[Obfuscator] Renaming constant variable in class static block

```
class x { static { const x = { x } = 0 ; } }
```

According to the JavaScript specification, running a JavaScript program `class x { static { const x = { x } = 0 ; } }` is expected to result in `ReferenceError`, but the output of Obfuscator is expected to result in `TypeError`.

The output of Obfuscator is as follows.

```
class x{static{const _0x4ecb98={x}=0x0;}}
```

In summary, the bug has arised due to renaming constant variable in class static block.

1. Original code and obfuscated code both evaluates `{ x } = 0` and `{x}=0x0` to initalize `x` in `x = { x } = 0` and `_0x4ecb98` in `_0x4ecb98={x}=0x0`.
2. When resolving `x` in `{x}`, the original code resolves to `x` in `const x` and the obfuscated code resolves to `x` in `class x`.
3. The **ReferenceError** exception arises in original code because the algorithm tries to set binding for uninitialized `x`.
4. The **TypeError** exception arises in obfuscated code because the algorithm tries to change the value of an immuatble binding `x` (class name).

Below is a detailed explanation using ECMAScript Specification. The first section is about how `class x { static { const x = { x } = 0 ; } }` result in `ReferenceError`. The second section is about how the evaluation step of obfuscated code is different from original code and why the exceptions are different.

Evaluation of `class x { static { const x = { x } = 0 ; } }` is done by following algorithms.

Evaluation of `class x { static { const x = { x } = 0 ; } }`

#1. Evaluation of *ClassDeclaration* of ECMAScript Specification

ClassDeclaration : **class** *BindingIdentifier ClassTail*

The algorithm calls BindingClassDeclarationEvaluation of this *ClassDeclaration*.

15.7.16 Runtime Semantics: Evaluation
ClassDeclaration **:** `class` *BindingIdentifier ClassTail*

1. Perform ? BindingClassDeclarationEvaluation of this *ClassDeclaration*.
2. Return empty.

#2. Operation BindingClassDeclarationEvaluation of ECMAScript Specification

ClassDeclaration : **class** *BindingIdentifier ClassTail*

In step 2, the algorithm calls ClassDefinitionEvaluation of *ClassTail* with arguments className and className.

ClassDeclaration : `class` *BindingIdentifier ClassTail*

1. Let *className* be StringValue of *BindingIdentifier*.
2. Let *value* be ? ClassDefinitionEvaluation of *ClassTail* with arguments *className* and *className*.
3. Set *value*.[[SourceText]] to the source text matched by *ClassDeclaration*.
4. Let *env* be the running execution context's LexicalEnvironment.
5. Perform ? InitializeBoundName(*className*, *value*, *env*).
6. Return *value*.

#3. Operation ClassDefinitionEvaluation of ECMAScript Speicifcation

In step 2 and 3, the algorithm creates a new declarative environment called `classEnv` and creates immutable binding of `x`.

ClassTail : *ClassHeritage*$_{opt}$ `{` *ClassBody*$_{opt}$ `}`

1. Let *env* be the LexicalEnvironment of the running execution context.
2. Let *classEnv* be NewDeclarativeEnvironment(*env*).
3. If *classBinding* is not **undefined**, then
   a. Perform *classEnv*.CreateImmutableBinding(*classBinding*, **true**).

In step 27, the algorithm initializes the binding of `x` with `F`, which is the evaluation result of *ClassDefinition* of x.

27. If *classBinding* is not **undefined**, then
    a. Perform ! *classEnv*.InitializeBinding(*classBinding*, *F*).

In step 31-b, the algorithm calls `Call` with arguments elementRecord.[[BodyFunction]] and F, where elementRecord.[[BodyFunction]] represents the `const x = { x } = 0 ;` and F represents the evaluation result of *ClassDefinition* of x.

31. For each element *elementRecord* of *staticElements*, do
    a. If *elementRecord* is a ClassFieldDefinition Record, then
       i. Let *result* be Completion(DefineField(*F*, *elementRecord*)).
    b. Else,
       i. Assert: *elementRecord* is a ClassStaticBlockDefinition Record.
       ii. Let *result* be Completion(Call(*elementRecord*.[[BodyFunction]], *F*)).

#4. Operation Call of the ECMAScript Specification

In step 3, the algorithm calls `F.[[Call]](V, argumentsList)`, where *F* represents the `const x = { x } = 0 ;`, *V* represents the evaluation result of *ClassDefintion* of x, and *argumentsList* is an empty List.

1. If *argumentsList* is not present, set *argumentsList* to a new empty List.
2. If IsCallable(*F*) is **false**, throw a **TypeError** exception.
3. Return ? *F*.[[Call]](*V*, *argumentsList*).

#5. Operation FunctionObject.[[Call]] of the ECMAScript specification.

In step 6, the algorithm calls OrdinaryCallEvaluateBody(F, argumentList), where F represents the `const x = { x } = 0 ;` and *argumentsList* is an empty List.

