GLitch chronicles: turning WebGL into a hammer

Pietro Frigo
Cristiano Giuffrida, Herbert Bos, Kaveh Razavi
GLitch: what?

First **Rowhammer** exploit from **JavaScript** on **mobile**

Fastest **JS-based** Rowhammer exploit

First **GPU-accelerated** bit flip
The chronicles of GLitch
GLitch: the chronicles

ME!
GLitch: the chronicles

\[ \text{ME!} \]

\[ 14-15-16-17 \]

Cristiano

DRAM

ROWHAMMER

BIT FLIPS
GLitch: the chronicles

ROWHAMMER

DRAM

BIT FLIPS

CRISTIANO
DRAM

DIMM
DRAM

Row 0
Row 1
Row 2

... 

Row $n$

Row buffer
DRAM

Row 0
Row 1
Row 2

... Row n ...

Row buffer

CELL
DRAM

Row 0
Row 1
Row 2

\[ \vdots \]

Row \( n \)

Row buffer

Capacitors leak charges ==> refresh
DRAM

Row 0
Row 1
Row 2
... 
Row n

Row buffer

CELL

0 1 1 1 0 0 0 1 0 1 0 1
DRAM

Row 0
Row 1
Row 2
Row n
Row buffer

Activate
DRAM

Row 0
Row 1
Row 2
Row n
Row buffer

Activate
DRAM

Row 0
Row 1
Row 2
Row \( n \)
Row buffer

Precharge
Rowhammer

Row 0
Row 1
Row 2
\[\cdots\]
Row \(n\)
Row buffer

Aggressor row
Victim row
Aggressor row
Rowhammer

Row 0
Row 1
Row 2
\ldots
Row n
Row buffer

\begin{tabular}{cccccccc}
0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\end{tabular}

Aggressor row
Victim row
Aggressor row

Row buffer
Rowhammer

Row 0
Row 1
Row 2
Row n
Row buffer

Aggressor row
Victim row
Aggressor row
Rowhammer

Row 0
Row 1
Row 2
... Row n
Row buffer

Aggressor row
Victim row
Aggressor row
Rowhammer

Row 0
Row 1
Row 2
Row n
Row buffer

Aggressor row
Victim row
Aggressor row
Rowhammer

Row 0
Row 1
Row 2

Row n

Row buffer

Aggressor row
Victim row
Aggressor row
Rowhammer

Row 0
Row 1
Row 2
Row n
Row buffer

Aggressor row
Victim row
Aggressor row
Rowhammer

Row 0
Row 1
Row 2
Row n
Row buffer

Aggressor row
Victim row
Aggressor row
Rowhammer

Row 0
Row 1
Row 2
Row n
Row buffer

Reproducible!
GLitch: the chronicles

[1] Flipping bits in memory without accessing them
GLitch: the chronicles

[1] Flipping bits in memory without accessing them
GLitch: the chronicles

[1] Flipping bits in memory without accessing them
GLitch: the chronicles

[1] Flipping bits in memory without accessing them
GLitch: the chronicles

[1] Flipping bits in memory without accessing them

CAN YOU HAMMER YOUR PHONE?
GLitch: the chronicles

[1] Flipping bits in memory without accessing them
[5] Flip Feng Shui: Breaking the **cloud**
Drammer: Flip Feng Shui Goes Mobile - VUSec

Drammer is the first instance of mobile Rowhammer and comprehends a deterministic Android root exploit that does not rely on any software vulnerability.

vusec.net

Victor van der Veen @vvdveen · 25 ott 2016
I wouldn't be surprised if we could pull this one from a browser actually...

