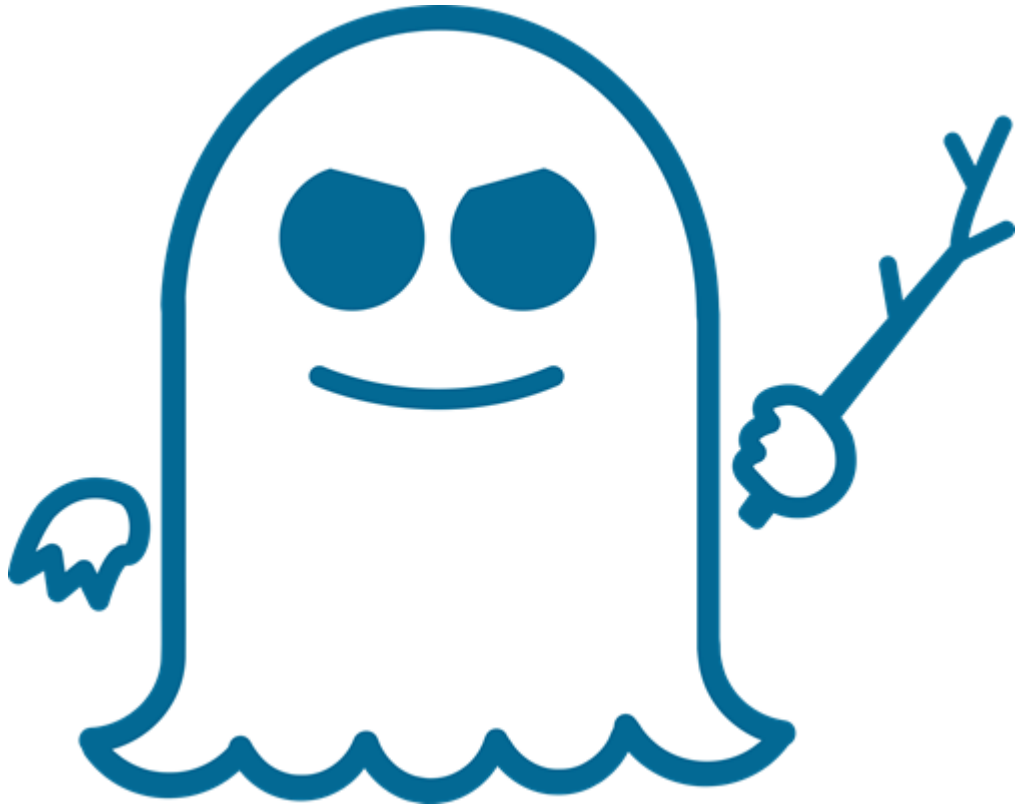


Spectre,

the juicy parts

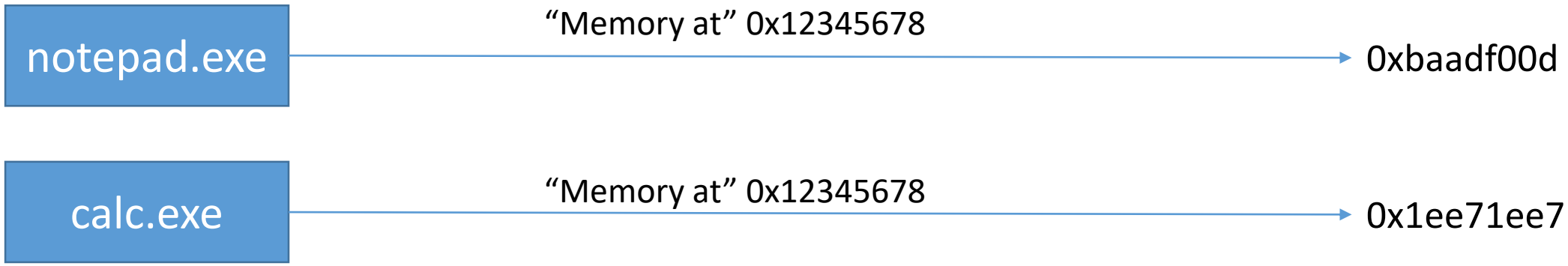
Ofek, Jan 2018

Organization

- Process Isolation
- Process Isolation loopholes:
  - Flush & Reload
  - Branch Poisoning
  - Gadgets
  - Tying it together (==Spectre)
- Optional:
  - Potential Mitigations
  - Impact

