Meltdown and Spectre Samples

Written in Assembly

U. Plonus

March 9, 2018

Contents

1 Introduction

1.1 Overview

TBD

1.2 Nasm

TBD

5 ⟨*preamble* 5⟩≡ (8a)

```
bits 64
        global          _start
        pagesize        equ  4096
```

2 Cache Access Timing

2.1 Introduction

TBD

2.2 Detect Cache Access Time

TBD

7a ⟨*tsc-64bit* 7a⟩≡ (7b)

```
        rdtsc
        shl     RDX,32
        add     RAX,RDX
```

TBD

7b ⟨*calculate-cache-access-time* 7b⟩≡ (8a)

```
_calccachetime:
        xor     RAX,RAX
        xor     RDX,RDX
        lfence
⟨tsc-64bit 7a⟩
        mov     R8,RAX
        mov     RCX,[RDI]
        lfence
⟨tsc-64bit 7a⟩
        sub     RAX,R8
        ret
```

2.3 Detect Cache Access Time

TBD

8a ⟨*cachetiming.asm* 8a⟩≡

```
⟨preamble 5⟩

⟨cachetiming-rodata 8c⟩

⟨cachetiming-uninitialized-data 8b⟩

section .text
_start:
     mov       RDI,suncached
     call      _print
⟨exitProgram 11b⟩

⟨calculate-cache-access-time 7b⟩

⟨xorshift-prng 12a⟩

⟨utilities 11a⟩
```

TBD

8b ⟨*cachetiming-uninitialized-data* 8b⟩≡ (8a)

```
section .bss
     align          pagesize
     data:          times 2 resb pagesize
```

TBD

8c ⟨*cachetiming-rodata* 8c⟩≡ (8a)

```
section .rodata
     suncached:     db "Uncached Access Time: ",0x00
     scached:       db "Cached Access Time: ",0x00
```

2.4 Read Array via Cache Access Time

TBD

3 Signals

3.1 Basics

TBD

3.2 Detecting Signals

TBD

3.3 Handling Signals

TBD

4 Utilities

4.1 Introduction

TBD

11a ⟨*utilities* 11a⟩≡ (8a)

⟨*nprint* 13a⟩

⟨*print* 13b⟩

4.2 Exit Program

TBD

11b ⟨*exitProgram* 11b⟩≡ (8a)

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

4.3 Random Number Generator

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

Parameters

RDI the address of the memory which is to be filled with random numbers

RSI the number of bytes that are filled with random numbers. This must be a multiple of 4

EDX the seed of the RNG

First we move the number of values to be generated to `RCX` (which is a counter in x86 processors) and divide it by 4 (because we use a 32bit RNG). Additionally we move the seed to `EAX`.

12a ⟨*xorshift-prng* 12a⟩≡ (8a) 12b ▷

```
_xorshift:
     mov     RCX,RSI
     shr     RCX,2
     mov     EAX,EDX
```

Now we can generate the next 32bit random number.

12b ⟨*xorshift-prng* 12a⟩+≡ (8a) ◁ 12a 12c ▷

```
.next_random:
     mov     EBX,EAX
     shl     EAX,13
     xor     EAX,EBX
     mov     EBX,EAX
     shr     EAX,17
     xor     EAX,EBX
     mov     EBX,EAX
     shl     EAX,5
     xor     EAX,EBX
```

Because we want to generate multiple random numbers we store the value of `EAX` to `[RDI]` and loop for the next random number.

12c ⟨*xorshift-prng* 12a⟩+≡ (8a) ◁ 12b

```
     stosd
     loop    .next_random
     ret
```

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

4.4 Printing Strings

4.4.1 Printing Strings with Length

The routine `_nprint` prints a string with the given length to `stdout`.

Parameters

RDI the number of bytes to print to `stdout`

RSI the address to the bytes to print to `stdout`

We move the number of bytes to print to `RDX` which is the 3rd parameter to the systemcall. Next we move the address of the bytes to print to `RSI` which is the 2nd parameter to the 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`.

At the end we return to the caller.

13a ⟨*nprint* 13a⟩≡ (11a)

```
_nprint:
      mov      RDX,RSI
      mov      RSI,RDI
      mov      RDI,1
      mov      RAX,1
      syscall
      ret
```

4.4.2 Printing C-Strings

The routine `_print` prints a null-terminated string to `stdout`.

Parameters

RDI the address to the null-terminated bytes to print to `stdout`

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.

13b ⟨*print* 13b⟩≡ (11a) 14a ▷

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

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.

14a ⟨*print* 13b⟩+≡ (11a) ◁13b 14b▷

```
.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 exchange the registers `RDI` and `RSI`. In `RDI` we now have the starting address of the bytes to print and in RSI we have the end address of the bytes to print. After that we calculate the number of bytes to print.

14b ⟨*print* 13b⟩+≡ (11a) ◁14a 14c▷

```
      xchg      RDI,RSI
      sub       RSI,RDI
```

TODO 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`.

14c ⟨*print* 13b⟩+≡ (11a) ◁14b

```
      call      _nprint
      ret
```

4.5 Printing Numbers

TBD

A Glossary

x86 x86 denotes a microprocessor architecture based on the 8086/8088 12

B Acronyms

RNG random number generator 12

C Code Chunks