Traduci dalla lingua originale: inglese

the grugq
@thegrugq

In risposta a @vvdveen e @vu5ec
love to see it happen. :)

Following
CHALLENGE ACCEPTED
Attacker primitives
Attacker primitives
Attacker primitives

#P1. Fast memory access
Attacker primitives

#P1. Fast memory access
Attacker primitives

#P1. Fast memory access
Attacker primitives

#P1. Fast memory access
Attacker primitives

#P1. Fast memory access
Attacker primitives

#P1. Fast memory access
Attacker primitives

#P1. Fast memory access
Attacker primitives

#P1. Fast memory access
Attacker primitives

#P1. Fast memory access
  - clflush (native)
Attacker primitives

#P1. Fast memory access
   - clflush (native)
Attacker primitives

#P1. Fast memory access
- clflush (native)
Attacker primitives

#P1. Fast memory access
- clflush (native)
Attacker primitives

#P1. Fast memory access
- clflush (native)
Attacker primitives

#P1. Fast memory access
- clflush (native)
Attacker primitives

#P1. Fast memory access
- clflush (native)
- eviction sets (JS)
Attacker primitives

#P1. Fast memory access

- `clflush` (native)
- eviction sets (JS)
Attacker primitives

#P1. Fast memory access
- clflush (native)
- eviction sets (JS)
Attacker primitives

#P1. Fast memory access
- clflush (native)
- eviction sets (JS)
Attacker primitives

#P1. Fast memory access

- clflush (native)
- eviction sets (JS)
Attacker primitives

#P1. Fast memory access

- clflush (native)
- eviction sets (JS)
Attacker primitives

#P1. Fast memory access

- clflush (native)
- eviction sets (JS)
Attacker primitives

#P1. Fast memory access
- clflush (native)
- eviction sets (JS)
Attacker primitives

#P1. Fast memory access
- clflush (native)
- eviction sets (JS)
Attacker primitives

#P1. Fast memory access

- clflush (native)
- eviction sets (JS)
Attacker primitives

#P1. Fast memory access
   - clflush (native)
   - eviction sets (JS)

#P2. Contiguous memory
Address translation

Virtual Memory

```c
char* buffer = malloc(sizeof(char)*KB(20));
```
Address translation

Virtual Memory

\[ \text{char* buffer} = \text{malloc} \left( \text{sizeof(char)} \times \text{KB(20)} \right); \]
Address translation

Virtual Memory

Physical Memory
Address translation

Virtual Memory          Physical Memory
Address translation

Virtual Memory -> Physical Memory
DRAM: organization
DRAM: organization
DRAM: organization

[Diagram showing the organization of DIMMs connected to Channels]
DRAM: organization

Ranks

DIMMs

Channels
DRAM: organization
DRAM: organization
Address translation: THPs

Virtual Memory

char* buffer = malloc(sizeof(char)*MB(3));
Address translation: THPs

Virtual Memory

```
char* buffer = malloc(sizeof(char)*MB(3));
```
Address translation: THPs
Address translation: THPs

Virtual Memory

Physical Memory
Address translation: THPs
Attacker primitives

#P1. Fast memory access
   - clflush (native)
   - eviction sets (JS)

#P2. Contiguous memory
   - THPs (native and JS)
Attacker primitives

#P1. Fast memory access
- clflush (native)
  - eviction sets (JS)

#P2. Contiguous memory
- THPs (native and JS)
Attacker primitives

#P1. Fast memory access
- clflush (native)
- eviction sets (JS)

#P2. Contiguous memory
- THPs (native and JS)
#P2. Eviction-based Rowhammer: arm

Caches: Large & random
#P2. Eviction-based Rowhammer: \textit{arm}

Caches: Large & random

Steps:

1. Read row \textit{n-1}
#P2. Eviction-based Rowhammer: \textit{arm}

Caches: Large & random

Steps:
1. Read row $n-1$
2. Read row $n+1$
#P2. Eviction-based Rowhammer: \textit{arm}

Caches: Large & random

Steps:

1. Read row \textit{n-1}
2. Read row \textit{n+1}
3. Evict++
#P2. Eviction-based Rowhammer: arm

Caches: Large & random

Steps:
1. Read row $n-1$
2. Read row $n+1$
3. Evict++
Eviction-based Rowhammer: \texttt{arm}