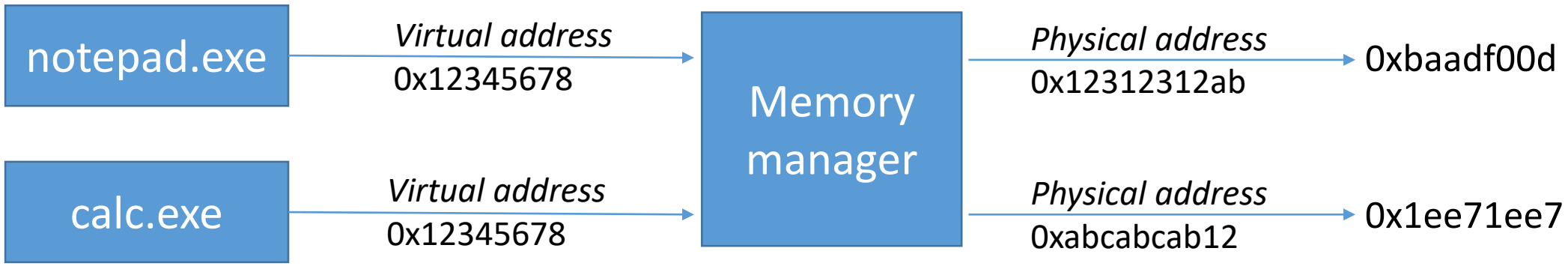
Process Isolation

- The CPU & OS present to each process a worldview wherein it is the only one in the world.



Process Isolation

- The CPU & OS present to each process a worldview wherein it is the only one in the world.



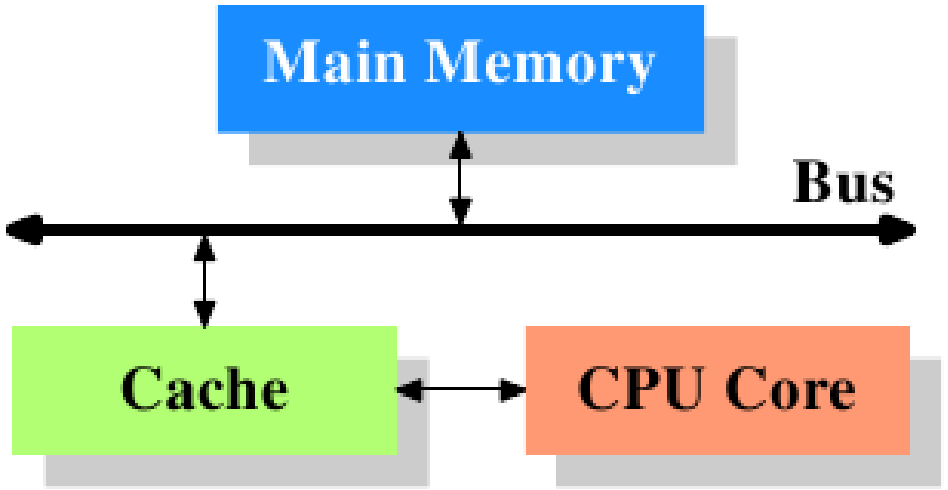
A. Flush + Reload Attack

Yuval Yarom, Katrina Falkner, 2014
https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-yarom.pdf

Extends on ‘Prime & Probe’ 2010 paper, and ‘Cache Games’ from 2011.

