EXPRACE: Exploiting Kernel Races through Raising Interrupts

Yoochan Lee†, Changwoo Min‡, Byoungyoung Lee†
Seoul National University†, Virginia Tech‡
Race Condition

Initial state
M = 0

Execution 1
Thread 1
W M = 1
R read M

Execution 2
Thread 1
W M = 1
R read M

Different execution order
Different execution results!
Race Condition Bug

- Race bug consist of two or more race pairs
  - Must be executed in a specific order
- Difficult to succeed
  - Two threads must be executed in order
- Brute force somehow works still

Thread 1

Thread 2

Time Window

UAF, OOB, etc
Problem: Multi-Variable Race

<table>
<thead>
<tr>
<th>RACE</th>
<th>CVE</th>
<th>Bruteforce work?</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2019-11486</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>CVE-2017-7533</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>CVE-2017-2636</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>CVE-2016-8655</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>CVE-2019-6974</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>CVE-2019-2025</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>CVE-2019-1999</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>CVE-2017-15265</td>
<td>✗</td>
<td></td>
</tr>
</tbody>
</table>

• We found several races are exploited with bruteforce

• However, some races cannot be exploited with bruteforce
Problem: Multi-Variable Race

- Some races are (practically) impossible to exploit
- If $T_1 < T_2$

<table>
<thead>
<tr>
<th>CVE</th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2019-6974</td>
<td>18</td>
<td>1210</td>
</tr>
<tr>
<td>CVE-2019-2025</td>
<td>50</td>
<td>600</td>
</tr>
<tr>
<td>CVE-2019-1999</td>
<td>150</td>
<td>1800</td>
</tr>
<tr>
<td>CVE-2017-15265</td>
<td>35</td>
<td>450</td>
</tr>
</tbody>
</table>

Access to memory $M_1$

Access to memory $M_2$
Goal: How do we exploit Multi-variable Race?

Idea: Extend the time window to make $|T_1 + T_{\text{Extend}}| > |T_2|$, making the window larger than $T_2$. 

$|T_1| < |T_2|$
Idea: Extending the time window?

Preempting the thread execution
Typical preemption methods

Kernel breakpoint

Kernel thread schedule

interrupt
Challenge:
Preemption is not under user’s control

Back to OS basics:
Users cannot control preemption methods
User-controlled Preemption through Interrupts

Fact: Users **CANNOT** raise interrupts *directly*

We found that users **CAN** raise interrupts *indirectly*
User-controlled Preemption through Interrupts

We found **four methods to indirectly raise interrupts**

- Reschedule interrupt
- Membarrier interrupt
- TLB shootdown interrupt
- Hardware interrupt
Requirement: Precise Interrupt Control

We should send an interrupt to a desired CPU core!

- **CPU pinning**
  - Reschedule interrupt

- **Same memory map**
  - Membarrier interrupt

- **Same memory map**
  - TLB shootdown interrupt

- **CPU pinning**
  - Hardware interrupt
Our Approach: Interrupt

- The key insight behind EXPRACE is in intentionally enlarging race window using interrupt.

Core 0
Syscall() (related to attack method)
Send interrupt to Core 1

Core 1
Interrupt Handler

Core 2

- The key insight behind EXPRACE is in intentionally enlarging race window using interrupt.
Example : Hardware Interrupt

```
yoochan@compsec:~$ cat /proc/irq/121/smp_affinity_list
> Core 7
```

Core 0

```
sk = socket()
connect(sk)
```

Send interrupt to Core 1

Core 2

```
A

T_1

IRQ Handler

T_1 +

T_{Extend}

D

B

C

T_{Extend}

T_2
```
## Real-World Races in Linux

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Baseline</th>
<th>Reschedule</th>
<th>membarrier</th>
<th>TLB shootdown</th>
<th>HW interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2019-6974</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2019-2025</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2019-1999</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2017-15265</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>11eb85ec...</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1a6084f8...</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>20f2e4c2...</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>484298f...</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>da1b9564...</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>e20a2e9c...</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

✗ denotes exploitation has failed for given 24 hours

All 10 cases are exploited with EXPRACE
Other OSes

- Reschedule and TLB shootdown has shown far more success numbers than baseline.
Conclusion

• We analyzed real-world kernel races and found an intrinsic condition separating easy-to-exploit and hard-to-exploit races.

• We developed EXPRACE, a generic race exploitation technique for Linux, Windows, OS X.

• EXPRACE demonstrated that it truly augments the exploitability of kernel races.