Loophole: Timing Attacks on Shared Event Loops in Chrome

Pepe Vila and Boris Köpf
We exploit 2 different shared Event Loops in Chrome:
We exploit 2 different shared Event Loops in Chrome:

I/O’s of the **Host Process**

Main thread’s of **Renderers**
We exploit 2 different shared Event Loops in Chrome:

I/O’s of the **Host Process**

Main thread’s of **Renderers**

And implement 3 different attacks:
We exploit 2 different shared Event Loops in Chrome:

I/O’s of the **Host Process**

Main thread’s of **Renderers**

And implement 3 different attacks:

**Page Identification**
We exploit 2 different shared Event Loops in Chrome:

I/O’s of the **Host Process**

Main thread’s of **Renderers**

And implement 3 different attacks:

**Page Identification**

**Inter-keystroke Timing**
We exploit 2 different shared Event Loops in Chrome:

I/O’s of the **Host Process**

Main thread’s of **Renderers**

And implement 3 different attacks:

**Page Identification**

**Inter-keystroke Timing**

**Covert Channel**
Shared Event Loop

FIFO queue

Dispatcher

---
time
Shared Event Loop

FIFO queue

Dispatcher

\[ e_0 \]
Shared Event Loop

FIFO queue

Dispatcher

e_0
Shared Event Loop

FIFO queue

Dispatcher

e_0

e_1

time
Shared Event Loop

FIFO queue

Dispatcher

\[ e_1 \]

\[ e_0 \]

time
Shared Event Loop

FIFO queue

Dispatcher

\[ e_0 \]

\[ e_1 \]

time
Shared Event Loop

FIFO queue

e_2

Dispatcher

e_1

e_0

time
Shared Event Loop

FIFO queue

Dispatcher

\[ e_2 \]

\[ e_0 \quad e_1 \]

time
Shared Event Loop

FIFO queue

Dispatcher

\[ e_2 \]

\[ e_0 \quad e_1 \]

time
Shared Event Loop

FIFO queue

Dispatcher

\[ e_0, e_1, e_2 \]

\[ \text{time} \]
Shared Event Loop

FIFO queue

Dispatcher

\[ e_0, e_1 \]

time
Shared Event Loop

FIFO queue

Dispatcher

\[ e_2 \]

\[ e_0 \quad e_1 \]

\[ e \]

\[ e_3 \]

\[ \text{time} \]
Shared Event Loop

FIFO queue

Dispatcher

\[ e_0, e_1, e_2 \]

\[ \text{time} \]
Shared Event Loop

FIFO queue

e_3

dispatcher

e_0 e_1 e_2

time
Shared Event Loop

FIFO queue

Dispatcher

\[ e_3 \]

\[ e_0 \quad e_1 \quad e_2 \]

time
Shared Event Loop

FIFO queue

Dispatcher

e_3

e_0 e_1 e_2
Shared Event Loop

FIFO queue

Dispatcher

\[e_4\]

\[e_3\]

\[e_0\] \[e_1\] \[e_2\]
Shared Event Loop

FIFO queue

Dispatcher

\[ e_4 \]

\[ e_0 \quad e_1 \quad e_2 \quad e_3 \]

time
Shared Event Loop

FIFO queue

Dispatcher

\[ e_0 \ e_1 \ e_2 \ \boxed{e_3} \]

\( e_4 \)
Shared Event Loop

FIFO queue

Dispatcher

\[ \text{e_0, e_1, e_2, e_3, e_4} \]

time
Shared Event Loop

FIFO queue

Dispatcher

Shared Event Loop diagram with time intervals and events.
Shared Event Loop

FIFO queue

Event-delay trace

d_0  d_1  d_2  d_3

e_0  e_1  e_2  e_3  e_4

time
SYSTEM/INTERNET

HOST PROCESS

- NETWORK REQUESTS
- IPC COMMUNICATION
- DISPATCHES USER ACTIONS
Spying on the Host

```javascript
function loop () {
    save(performance.now());
    fetch(new Request("http://0/"))
        .catch(loop);
}
loop();
</script>

Timing resolution of ~500 μs
Spying on the Host

```html
<script>
function loop () {
    save(performance.now());
    fetch(new Request("http://0/"))
        .catch(loop);
}
loop();
</script>

Timing resolution of ~500 μs

With some smarter techniques we obtain <100 μs
(see the paper)
HOST PROCESS

SYSTEM/INTERNET

RENDERER 1

tab1 | trusted.com

• JAVASCRIPT EXECUTION
• RESOURCE PARSING
• LAYOUT & RENDERING
HOST PROCESS

SYSTEM/INTERNET

RENDERER 1

tab1 | trusted.com

iframe |

SHARED BETWEEN IFRAMES, POPUPS, MAX #RENDERER EXCEEDED…
Spying on the Renderer

```javascript
function loop() {
    save(performance.now());
    self.postMessage(0, "*");
}
self.onmessage = loop;
loop();
</script>

Timing resolution of <25 μs
<table>
<thead>
<tr>
<th>Event Type</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>μ-arch events</td>
<td>&lt;5μs</td>
</tr>
<tr>
<td>loop()</td>
<td>25 μs</td>
</tr>
<tr>
<td>Mouse movement</td>
<td>100 μs</td>
</tr>
<tr>
<td>GC scavenging</td>
<td>&lt;1 ms</td>
</tr>
<tr>
<td>JS event handlers</td>
<td>&gt;2 ms</td>
</tr>
</tbody>
</table>
Duration of Events

μ-arch events
loop()
Mouse movement
GC scavenge
JS event handlers

<5μs  25 μs  100 μs  <1 ms  >2 ms

Responsive code is harder to identify
LoopScan Tool

https://github.com/cgvwzq/loopscan
Web Page Identification
& Inter-keystroke Timing
Web Page Identification

Monitor the EventLoop while page loading
Dynamic Time Warping

DTW is resistant to delays in the occurrence of events
Dynamic Time Warping

DTW is resistant to delays in the occurrence of events 2-4 seconds of measuring
Dynamic Time Warping

DTW is resistant to delays in the occurrence of events.

2-4 seconds of measuring

One trace for training
Web Page Identification

500 pages x 30 traces x 3 machines x 2 event loops

<table>
<thead>
<tr>
<th>Renderer’s main thread:</th>
<th>75%</th>
<th>(Linux desktop)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host’s I/O thread:</td>
<td>23%</td>
<td>(Macbook Pro)</td>
</tr>
</tbody>
</table>

(recognition rates below 5% across machines)

R-library and datasets:  
https://github.com/cgvwzq/rlang-loophole
Inter-keystroke Timing

We obtain the **password length** and **time between consecutive pressed keys**
Inter-keystroke Timing

10,000 passwords
90% accuracy
precision: $\sigma = 6.1$ ms
Inter-keystroke Timing

10,000 passwords
90% accuracy
precision: $\sigma = 6.1$ ms

More precision than network based attacks.

Less noise than in micro-architectural attacks.

No privileges. No training.
Countermeasures

- Reduce clock resolution
- Site Isolation Project
- CPU throttling
- Rate limiting
Countermeasures

• Reduce clock resolution
• Site Isolation Project
• CPU throttling
• Rate limiting
Conclusions

• Shared event loops in Chrome are vulnerable to timing side-channels

• We systematically study how this channel can be used for different attacks

• Fundamental design issues that need to be addressed
Thank you! :)  
Questions?