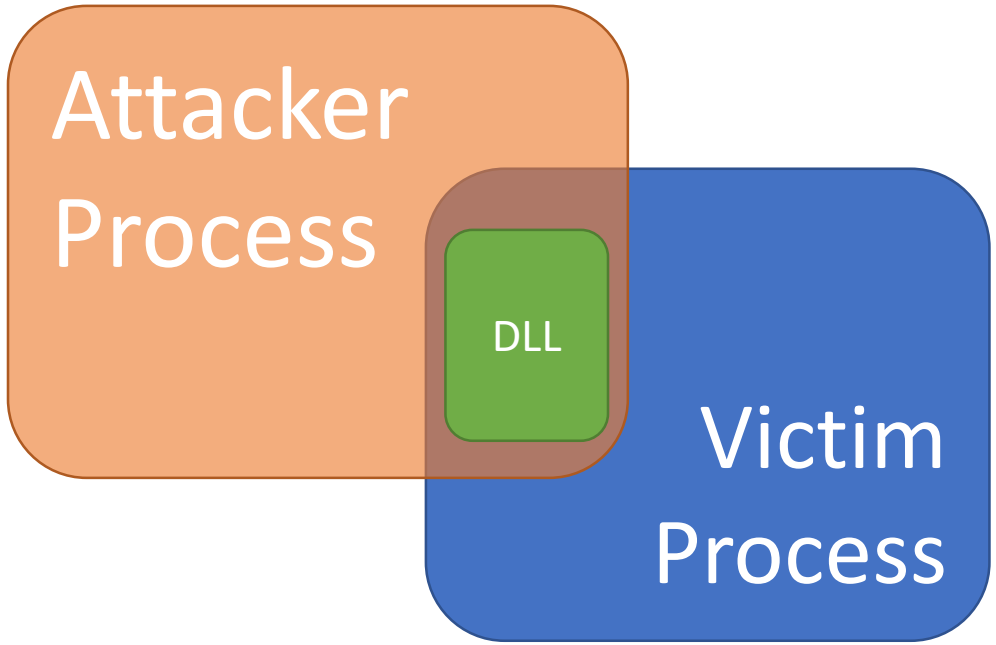
Process Isolation Loophole 1: Caches

- Memory hierarchy
  - L1 i/d ~0.5 ns
  - L2 per core ~7 ns
  - L3 common ~25 ns
  - RAM ~100ns



- Shared Pages
  - Common DLLs are loaded at memory *once*, used by multiple processes.
  - Tagged at the cache level too.

Flush & Reload



Basic attacker course of action:

- Load the victim DLL of interest
- For any particular address within the DLL:
  - Make sure it is out of the cache
  - Access it, measure the time.
  - Less than 80 ns? ***Just used by the victim.***

