JavaScript Zero

Real JavaScript and Zero Side-Channel Attacks

Michael Schwarz, Moritz Lipp, Daniel Gruss
20.02.2018

www.iaik.tugraz.at
• Analysis of current microarchitectural and side-channel attacks
• Identifying building blocks for attacks
• Countermeasures for preventing attacks
• Implementation of countermeasures
• Evaluation of countermeasures
Currently 11 microarchitectural and side-channel attacks in JavaScript
Landscape of Microarchitectural Attacks

- Currently 11 microarchitectural and side-channel attacks in JavaScript
- Analyse requirements for every attack
Currently 11 microarchitectural and side-channel attacks in JavaScript

- Analyse requirements for every attack
- Results in 5 categories
Currently 11 microarchitectural and side-channel attacks in JavaScript

Analyse requirements for every attack

Results in 5 categories

- Memory addresses
- Accurate timing
- Multithreading
- Shared data
- Sensor API
Currently 11 microarchitectural and side-channel attacks in JavaScript

- Analyse requirements for every attack
- Results in 5 categories
  - Memory addresses
  - Accurate timing
  - Multithreading
  - Shared data
  - Sensor API

- Every attack is in at least one category
• Language does not provide addresses to programmer
• Language does not provide addresses to programmer
• Closest to virtual address: array indices
• Language does not provide *addresses* to programmer
• Closest to virtual address: *array* indices
• Detect beginning of physical pages through high timing on page faults
Nearly all attacks require accurate timing
Nearly all attacks require accurate timing.
No absolute timestamps required, only time differences.
• Nearly all attacks require **accurate timing**
• No absolute timestamps required, only **time differences**
• Required accuracy varies between milliseconds and nanoseconds
• JavaScript introduced multi threading with web workers
• JavaScript introduced multi threading with web workers
• Real concurrency in applications
• JavaScript introduced **multi threading** with web workers
• Real concurrency in applications
• Enables new side-channel attacks
- Usually no shared data between threads due to synchronization issues
• Usually no shared data between threads due to synchronization issues
• Exception: `SharedArrayBuffer`
• Usually no shared data between threads due to synchronization issues
• Exception: SharedArrayBuffer
• Only useful in combination with web workers
• Usually no shared data between threads due to synchronization issues
• Exception: SharedArrayBuffer
• Only useful in combination with web workers
• Not enabled by default
• Some side-channel attacks only require access to sensors
• Some side-channel attacks only require access to sensors
• Some sensors can be used without user consent, e.g., ambient light
• Some side-channel attacks only require access to sensors
• Some sensors can be used without user consent, e.g., ambient light
• Every sensor is exploitable
Defenses
• Countermeasures have to address all categories
• Countermeasures have to address all categories
• Should not be visible to the programmer
Countermeasures

- Countermeasures have to address **all categories**
- Should not be visible to the programmer
- Implementation is on the “microarchitectural” level of JavaScript
Countermeasures

- Countermeasures have to address all categories
- Should not be visible to the programmer
- Implementation is on the “microarchitectural” level of JavaScript
- If no category is usable for attacks anymore, future attacks are hard
• Ensure arrays are memory backed and not linear
• Ensure arrays are memory backed and not linear
• Additionally, add random dummy accesses
• Ensure arrays are memory backed and not linear
• Additionally, add random dummy accesses
• Prevents many microarchitectural attacks
• Ensure arrays are memory backed and not linear
• Additionally, add random dummy accesses
• Prevents many microarchitectural attacks
• Side effect: make exploits harder where addresses are required
• Reducing the resolution of `performance.now()` is a first step
• Reducing the resolution of `performance.now()` is a first step
• Only rounding the timestamps is not sufficient
Accurate Timing

- Reducing the resolution of `performance.now()` is a first step
- Only rounding the timestamps is not sufficient
- Fuzzy time (Vattikonda et al.) adds random jitter
• Reducing the resolution of `performance.now()` is a first step
• Only rounding the timestamps is not sufficient
• Fuzzy time (Vattikonda et al.) adds random jitter
• Timestamps are still monotonic, but clock edges are randomized
• Only real solution is to prevent multithreading
• Only real solution is to prevent multithreading
• Some attacks can be prevented by adding random delays to postMessage
Multithreading

