram* 11b⟩≡ (7a)

```
      xor      RDI,RDI
      mov      RAX,60
      syscall
```

Uses `RAX` and `RDI`.

4.3 Random Number Generator

To initialize the data a random number generator is used. The sample programs use xorshift[1] as random number generator.

The start of the memory area to fill with random numbers is given in `RDI` and the number of bytes to fill in `RSI`. `RSI` must be a multiple of 4. The seed of the rng is given in `EDX`.

12a ⟨*xorshift-prng* 12a⟩≡ (7a)

```
_xorshift:
        mov     RCX,RSI
        shr     RCX,2
.next_random:
        mov     EBX,EDX
        shl     EDX,13
        xor     EDX,EBX
        mov     EBX,EDX
        shr     EDX,17
        xor     EDX,EBX
        mov     EBX,EDX
        shl     EDX,5
        xor     EDX,EBX
        stosd
        loop    .next_random
        ret
```

Uses `EBX`, `EDX`, `RCX`, and `RSI`.

4.4 Printing Text

The routine `_print` prints a null-terminated string to the terminal (stdout). The only argument passed in to the routine (in `RDI`) is the address of the string to print.

So first we start with clearing `AL` (setting it to null) and saving the address of the string to `RSI`. We're using `RSI` because we later need the address to calculate the length of the string and also `RSI` is the register that we need to use for the string address in the systemcall.

12b ⟨*print* 12b⟩≡ (11a) 13a ▷

```
_print:
        xor     AL,AL
        mov     RSI,RDI
```

Uses `AL`, `RDI`, and `RSI`.

[1] https://en.wikipedia.org/wiki/Xorshift

Next we search for the terminating `null` ('`\0`') character. For this we use the instruction `scasb` (scan string byte) which compares the byte at the address `[RDI]` with the value in `AL` and sets the flags accordingly. When the byte at `[RDI]` is not the value of `AL` the the next instruction (`jne`) jumps to the given label (`.next_char` in this case).

`scasb` additionally increments `RDI` so that we go through the string until `\0` is found.

13a ⟨*print* 12b⟩+≡ (11a) ◁12b 13b▷

```
.next_char:
        scasb
        jne       .next_char
```

After we have found the string termination we calculate the number of bytes that the string has. For this we copy the value of the last byte read (which is in `RDI`) to `RDX` and subtract the start of the string (which we saved to `RSI`).

Now we have the address of the string in `RSI` and the length of the string in `RDX` which are the 2nd and 3rd argument in a systemcall. The 1st argument (in `RDI`) to the systemcall is the file descriptor (`1` is stdout). Additionally the number of the systemcall (`1`) is passed in `RAX`. The systemcall (`syscall`) now prints `RDX` bytes from `[RSI]` to the file descriptor `RDI`.

13b ⟨*print* 12b⟩+≡ (11a) ◁13a 13c▷

```
        mov       RDX,RDI
        sub       RDX,RSI
        mov       RAX,1
        mov       RDI,1
        syscall
```

Uses `RAX`, `RDI`, `RDX`, and `RSI`.

Now that we are done and can return to the caller.

13c ⟨*print* 12b⟩+≡ (11a) ◁13b

```
        ret
```

4.5 Printing Numbers

TBD