Caches: Large & random

Steps:
1. Read row $n-1$
2. Read row $n+1$
3. Evict++
#P2. Eviction-based Rowhammer: \textit{arm}

Caches: Large & random

Steps:

1. Read row \(n-1\)
2. Read row \(n+1\)
3. Evict++
#P2. Eviction-based Rowhammer: \textit{arm}

Caches: Large & random

Steps:

1. Read row \(n-1\)
2. Read row \(n+1\)
3. Evict++
#P2. Eviction-based Rowhammer: \textit{arm}

Caches: Large & random

Steps:

1. Read row $n-1$
2. Read row $n+1$
3. Evict++
#P2. Eviction-based Rowhammer: \textit{arm}

Caches: Large & random

Steps:

1. Read row \( n-1 \)
2. Read row \( n+1 \)
3. Evict++
4. Read row \( n-1 \)
   ...

Attacker primitives

#P1. Fast memory access
   ✖ clflush (native)
   ✖ eviction sets (JS)

#P2. Contiguous memory
   ✖ THPs (native and JS)
Attacker primitives

#P1. Fast memory access
- clflush (native)
- eviction sets (JS)

#P2. Contiguous memory
- THPs (native and JS)
Attacker primitives

#P1. Fast memory access
- clflush (native)
- eviction sets (JS)

#P2. Contiguous memory
- THPs (native and JS)

ION == DMA memory
Uncached & Contiguous
We are screwed.
WHAT IF YOU CHANGE ATTACK VECTOR?
Attack Vector

WHAT IF YOU CHANGE ATTACK VECTOR?
Attack Vector

WHAT IF YOU CHANGE ATTACK VECTOR?
Attack Vector

WHAT IF YOU CHANGE ATTACK VECTOR?
Attacker primitives

#P1. Fast memory access

#P2. Contiguous memory
Attacker primitives

#P1. DRAM Access

#P2. Fast memory access

#P3. Contiguous memory
Attacker primitives

#P1. DRAM Access
#P2. Fast memory access
#P3. Contiguous memory
Understanding the GPU
#P1. GPU: The rendering pipeline
#P1. GPU: The rendering pipeline
#P1. GPU: The rendering pipeline
#P1. GPU: The rendering pipeline

Input (CPU) → Vertex Shader → Fragment Shader → Output (Framebuffer)
#P1. GPU: The rendering pipeline
#P1. GPU: The rendering pipeline
#P1. GPU: The rendering pipeline

- **Input (CPU)**
- **Vertex Shader**
- **Fragment Shader**
- **Output (Framebuffer)**

![Diagram of the rendering pipeline with vertices and textures](image)
#P1. GPU: The rendering pipeline

Input (CPU) → Vertex Shader → Fragment Shader → Output (Framebuffer)

Texture
# P1. GPU: The rendering pipeline
#P1. GPU: The architecture
#P1. GPU: The architecture
#P1. GPU: The architecture
#P1. GPU: The architecture
#P1. GPU: The architecture
#P1. GPU: The architecture
#P1. GPU: The architecture
#P1. GPU: The architecture
#P1. DRAM access
#P1. DRAM access

1. Read Vertices
2. Read Textures
3. Write to Framebuffer
#P1. DRAM access

1. Read Vertices
2. Read Textures ==> most predictable
3. Write to Framebuffer
# P1. DRAM access: texture sampling

```c
uniform sampler2D tex;

void main() {
  vec2 coord = vec2(0,0);
  gl_FragColor = texture2D(tex, coord);
}
```
**#P1. DRAM access: texture sampling**

```cpp
uniform sampler2D tex;

void main() {
  vec2 coord = vec2(0, 0);
  gl_FragColor = texture2D(tex, coord);
  tex[coord]
}
```
Attacker primitives