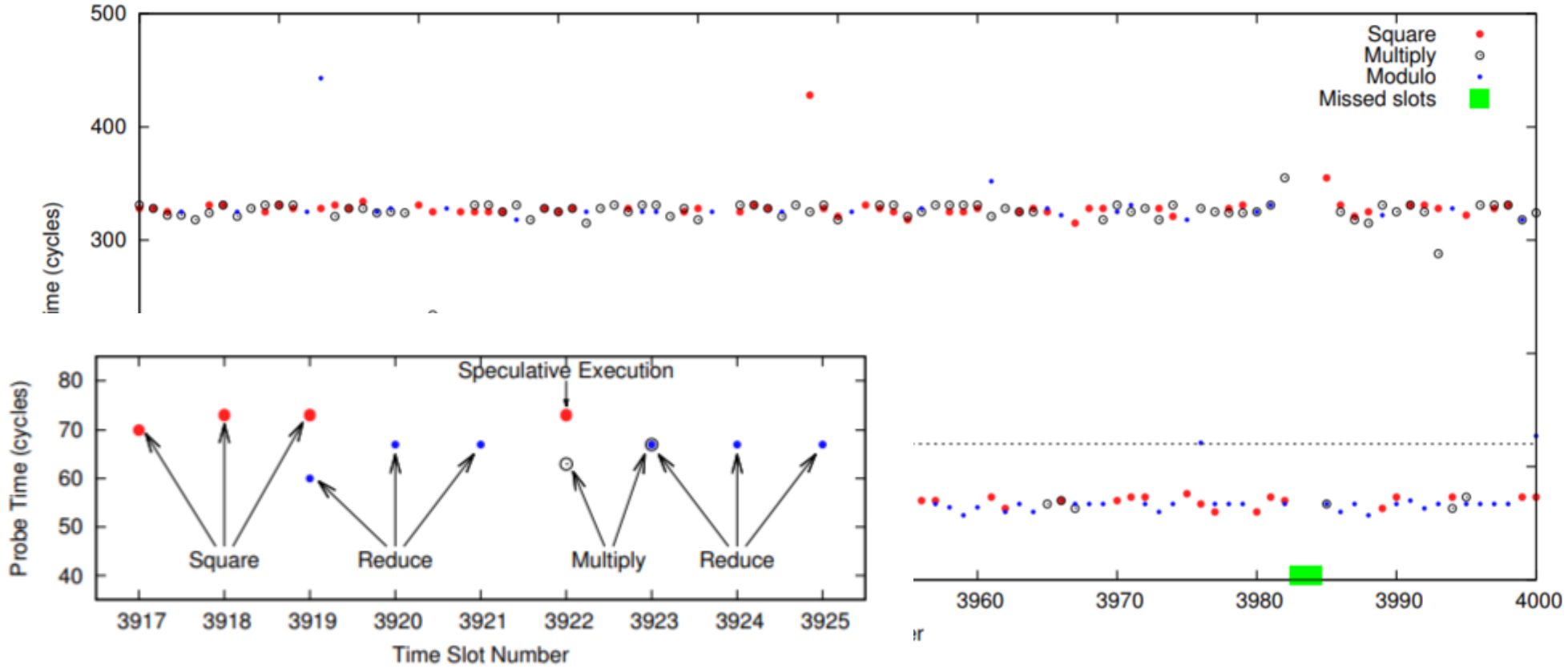
Flush & Reload

- Original attack traced code execution in GPG (Gnu PrivacyGuard), an open source RSA implementation.

*"…Each occurrence of **Square-Reduce-Multiply-Reduce** within the sequence corresponds to a bit whose value is 1.*

*Occurrences of **Square-Reduce** that are not followed by a Multiply correspond to bits whose values are 0. …"*

Flush & Reload



On some systems, this attack had 99% success in obtaining RSA keys

Flush + Reload

- 2017 Novelty: leak *values* in memory, not just address usage!

```
y = array2[array1[x] * 256];
```

- The value in x dictates the memory chunk accessed.

Process Isolation Loophole 2: Speculative Execution

- *CPUs can't wait,* and execute ahead of time (Out Of Order).
- Naïve execution induces many such waits.

```
…
cmp   [var], 0
jne    SomeFunc
…
```

```
…
call [eax]
…
```

Process Isolation Loophole 2: Speculative Execution

- Several mechanisms are in place for ‘informed guesses’ on branch destinations.
- *Mostly*, this branch will be true:

```
for (int i=0; i<1000; ++i)
```

- Virtual function calls (`call [eax]`) in a succession from the same address, *mostly* direct to the same destination.

Process Isolation Loophole 2: Speculative Execution

- CPUs maintains ‘history caches’ to predict branch destinations and perform ***speculative execution***. When the branch is retired, speculative results based on wrong guesses are dumped.

- The loopholes:
  1. *Branch prediction is per-processor, not per process. A process can **train** a branch, and thereby direct speculative execution on a different process.*
  2. *Speculative execution can go where regular code can’t. (Meltdown)*

Spectre I

Suppose:

1. this victim process has a code snippet similar to:

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

2. The attacker can control x - via file, external parameter etc.

Spectre I

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

- Then the attacker process:
  1. Trains the branch `x < array1_size` to be true,
  2. Flushes x out of the cache.

A CPU delay is incurred at the branch.
The statement `array2[array1[x]*256]` is executed speculatively - without bounds restrictions!

  3. Sniffs x via flush & reload.

C. Gadgets

Return Oriented Programming

Hovav Shacham 2007
https://cseweb.ucsd.edu/~hovav/dist/geometry.pdf

Process Isolation Loophole 3: Existing code is rich enough to do anything

- In a large enough piece of machine code, you can find anything you want.
  - The SW equivalent of a million monkeys typing for a million years.
- In particular for x86/x64 architectures: dense code.