10.2.1 [[Call]] (*thisArgument*, *argumentsList*)

The [[Call]] internal method of an ECMAScript function object *F* takes arguments *thisArgument* (an ECMAScript language value) and *argumentsList* (a List of ECMAScript language values) and returns either a normal completion containing an ECMAScript language value or an abrupt completion. It performs the following steps when called:

1. Let *callerContext* be the running execution context.
2. Let *calleeContext* be PrepareForOrdinaryCall(*F*, **undefined**).
3. Assert: *calleeContext* is now the running execution context.
4. If *F*.[[IsClassConstructor]] is **true**, then
   a. Let *error* be a newly created **TypeError** object.
   b. NOTE: *error* is created in *calleeContext* with *F*'s associated Realm Record.
   c. Remove *calleeContext* from the execution context stack and restore *callerContext* as the running execution context.
   d. Return ThrowCompletion(*error*).
5. Perform OrdinaryCallBindThis(*F*, *calleeContext*, *thisArgument*).
6. Let *result* be Completion(OrdinaryCallEvaluateBody(*F*, *argumentsList*)).

#6. Operation OrdinaryCallEvaluateBody of the ECMAScript specification.

The algorithm calls EvaluateBody of F.[[ECMAScriptCode]] with arguments F and argumentList, where F represents the `const x = { x } = 0 ;` and *argumentsList* is an empty List.

10.2.1.4 OrdinaryCallEvaluateBody (*F*, *argumentsList*)

The abstract operation OrdinaryCallEvaluateBody takes arguments *F* (a function object) and *argumentsList* (a List) and returns either a normal completion containing an ECMAScript language value or an abrupt completion. It performs the following steps when called:

1. Return ? EvaluateBody of *F*.[[ECMAScriptCode]] with arguments *F* and *argumentsList*.

#7. Operation ClassStaticBlockBody.EvaluateBody of the ECMAScript specification.

The algorithm calls EvaluateClassStaticBlockBody of ClassStaticBlockBody with argument functionObject, where functionObject represents the `const x = { x } = 0 ;`.

ClassStaticBlockBody **:** *ClassStaticBlockStatementList*

1. Assert: *argumentsList* is empty.
2. Return ? EvaluateClassStaticBlockBody of *ClassStaticBlockBody* with argument *functionObject*.

#8. Operation ClassStaticBlockBody.EvaluateClassStaticBlockBody of the ECMAScript specification.

In step 1, the algorithm calls `FunctionDeclarationInstantiation(functionObject, <<>>)`, where functionObject represents the `const x = { x } = 0 ;`. This step creates binding of `x`, which represents the constant variable name `const x`, which is **not initialized** at this point.

In step 2, the algorithm calls the evaluation of *ClassStaticBlockStatementList*, where *ClassStaticBlockStatementList* represents `const x = { x } = 0 ;`

15.7.12 Runtime Semantics: EvaluateClassStaticBlockBody

The syntax-directed operation EvaluateClassStaticBlockBody takes argument *functionObject* and returns either a normal completion containing an ECMAScript language value or an abrupt completion. It is defined piecewise over the following productions:

ClassStaticBlockBody : *ClassStaticBlockStatementList*

1. Perform ? FunctionDeclarationInstantiation(*functionObject*, « »).
2. Return the result of evaluating *ClassStaticBlockStatementList*.

#9. Evaluation of *LexicalBinding* in ECMAScript specification.

LexicalBinding represents `x = {x} = 0`, and it can be divided into *BindingIdentifier* `x` and *Initializer* `= {x} = 0`.

This algorithm intends to resolve *bindingId* and evaluate *Initializer* to Initialize Binding (in step 5).

In step 4, the algorithm calls the evaluation of *Initializer*.

LexicalBinding : *BindingIdentifier* *Initializer*

1. Let *bindingId* be StringValue of *BindingIdentifier*.
2. Let *lhs* be Completion(ResolveBinding(*bindingId*)).
3. If IsAnonymousFunctionDefinition(*Initializer*) is **true**, then
   a. Let *value* be ? NamedEvaluation of *Initializer* with argument *bindingId*.
4. Else,
   a. Let *rhs* be the result of evaluating *Initializer*.
   b. Let *value* be ? GetValue(*rhs*).
5. Perform ? InitializeReferencedBinding(*lhs*, *value*).
6. Return empty.

#10. Evaluation of *AssignmentExpression* in ECMAScript specification.

AssignementExpression represents `{x} = 0`, and it can be divided into *LeftHandSideExpression* `{x}` and *AssignmentExpression* `0`.

In step 5, the algorithm calls DestructuringAssignmentEvaluation of assignmentPattern with argument rval, where rval represents 0 and assignmentPattern represents `{x}`.