#P1. DRAM access ✔

#P2. Fast memory access

#P3. Contiguous memory
Attacker primitives

#P1. DRAM access ✓
#P2. Fast memory access
#P3. Contiguous memory
#P2. Fast memory access
#P2. Fast memory access

```plaintext
uniform sampler2D tex;

void main() {
  vec2 coord = vec2(0,0);
  vec4 a = texture2D(tex, coord);
  vec4 b = texture2D(tex, coord);
}
```
#P2. Fast memory access

```cpp
uniform sampler2D tex;

void main() {
    vec2 coord = vec2(0,0);
    vec4 a = texture2D(tex, coord);
    vec4 b = texture2D(tex, coord);
}
```
#P2. Fast memory access

1. `uniform sampler2D tex;`
2. `void main() {`
3. `vec2 coord = vec2(0,0);`
4. `vec4 a = texture2D(tex, coord);`
5. `vec4 b = texture2D(tex, coord);`
6. `}`
#P2. Fast memory access

1. `uniform sampler2D tex;`
2. `void main() {`
3. `vec2 coord = vec2(0,0);`
4. `vec4 a = texture2D(tex, coord);`
5. `vec4 b = texture2D(tex, coord);`
6. `}`
133

#P2. Fast memory access

```c
uniform sampler2D tex;

void main() {
    vec2 coord = vec2(0,0);
    vec4 a = texture2D(tex, coord);
    vec4 b = texture2D(tex, coord);
}
```
#P2. Fast memory access

```c
1 uniform sampler2D tex;
2
3 void main() {
4   vec2 coord = vec2(0,0);
5   vec4 a = texture2D(tex, coord);
6   vec4 b = texture2D(tex, coord);
7 }
```
#P2. Fast memory access

1 \texttt{uniform sampler2D \textit{tex};}
2
3 \texttt{void \textit{main}() \{}
4 \qquad \texttt{vec2 \textit{coord} = vec2(0,0);} \\
5 \qquad \texttt{vec4 \textit{a} = texture2D(\textit{tex}, \textit{coord});} \\
6 \qquad \texttt{vec4 \textit{b} = texture2D(\textit{tex}, \textit{coord});} \quad \rightarrow \\
7 \texttt{\}}
#P2. Fast cache eviction
#P2. Eviction-based Rowhammer: GPU

Caches: Small & Deterministic
#P2. Eviction-based Rowhammer: GPU

Caches: Small & Deterministic

Steps:

1. Read row n-1
#P2. Eviction-based Rowhammer: GPU

Caches: Small & Deterministic

Steps:

1. Read row \( n-1 \)
2. Read row \( n+1 \)
#P2. Eviction-based Rowhammer: GPU

Caches: Small & Deterministic

Steps:

1. Read row \( n-1 \)
2. Read row \( n+1 \)
3. Evict++
#P2. Eviction-based Rowhammer: GPU

Caches: Small & Deterministic

Steps:

1. Read row \( n-1 \)
2. Read row \( n+1 \)
3. Evict++
#P2. Eviction-based Rowhammer: GPU

Caches: Small & Deterministic

Steps:
1. Read row n-1
2. Read row n+1
3. Evict++
#P2. Eviction-based Rowhammer: GPU

Caches: Small & Deterministic

Steps:

1. Read row $n-1$
2. Read row $n+1$
3. Evict++
#P2. Eviction-based Rowhammer: GPU

Caches: Small & Deterministic

Steps:
1. Read row \( n-1 \)
2. Read row \( n+1 \)
3. Evict++
# P2. Eviction-based Rowhammer: GPU

Caches: Small & Deterministic

Steps:

1. Read row $n-1$
2. Read row $n+1$
3. Evict++
#P2. Eviction-based Rowhammer: GPU

Caches: Small & Deterministic

Steps:
1. Read row $n-1$
2. Read row $n+1$
3. Evict++
#P2. Eviction-based Rowhammer: GPU