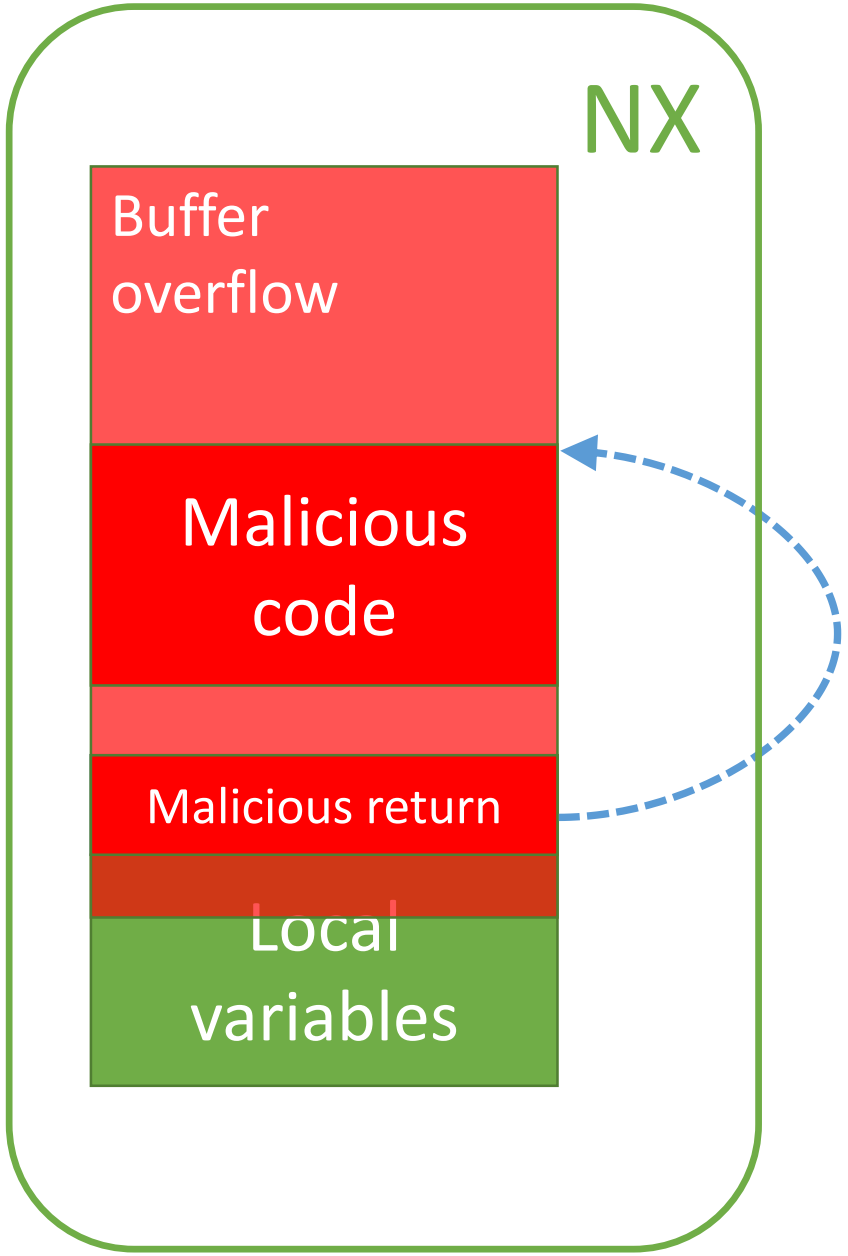
Gadgets

```
00007FF9A80C09B0 4C 8B D1              mov         r10,rcx
00007FF9A80C09B3 B8 5B 00 00 00        mov         eax,5Bh
00007FF9A80C09B8 F6 04 25 08 03 FE 7F 01 test          byte ptr [7FFE0308h],1
00007FF9A80C09C0 75 03                 jne         NtWaitForMultipleObjects+15h (07FF9A80C09C5h)
00007FF9A80C09C2 0F 05                 syscall
00007FF9A80C09C4 C3                    ret
```

```
00007FF9A80C09B6 ??                    ?? ??
00007FF9A80C09B7 ??                    ?? ??
00007FF9A80C09B8 ??                    ?? ??
00007FF9A80C09B9 ??                    ?? ??
00007FF9A80C09BA 25 08 03 FE 7F        and         eax,7FFE0308h
00007FF9A80C09BF 01 75 03              add         dword ptr [rbp+3],esi
00007FF9A80C09C2 0F 05                 syscall
00007FF9A80C09C4 C3                    ret
00007FF9A80C09C5 CD 2F                 int         2Fh
```

