Two methods for exploiting speculative control flow hijacks

Andrea Mambretti

Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Anil Kurmus
### Multiple vulnerabilities

<table>
<thead>
<tr>
<th>CVE</th>
<th>Variant</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017-5753</td>
<td>Spectre v1</td>
<td>Bounds Check Bypass</td>
</tr>
<tr>
<td>2017-5715</td>
<td>Spectre v2</td>
<td>Branch Target Injection</td>
</tr>
<tr>
<td>2017-5754</td>
<td>Meltdown</td>
<td>Rogue Data Cache Load</td>
</tr>
<tr>
<td>2018-3640</td>
<td>Spectre v3a</td>
<td>Rogue System Register Read</td>
</tr>
<tr>
<td>2018-3639</td>
<td>Spectre v4</td>
<td>Speculative Store Bypass</td>
</tr>
<tr>
<td>2018-3665</td>
<td>Spectre-FP</td>
<td>Lazy FP State Restore</td>
</tr>
<tr>
<td>2018-3693</td>
<td>Spectre v1.1</td>
<td>Bounds Check Bypass Store</td>
</tr>
</tbody>
</table>
Spectre v1 - Bounds Check Bypass

```java
if (x < array1_size) {
    y = array2[array1[x]]
}
```

Example:
- `array1_size = 8`
- `x = 15` (attacker controlled)

Speculative Execution Trigger

```
array1 = [s, 3, c, r, 3, 7]
array2 = [array2, ...]
```

```
array2 + 0x63
```
Speculative CFH Attack Breakdown

Attacker injection
  (e.g. Branch Predictor Training)

Speculative Control Flow Hijack
  lure the victim to execute the vulnerable code

*Side Channel Send* gadget executed inside the victim

*Side Channel Receive* gadget executed inside the attacker
Branch Target Buffer

<table>
<thead>
<tr>
<th>f(PC)</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>E</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>B</td>
</tr>
<tr>
<td>Z</td>
<td>G</td>
</tr>
</tbody>
</table>

(Simplified) Branch Target Buffer

Normal Exec

mov rax, [0xc0ff33]

A: call *rax

E: mov rbx, 20
ret

V: mov rbx, 10
ret

Memory

0xc0ff33

Not cached

44

E

1235

666
Spectre v2 - Branch Target Injection (BTI)

(Simplified) Branch Target Buffer

<table>
<thead>
<tr>
<th>f(PC)</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Evil</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>B</td>
</tr>
<tr>
<td>Z</td>
<td>G</td>
</tr>
</tbody>
</table>

Victim Thread

- mov rax, [0xc0ff33]
- call *rax
- E: mov rbx, 20
- ret
- V: mov rbx, 10
- ret
- Evil: GADGET

Attacker Thread

- repeat: mov rax, Evil
- loop repeat
- Evil: ret
- Leak: GADGET

Memory

- 44
- E
- 1235
- 666
- 0xc0ff33 Not cached
Control Flow Hijack - Gadget

Spectre v2 and other CF hijack techniques use Spectre v1 gadget as “side channel send”

Project Zero Spectre v2 Proof-of-Concept relies on Kernel extended Berkeley Packet Filter (eBPF) JIT mechanism to inject a suitable gadget

Are there other (easier to find) gadgets that can be used?
Our Contribution - New SC Send gadgets

Instruction cache:

timing the execution of a piece of code that is executed conditionally based on a secret

Branch Predictor (Double BTI):

let the victim program train the Branch Predictor using a secret computed value
Instruction Cache - POC

Attacker

mov rdx, Evil
A: call *rdx

rax = 0

rax != 0

B:

rdtsc
call fun1
rdtsc

time

Victim

mov rax, secret
A: call *rdx

Evil: cmp rax, 0
je end

B: call fun1
end: ...

fun1 is in a shared memory area between attack and victim process

BTI Gadget Hijack

Side Channel Send (i-cache gadget)

Training

Side Channel Receive (i-cache timing)
Double BTI - POC Phase 1

Attacker

```
mov rdx, Evil
A: call *rdx
```

```
Evil: ret
call *rax
ret
```

Victim

```
mov ax, secret_byte
shl eax, 16
add rax, BASE
mov rdx, Good
A: call *rdx
```

```
Evil: nop
call *rax
...
```

```
Good: ret
```

<table>
<thead>
<tr>
<th>f(PC)</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Evil</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>X</td>
<td>B</td>
</tr>
<tr>
<td>Z</td>
<td>G</td>
</tr>
<tr>
<td>Evil</td>
<td>fun(secret)</td>
</tr>
</tbody>
</table>

BTI Gadget Hijack

Side Channel Send (reverse BTI gadget)
Double BTI - POC Phase 2

Attacker

mov rdx, Evil
A: call *rdx

Evil:

nop
call *rax
ret

addr0: mov rax, QWORD[array + 0]
   ret
addr1: mov rax, QWORD[array + 1]
   ret
...
addr71: mov rax, QWORD[array+71]
   ret
...
addr255: mov rax, QWORD[array+255]
   ret

Side Channel Receive

Evil:  call *rdx

f(PC) | Target
---|---
A  | Evil
D  | E    
X  | B    
Z  | G
Evil | fun(secret)

fun(secret) => addrX with \{X ∈ \mathbb{N} | X ∈ [0, 255]\}
e.g. fun(0) = addr0, fun(255) = addr255

secret_byte = 71 = ‘G’

time
Results

Icache attack

<table>
<thead>
<tr>
<th>Secret</th>
<th>Success Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>80.84% +/- 1.37</td>
</tr>
<tr>
<td>1</td>
<td>97.29% +/- 0.11</td>
</tr>
</tbody>
</table>

Double BTI
Mitigations

Indirect Branch Restricted Speculation (IBRS) and Indirect Branch Predictor Barrier (IBPB) does not apply to user-space attacks.

Single Thread Indirect Branch Predictors (STIBP) mitigates our attacks

Current STIBP default setting leaves to the application the burden of requesting the protection through either SECCOMP, or the prctl interface.

Retpoline stops our attacks, though the application has to be recompiled with it
Conclusions

We introduced two new SC send gadgets and tested them in BTI attacks (applicable to other Control Flow Hijack attacks, e.g. ret2spec)

Through the I-cache gadget we can leak 1 bit at the time

Through the double BTI gadget we can leak 1 byte at the time with very good signal

Current mitigations do not protect applications unless specifically requested
Questions?