- Only real solution is to prevent multithreading
- Some attacks can be prevented by adding random delays to `postMessage`
- Prevents certain timing primitives and attacks on the event-queue latency
• Best countermeasures: do not allow shared data
Best countermeasures: do not allow shared data

Many attacks exploit fast concurrent access to `SharedArrayBuffer`
• Best countermeasures: do not allow shared data
• Many attacks exploit fast concurrent access to `SharedArrayBuffer`
• Alternative: delay access to buffer
• Best countermeasures: do not allow shared data
• Many attacks exploit fast concurrent access to `SharedArrayBuffer`
• Alternative: delay access to buffer
• Degrades resolution of timing primitive to microseconds
• Reduce resolution and update frequency of sensors
- Reduce resolution and update frequency of sensors
- Sensor APIs should always ask user for permission
• Reduce resolution and update frequency of sensors
• Sensor APIs should always ask user for permission
• Every sensor is usable for attacks, even ambient light sensor
• Reduce resolution and update frequency of sensors
• Sensor APIs should always ask user for permission
• Every sensor is usable for attacks, even ambient light sensor
• To not break existing applications, sensors return constant value
Implementation
• Best solution is to implement defenses in the browser core
• Best solution is to implement defenses in the browser core
• Maintaining a browser fork is hard work
- Best solution is to implement defenses in the browser core
- Maintaining a browser fork is hard work
- We want a generic solution for multiple browsers
• Best solution is to implement defenses in the browser core
• Maintaining a browser fork is hard work
• We want a generic solution for multiple browsers
• Parsing JavaScript is hard
Designing the Countermeasure

- Best solution is to implement defenses in the browser core
- Maintaining a browser fork is hard work
- We want a generic solution for multiple browsers
- Parsing JavaScript is hard
- Implementation in JavaScript → Virtual machine layering
Designing the Countermeasure

• Best solution is to implement defenses in the browser core
• Maintaining a browser fork is hard work
• We want a generic solution for multiple browsers
• Parsing JavaScript is hard
• Implementation in JavaScript → Virtual machine layering
• Proof-of-concept is implemented as browser extension
Functions and properties are replaced by wrappers.
• Functions can be re-defined in JavaScript

```javascript
var original_reference = window.performance.now;
window.performance.now = function() { return 0; }
```

// call the new function (via function name)
alert(window.performance.now());
// == alert(0)

// call the original function (only via reference)
alert(original_reference.call(window.performance));
- Functions can be re-defined in JavaScript

```javascript
var original_reference = window.performance.now;
window.performance.now = function() { return 0; };
// call the new function (via function name)
alert(window.performance.now()); // == alert(0)
```
• Functions can be re-defined in JavaScript

```javascript
var original_reference = window.performance.now;
window.performance.now = function() { return 0; };

// call the new function (via function name)
alert(window.performance.now()); // == alert(0)

// call the original function (only via reference)
alert(original_reference.call(window.performance));
```
• Functions can be **re-defined** in JavaScript

```javascript
var original_reference = window.performance.now;
window.performance.now = function() { return 0; };

// call the new function (via function name)
alert(window.performance.now()); // == alert(0)

// call the original function (only via reference)
alert(original_reference.call(window.performance));
```

• Properties can be replaced by **accessor properties**
Virtual Machine Layering for Objects

- Objects are proxied

```javascript
new Object
Proxy(Object)
```

- All properties and functions are handled by the original object
- Functions and properties can be overwritten in the proxy object
Virtual Machine Layering for Objects

- Objects are proxied

```
new Object
```

- All properties and functions are handled by the original object
• Objects are proxied

- All properties and functions are handled by the original object
- Functions and properties can be overwritten in the **proxy object**
• Attacker tries to circumvent JavaScript Zero
Self Protection

- Attacker tries to circumvent JavaScript Zero
- Self protection is necessary if implemented in JavaScript
• Attacker tries to circumvent JavaScript Zero
• **Self protection** is necessary if implemented in JavaScript
• Use closures to hide all references to original functions

```javascript
(function () {
  // original is only accessible in this scope
  var original = window.performance.now;
  window.performance.now = ...
})(());
```
• Attacker tries to circumvent JavaScript Zero
• **Self protection** is necessary if implemented in JavaScript
• Use closures to hide all references to original functions

```javascript
(function() {
    // original is only accessible in this scope
    var original = window.performance.now;
    window.performance.now = ...
})();
```