Context: Return Oriented Programming

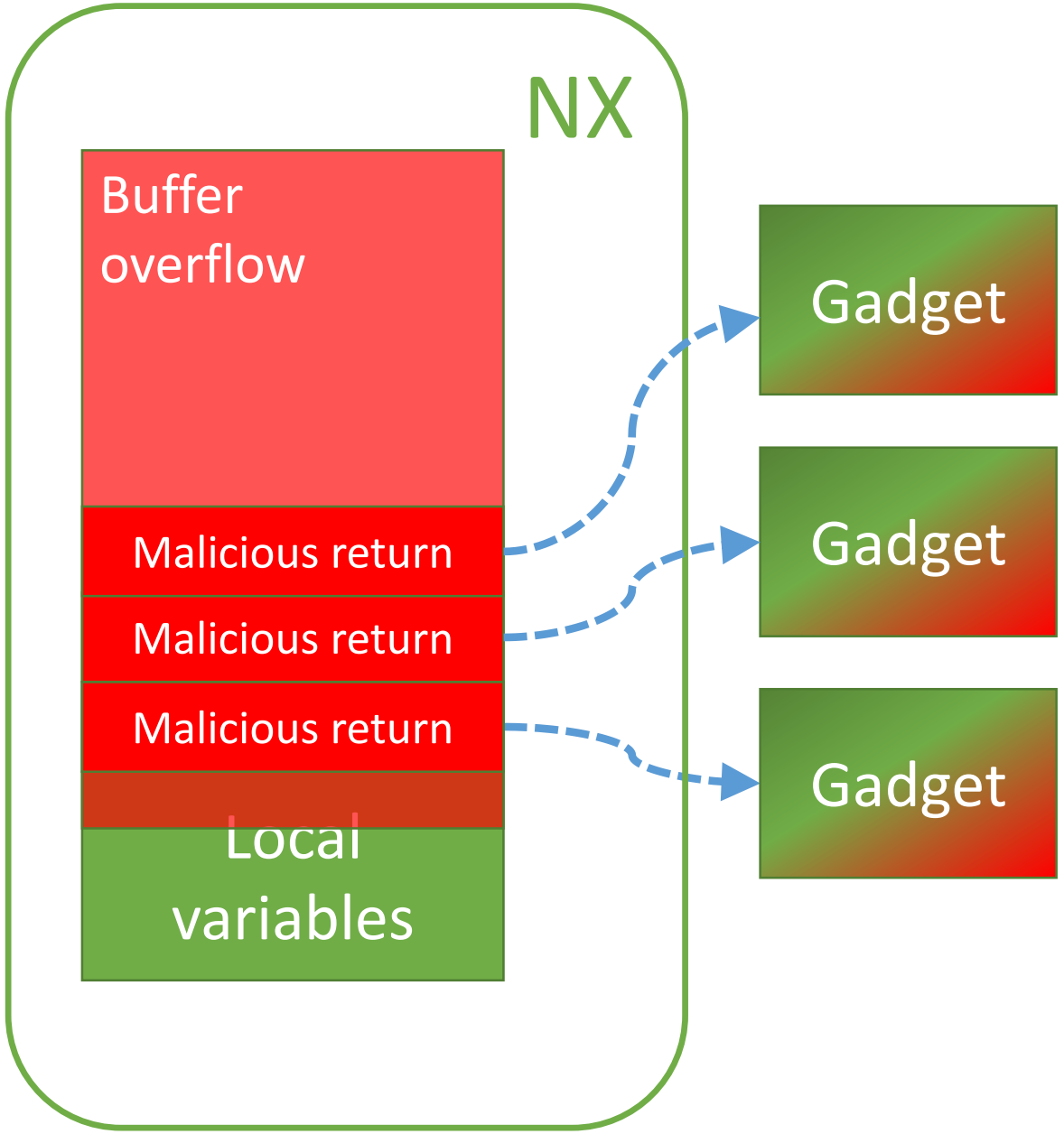
- History:
  - Stack-smashing
  - DEP



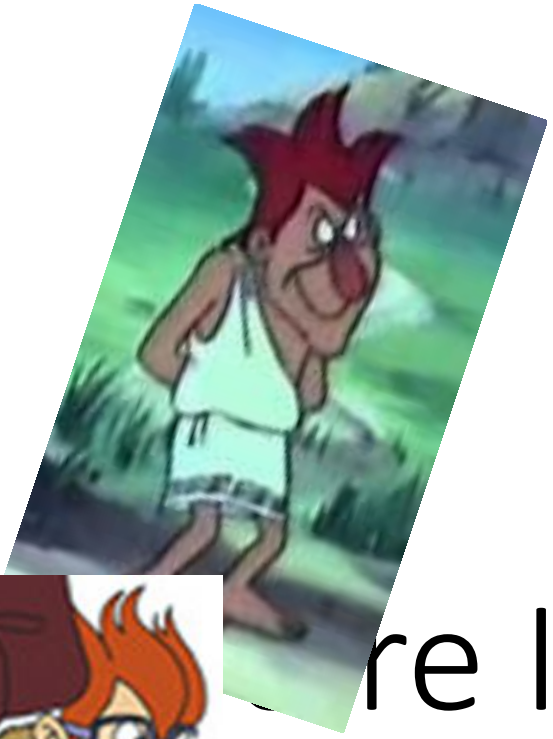
Context: Return Oriented Programming

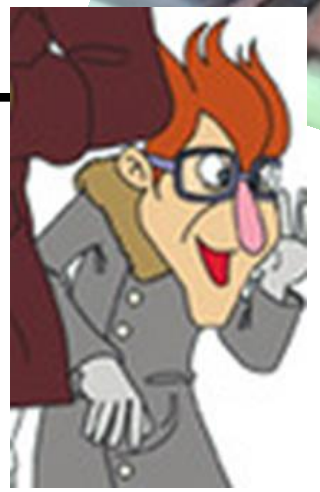
- History:
  - Stack-smashing
  - DEP
  - RoP

- More or less turing-complete machine from gadget building blocks.
- Code Execution exploit, of buffer overrun vulnerability



D. Tyi[illegible] al[illegible]er – [illegible]re II





Abilities thus far

- Process Isolation
- Process Isolation loopholes:
  - Flush & Reload
  - Branch Poisoning
  - Gadgets
  - Tying it together (==Spectre)
- Optional:
  - Potential Mitigations
  - Impact

3. Sniff memory value at address accessed by the victim

2. Direct **speculative execution** at victim process to the gadget

1. Find **gadgets** to direct execution to.

Spectre II - Toy Example Attack

1. At the victim process, locate this gadget:

```
mov eax [ebx]
add 1 [eax]
```

2. At joint attacker/victim process, locate this frequently-executed branch:

```
jmp [0x00123456]
```

3. Suppose at the branch location ebx is some unused external function argument.
4. Control ebx to point to a memory location of interest.
5. At the attacker process, train the jump to reach the gadget. It is now executed speculatively.
6. At the attacker process, use Flush and Reload to sniff which address was read.

Spectre II – PoC

1. The victim gadget: the sequence of bytes
   `13 BC 13 BD 13 BE 13 12 17` found in ntdll.dll:

```
adc   edi,dword ptr [ebx+edx+13BE13BDh]
adc   dl,byte ptr [edi]
```

2. The victim branch:

```
Sleep(0) :
   jmp dword ptr ds:[76AE0078h]
```

   At a point where ebx and edi contain file data and are ignored by Sleep()