Caches: Small & Deterministic

Steps:

1. Read row $n-1$
2. Read row $n+1$
3. Evict++
4. Read row $n-1$
#P2. Eviction-based Rowhammer: GPU

![Graph showing the relationship between the number of NOP instructions and the number of observed bit flips and time per read. The graph indicates a fast enough access time of approximately 180 ns.](image)
Attacker primitives

#P1. DRAM access ✔
#P2. Fast memory access
#P3. Contiguous memory
Attacker primitives

#P1. DRAM access ✔

#P2. Fast memory access ✔

#P3. Contiguous memory
Attacker primitives

#P1. DRAM access ✓
#P2. Fast memory access ✓
#P3. Contiguous memory
#P3. Memory Allocation
# P3. Memory Allocation

while (num_tex--) {
  // size 4KB
  tex[num_tex] = gl.createTexture();
  fill_tex(tex[num_tex])
}
while (num_tex--) {
    // size 4KB
    tex[num_tex] = gl.createTexture();
    fill_tex(tex[num_tex])
}

#P3. Memory Allocation
#P3. Memory Allocation

```cpp
while (num_tex--) {
    // size 4KB
    tex[num_tex] = gl.createTexture();
    fill_tex(tex[num_tex])
}
```
# P3. Memory Allocation

```cpp
while (num_tex--) {
    // size 4KB
    tex[num_tex] = gl.createTexture();
    fill_tex(tex[num_tex])
}
```
#P3. Memory Allocation

```cpp
while (num_tex--) {
    // size 4KB
    tex[num_tex] = gl.createTexture();
    fill_tex(tex[num_tex]);
}
```
#P3. Memory Allocation

```cpp
while (num_tex--) {
    // size 4KB
    tex[num_tex] = gl.createTexture();
    fill_tex(tex[num_tex])
}
```
#P3. Memory Allocation

```javascript
while (num_tex--) {
    // size 4KB
    tex[num_tex] = gl.createTexture();
    fill_tex(tex[num_tex])
}
```
#P3. Memory Allocation

```cpp
while (num_tex--) {
    // size 4KB
    tex[num_tex] = gl.createTexture();
    fill_tex(tex[num_tex])
}
```
#P3. Memory Allocation

```c
while (num_tex--) {
    // size 4KB
    tex[num_tex] = gl.createTexture();
    fill_tex(tex[num_tex])
}
```
#P3. Memory Allocation

```cpp
while (num_tex--) {
    // size 4KB
    tex[num_tex] = gl.createTexture();
    fill_tex(tex[num_tex])
}
```
#P3. Memory Allocation

```java
while (num_tex--) {
    // size 4KB
    tex[num_tex] = gl.createTexture();
    fill_tex(tex[num_tex])
}
```
#P3. Memory Allocation

```java
while (num_tex--) {
    // size 4KB
    tex[num_tex] = gl.createTexture();
    fill_tex(tex[num_tex])
}
```
#P3. Memory Allocation

```cpp
while (num_tex--) {
    // size 4KB
    tex[num_tex] = gl.createTexture();
    fill_tex(tex[num_tex])
}
```
#P3. Memory Allocation

```java
while (num_tex--) {
    // size 4KB
    tex[num_tex] = gl.createTexture();
    fill_tex(tex[num_tex])
}
```
#P3. Memory Allocation

**Hammerable**

**Non hammerable**

**How do we discern them?**
SIDE CHANNELS
SIDE CHANNELS EVERYWHERE
#P3. DRAM Reads: recap

Row buffer
#P3. DRAM Reads: recap

Row buffer
#P3. DRAM Reads: recap

Row buffer
#P3. DRAM Reads: recap

Row buffer
#P3. DRAM Reads: recap
#P3. DRAM Reads: recap

Row buffer
#P3. Contiguous Memory: Detection

Row buffer
#P3. Contiguous Memory: Detection

Row buffer
#P3. Contiguous Memory: Detection

Row buffer
#P3. Contiguous Memory: Detection

Row buffer
#P3. Contiguous Memory: Detection

Row buffer
#P3. Contiguous Memory: Detection

Row buffer
#P3. Contiguous Memory: Detection
#P3. Contiguous Memory: Detection
#P3. Contiguous Memory: Detection