• Prevent objects from being modified: `Object.freeze`
Evaluation
• Border of pages leak 12 or 21 bits (depending on page size)
• Border of pages leak 12 or 21 bits (depending on page size)
• Create huge array
• Border of pages leak 12 or 21 bits (depending on page size)
• Create huge array
• Iterate over array, measure access time
Border of pages leak 12 or 21 bits (depending on page size)
Create huge array
Iterate over array, measure access time
Page border raise pagefault, taking significantly longer to access
Array offset [KB]

Access time [cycles]

$10^5$

$10^4$
Page Border Detection with Random Access

Access time [cycles] vs. Array offset [KB]

Access time ranges from 0 to \(10^5\) cycles.

Michael Schwarz, Moritz Lipp, Daniel Gruss — www.iaik.tugraz.at
• Multithreading allows to detect interrupts
• Multithreading allows to detect interrupts
• Endless loop which counts number of increments in time window
• Multithreading allows to detect interrupts
• Endless loop which counts number of increments in time window
• Different number of increments indicate interrupt
• Multithreading allows to detect **interrupts**
• Endless loop which counts number of increments in time window
• Different number of increments indicate interrupt
• Fuzzy time prevents deterministic equally-sized time window
• Messages between web workers are handled in the event queue
• Messages between web workers are handled in the event queue
• User activity is also handled in the event queue
• Messages between web workers are handled in the event queue
• User activity is also handled in the event queue
• Posting many messages allows to measure latency
- Messages between web workers are handled in the event queue
- User activity is also handled in the event queue
- Posting many messages allows to measure latency
- Latency indicates user input
Event Queue Spying

![Graph showing runtime and delta versus time]

Michael Schwarz, Moritz Lipp, Daniel Gruss — www.iaik.tugraz.at
• `SharedArrayBuffer` allows to build a timing primitive with the highest resolution
• **SharedArrayBuffer** allows to build a timing primitive with the highest resolution

• One web worker continuously increments variable in the shared array
• **SharedArrayBuffer** allows to build a timing primitive with the highest resolution
• One web worker continuously increments variable in the shared array
• Other worker uses this as a timestamp
• SharedArrayBuffer allows to build a timing primitive with the highest resolution
• One web worker continuously increments variable in the shared array
• Other worker uses this as a timestamp
• Adding random delay to access degrades resolution
SharedArrayBuffer with Random Delay

![Chart showing cache hit and cache miss over time](chart.png)

- Number of cases
- Access time [buffer increments]

**Cache Hit**

**Cache Miss**
<table>
<thead>
<tr>
<th>Defense</th>
<th>Prevents</th>
<th>Rowhammer.js</th>
<th>Page Duplication</th>
<th>DRAM Covert Channel</th>
<th>Anti-ASLR</th>
<th>Cache Eviction</th>
<th>Keystore Timing</th>
<th>Browser</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer ASLR</td>
<td></td>
<td></td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Array preloading</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Non-deterministic array</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Array index randomization</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Low-resolution timestamp</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Fuzzy time</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>◊*</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>WebWorker polyfill</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Message delay</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Slow SharedArrayBuffer</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>No SharedArrayBuffer</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

Symbols indicate whether a policy fully prevents an attack, (●), partly prevents and attack by making it more difficult (○), or does not prevent an attack (○). A star (*) indicates that all policies marked with a star must be combined to prevent an attack.
User Experience

Top 25 Alexa domains

- Active policies
- No active policies
• Just rounding timers is not sufficient
• Just rounding timers is not sufficient
• Multithreading and shared data allow to build new timers
• Just rounding timers is not sufficient
• Multithreading and shared data allow to build new timers
• Microarchitectural attacks in the browser are possible at the moment
• Just rounding timers is not sufficient
• Multithreading and shared data allow to build new timers
• Microarchitectural attacks in the browser are possible at the moment
• Efficient countermeasures can be implemented in browsers
• Just rounding timers is not sufficient
• Multithreading and shared data allow to build new timers
• Microarchitectural attacks in the browser are possible at the moment
• Efficient countermeasures can be implemented in browsers
• More microarchitectural attacks in JavaScript will appear