Spectre II - General

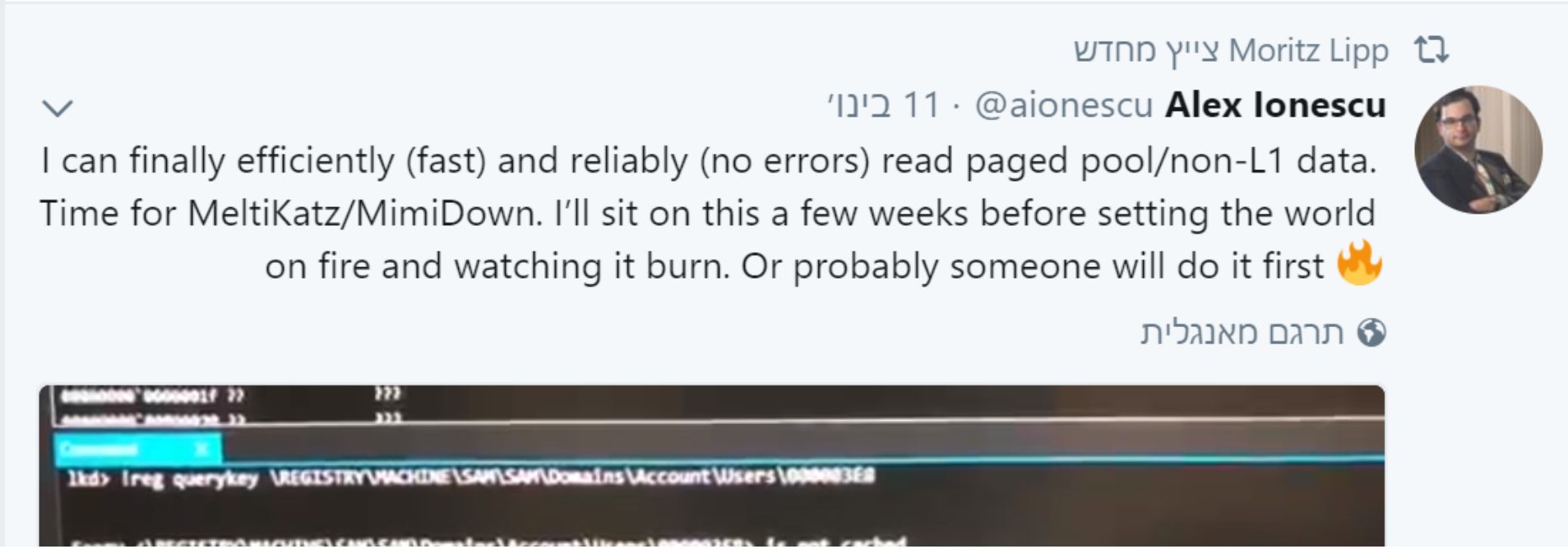
1. At the victim process, find a gadget that accesses memory via the attacker controlled registers.
2. At the victim process, find a frequently executed indirect jump:

   `Jmp / call` `eax / [eax] / [0x12345678]`

   at a site where the attacker can control some registers.
3. At the attacker process, train the jump to reach the gadget.
4. At the attacker process, use Flush and Reload to sniff the speculative execution results.

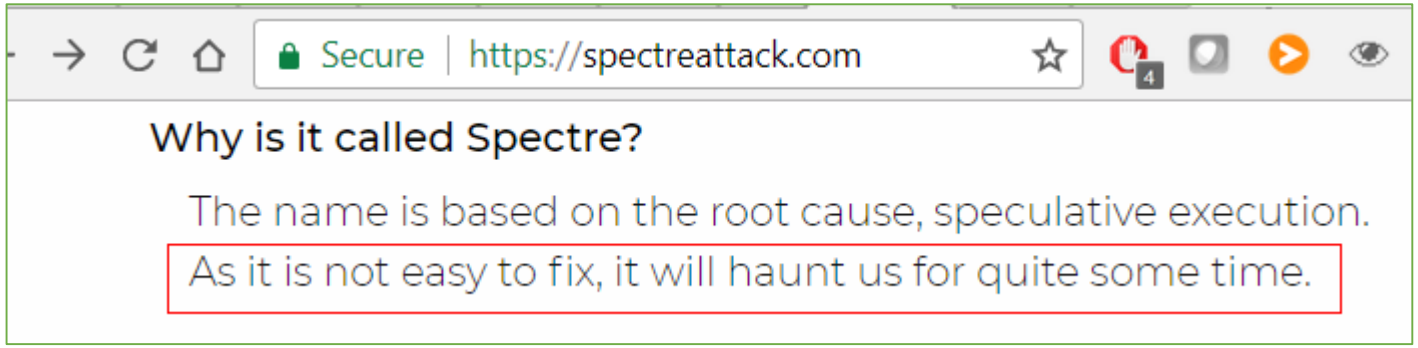
Impact

- https://spectreattack.com/



Mitigations

- Timer resolution
- Speculation Barrier (https://github.com/ARM-software/speculation-barrier)
  - Hard on performance (image)
- Retpoline (http://lists.llvm.org/pipermail/llvm-commits/Week-of-Mon-20180101/513630.html)
- Not really.



Q?