![Diagram of contiguous memory with a row buffer highlighted.](image-url)
#P3. Contiguous Memory: Detection

Row buffer
#P3. Contiguous Memory: Detection

Row buffer
#P3. Contiguous Memory: Detection
#P3. Contiguous Memory: Detection
#P3. Contiguous Memory: Detection

[Diagram of memory layout with row buffer highlighted]
#P3. Contiguous Memory: Detection
#P3. Contiguous Memory: Detection
#P3. Contiguous Memory: Detection
#P3. Contiguous Memory: Detection
#P3. Contiguous Memory: Detection

Row buffer
#P3. Contiguous Memory: Detection

Row buffer
#P3. Contiguous Memory: Detection

Row buffer

Slow & Noisy
#P3. Contiguous Memory: Detection
#P3. Contiguous Memory: Detection

Row buffer
#P3. Contiguous Memory: Detection

Row buffer
#P3. Contiguous Memory: Detection

Row buffer
#P3. Contiguous Memory: Detection

Row buffer
#P3. Contiguous Memory: Detection

Row buffer
#P3. Contiguous Memory: Detection

Row buffer
#P3. Contiguous Memory: Detection

Row buffer
#P3. Contiguous Memory: Detection

Row buffer
#P3. Contiguous Memory: Detection

![Diagram showing contiguous memory detection]
#P3. Contiguous Memory: Detection
#P3. Contiguous Memory: Detection

Row buffer
#P3. Contiguous Memory: Detection
# P3. WebGL-based timers

**EXT_DISJOINT_TIMER_QUERY**  
(Extension - Explicit)

- Similar to `clock_gettime()`
- High resolution

**WebGLSync**  
(WebGL2 - Implicit)

- Sync CPU and GPU
- More coarse-grained
Attacker primitives

#P1. DRAM access ✓
#P2. Fast memory access ✓
#P3. Contiguous memory ✓
What do we do with these primitives?
GLitch
GLitch: in a nutshell

Flip feng shui in JS:

1. Memory templating
2. Memory massaging
3. Exploitation
GLitch: in a nutshell

Flip feng shui in JS:

1. Memory templating
2. Memory massaging
3. Exploitation
### Exploitation: JS Arrays

<table>
<thead>
<tr>
<th>1</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.878e+65</td>
<td>double</td>
</tr>
<tr>
<td>*obj</td>
<td>object</td>
</tr>
<tr>
<td>0.3</td>
<td>double</td>
</tr>
<tr>
<td>*str</td>
<td>string</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NaN</th>
<th>NaN-boxing</th>
</tr>
</thead>
</table>

1. `var arr = new Array(100);`
2. `arr[0] = 1 // int`
3. `arr[1] = 1.878e+65 // double`
4. `arr[3] = new Array(0x12) // object`
IEEE-754 floating point (double)

1.125 = 1125 * 10^(-3)

==

!=

11.25 = 1125 * 10^(-2)

↑

Significand

↑

exp
IEEE-754 floating point (double)

1.125 = 1125 * 10^(-3)

11.25 = 1125 * 10^(-2)
IEEE-754 floating point (double)

1.125 = 1125 \times 10^{(-3)}

11.25 = 1125 \times 10^{(-2)}

NaN
IEEE-754 floating point (double)

\[ 1.125 = 1125 \times 10^{(-3)} \]
\[ = 1125 \times 10^{(-2)} \]

\[ 11.25 = 1125 \times 10^{(-2)} \]

WHAT IF WE STORE POINTERS?

2\(^{53}-1\) unused values
Exploitation: NuN-boxing (32bit)

\[
a = (\text{double}) \ 0x7fffff8c9a8b7c4d
\]

\[
b = (\text{double}) \ 0xffffffff8c9a8b7c4d
\]

64 bits
Exploitation: NuN-boxing (32bit)

\[ a = \text{(double)} \, 0x7ffffff8c9a8b7c4d \]
\[ b = \text{(double)} \, 0xffffffff8c9a8b7c4d \]

64 bits

(tag < 0xffffffff80)

true

double

false

*object
Exploitation: NuN-boxing (32bit)

arr[1] = 0x7fffffff8c9a8b7c4d
arr[2] = 0xffffffff8c9a8b7c4d
Exploitation: NuN-boxing (32bit)

\[
\text{arr[1]} = 0x7ffffff8c9a8b7c4d \\
\text{arr[2]} = 0xffffffff8cf9a8b7c4d
\]

Same payload, 1-bit difference in tag.
Exploitation: Type Flipping

arr[1] = 0xffffffff8c9a8b7c4d
arr[2] = 0xffffffff8c9a8b7c4d
Exploitation: Type Flipping

\[ arr[1] = 0x7fffffff8c9a8b7c4d \]
\[ arr[2] = 0x7fffffff8c9a8b7c4d \]
Exploitation: Type Flipping

```plaintext
arr[1] = 0x7fffffff8c9a8b7c4d
arr[2] = 0x7fffffff8c9a8b7c4d
```

Diagram showing the exploitation of type flipping with variables and pointers.
Exploitation: Type Flipping

\[ \text{arr}[1] = 0x\text{fffffff}8c9a8b7c4d \]
\[ \text{arr}[2] = 0x7\text{fffffff}8c9a8b7c4d \]
Exploitation: Type Flipping

\[
\text{arr}[1] = 0xfffffff8c9a8b7c4d \\
\text{arr}[2] = 0x7fffff8c9a8b7c4d
\]
Exploitation: Type Flipping

\[
\text{arr}[1] = 0xffffffff8c9a8b7c4d
\]

\[
\text{arr}[2] = 0x7ffffffff8c9a8b7c4d
\]

2 Primitives:

- #1 Arbitrary Leak [1-to-0]
- #2 Arbitrary Craft [0-to-1]
Exploitation: Arbitrary R/W

Virtual Memory
Exploitation: Arbitrary R/W

Virtual Memory
Exploitation: Arbitrary R/W

Virtual Memory
Exploitation: Arbitrary R/W

Virtual Memory

var buff = new ArrayBuffer(100);
Exploitation: Arbitrary R/W

Virtual Memory

var buff = new ArrayBuffer(100);
Exploitation: Arbitrary R/W

Virtual Memory

```
var buff = new ArrayBuffer(100);
val = buff[108];  // ERROR OoB!
```
Exploitation: Arbitrary R/W

```javascript
var buff = new ArrayBuffer(100);
val = buff[108]; // ERROR OoB!

buff[K] = create_fake_buff();
var fake_buff = *buff[K];
```
Exploitation: Arbitrary R/W

Virtual Memory

buff[K] = create_fake_buff();
var fake_buff = *buff[K];

Challenges:
Exploitation: Arbitrary R/W

Virtual Memory

<table>
<thead>
<tr>
<th>*data, size</th>
<th>unknown_fields</th>
</tr>
</thead>
</table>

\[
\text{buff}[K] = \text{create\_fake\_buff}();
\]

\[
\text{var} \ \text{fake\_buff} = \text{*buff}[K];
\]

Challenges:
- unknown header fields (e.g., GC root)
Exploitation: Arbitrary R/W

Virtual Memory

<table>
<thead>
<tr>
<th>*data, size</th>
<th>unkown_fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>*data, size</td>
<td>unkown_fields</td>
</tr>
</tbody>
</table>

```
buff[K] = create_fake_buff();

var fake_buff = *buff[K];
```

Challenges:
- unknown header fields (e.g., GC root)
Exploitation: Arbitrary R/W

Virtual Memory

Known!

off?

*data, size
unknown_fields

*data, size
unknown_fields

*data, size
unknown_fields

*data, size
unknown_fields

buf[K] = create_fake_buff();

var fake_buff = *buf[K];

Challenges:
- unknown header fields (e.g., GC root)
- unknown data location
Exploitation: Arbitrary R/W

Virtual Memory

```
buff[K] = create_fake_buff();
var fake_buff = *buff[K];
```

Challenges:
- unknown header fields (e.g., GC root)
  ✓ unknown data location

Inline ArrayBuffer
Exploitation: Arbitrary R/W

Virtual Memory

\[ \text{buff}[K] = \text{create\_fake\_buff}(); \]
\[ \text{var fake\_buff} = *\text{buff}[K]; \]

Challenges:
- unknown header fields (e.g., GC root)
✓ unknown data location

Inline ArrayBuffers
Exploitation: Arbitrary read

class JSString => UTF-16 strings [0x0000 – 0xffff]
{
  uint32_t flags; // type of string
  uint32_t length; // sizeof(buff_header) 0x30
  char16_t* string;
  *buff_header
}
Exploitation: Arbitrary R/W

Virtual Memory

<table>
<thead>
<tr>
<th>*data, size</th>
<th>unknown_fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte[]</td>
<td></td>
</tr>
</tbody>
</table>

```
buff[0] = create_fake_buff();
var fake_buff = *buff[0];
```

Challenges:

- unknown header fields (e.g., GC root)
  - unknown data location
Exploitation: Arbitrary R/W

Virtual Memory

- *data, size
- unknown_fields
- flag, len, *str

```
buff[0] = create_fake_buff();
var fake_buff = *buff[0];

Challenges:
- unknown header fields (e.g., GC root)
  ✓ unknown data location
```
Exploitation: Arbitrary R/W

Virtual Memory

- *data, size
- unknown_fields
- flag, len, *str

buff[0] = create_fake_buff();

var fake_buff = *buff[0];

Challenges:

- ✓ unknown header fields (e.g., GC root)
- ✓ unknown data location
Exploitation: Recap

Exploit in 3 stages:

1. Virtual Memory
   - header
   - byte[]
   - "secret" data

2. Create fake string (read only) to reference str

3. Create fake ArrayBuffer to reference fake_buff

4. "secret" data
Exploitation: Recap

Exploit in 3 stages:
1. Break ASLR
Exploitation: Recap

Exploit in 3 stages:
1. Break ASLR
2. Arbitrary read

Virtual Memory

0xdeadbeef
0x9a8b7c4d
*real_buff
0x9a8b7c50

header

str 0x9a8b7c4d
byte[]

“secret” data

0xdeadbeef
arr[]
Exploitation: Recap

Exploit in 3 stages:
1. Break ASLR
2. Arbitrary read
Exploitation: Recap

Exploit in 3 stages:
1. Break ASLR
2. Arbitrary read
3. Arbitrary write
Exploitation: Recap

Exploit in 3 stages:
1. Break ASLR
2. Arbitrary read
3. Arbitrary write
Exploitation: Recap

Exploit in 3 stages:
1. Break ASLR ==> 1-to-0
2. Arbitrary read
3. Arbitrary write \[0\text{-to-1}\]

RUNS IN ~116 s ON AVERAGE
Demo
Disclosure & Mitigations

- Disclosure process with the help of Dutch NCSC (CVE-2018-10229)
- Chrome & Firefox released partial mitigations against timers. (Will be reenabled soon ︵(ツ)︿)
Conclusions

- **First** Rowhammer attack from JS on mobile
- **GPU** as new attack vector
- Takeaway: Redefine the **threat model** (DSP, FPGAs, ...)

https://www.vusec.net/projects/glitch/