13.15.2 Runtime Semantics: Evaluation

AssignmentExpression **:** *LeftHandSideExpression* **=** *AssignmentExpression*

1. If *LeftHandSideExpression* is neither an *ObjectLiteral* nor an *ArrayLiteral*, then
   a. Let *lref* be the result of evaluating *LeftHandSideExpression*.
   b. ReturnIfAbrupt(*lref*).
   c. If IsAnonymousFunctionDefinition(*AssignmentExpression*) and IsIdentifierRef of *LeftHandSideExpression* are both **true**, then
      i. Let *rval* be ? NamedEvaluation of *AssignmentExpression* with argument *lref*.[[ReferencedName]].
   d. Else,
      i. Let *rref* be the result of evaluating *AssignmentExpression*.
      ii. Let *rval* be ? GetValue(*rref*).
   e. Perform ? PutValue(*lref*, *rval*).
   f. Return *rval*.
2. Let *assignmentPattern* be the *AssignmentPattern* that is covered by *LeftHandSideExpression*.
3. Let *rref* be the result of evaluating *AssignmentExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Perform ? DestructuringAssignmentEvaluation of *assignmentPattern* with argument *rval*.
6. Return *rval*.

#11. Operation ObjectAssignmentPattern.DestructuringAssignment in ECMAScript specification.

ObjectAssignemntPattern represents `{x}` and it can be interpreted into *AssignmentPropertyList* `x`.

In step 2, the algorithm calls PropertyDestructuringAssignmentEvaluation of AssignmentPropertyList with argument value, where value represents 0.

ObjectAssignmentPattern **:**
  { *AssignmentPropertyList* **}**
  { *AssignmentPropertyList* **, }**

1. Perform ? RequireObjectCoercible(*value*).
2. Perform ? PropertyDestructuringAssignmentEvaluation of *AssignmentPropertyList* with argument *value*.
3. Return unused.

#12. Operation AssignmentProperty.PropertyDestructuringAssignmentEvaluation in ECMAScript specification.

In step 1, P refers to `x`.

In step 2, lref represents the constant variable `x`.

In step 5, the algorithm calls PutValue(lref, v), where lref represents a reference of constant variable `x` and v represents *undefined* value.

AssignmentProperty : *IdentifierReference* *Initializer*`opt`

1. Let *P* be StringValue of *IdentifierReference*.
2. Let *lref* be ? ResolveBinding(*P*).
3. Let *v* be ? GetV(*value*, *P*).
4. If *Initializer*`opt` is present and *v* is **undefined**, then
   a. If IsAnonymousFunctionDefinition(*Initializer*) is **true**, then
      i. Set *v* to ? NamedEvaluation of *Initializer* with argument *P*.
   b. Else,
      i. Let *defaultValue* be the result of evaluating *Initializer*.
      ii. Set *v* to ? GetValue(*defaultValue*).
5. Perform ? PutValue(*lref*, *v*).
6. Return « *P* ».

#13. Operation PutValue in ECMAScript specification.

The operation PutValue takes arguments *V* and *W*, where *V* represents a reference of constant variable `x` and *W* represents *undefined* value.

In step 6-c, the algorithm calls SetMutableBinding with arguments base, V.[[ReferencedName]], W, and V.[[Strict]], where base represents the environment that binding of constant variable `x` is stored.

6. Else,
   a. Let *base* be *V*.[[Base]].
   b. Assert: *base* is an Environment Record.
   c. Return ? *base*.SetMutableBinding(*V*.[[ReferencedName]], *W*, *V*.[[Strict]]) (see 9.1).

#14. Operation SetMutableBinding in ECMAScript specification.

In step 3, the algorithm throws a **ReferenceError** exception, because the binding for N, which is `x` in envRec has not yet been initialized.

In detail, the `x` in envRec refers the `x` in `const x`. Therefore, it has not been initialized.

9.1.1.1.5 SetMutableBinding (*N*, *V*, *S*)

The SetMutableBinding concrete method of a declarative Environment Record *envRec* takes arguments *N* (a String), *V* (an ECMAScript language value), and *S* (a Boolean) and returns either a normal completion containing unused or an abrupt completion. It attempts to change the bound value of the current binding of the identifier whose name is the value of the argument *N* to the value of argument *V*. A binding for *N* normally already exists, but in rare cases it may not. If the binding is an immutable binding, a **TypeError** is thrown if *S* is **true**. It performs the following steps when called:

1. If *envRec* does not have a binding for *N*, then
   a. If *S* is **true**, throw a **ReferenceError** exception.
   b. Perform *envRec*.CreateMutableBinding(*N*, **true**).
   c. Perform ! *envRec*.InitializeBinding(*N*, *V*).
   d. Return unused.
2. If the binding for *N* in *envRec* is a strict binding, set *S* to **true**.
3. If the binding for *N* in *envRec* has not yet been initialized, throw a **ReferenceError** exception.
4. Else if the binding for *N* in *envRec* is a mutable binding, change its bound value to *V*.
5. Else,
   a. Assert: This is an attempt to change the value of an immutable binding.
   b. If *S* is **true**, throw a **TypeError** exception.
6. Return unused.

In short, in #9, the evaluation of LexicalBinding, the algorithm evaluates AssignmentExpression `{x}=0` to initalize the value of the constant variable `x`. However, while evaluating `{x}=0`, in #12, the algorithm resolves the `x` in the AssignmentExpression as the constant variable `x` and tries to assign value. Then, **ReferenceError** occurs in # 14, because the constant variable `x` has not been initialized yet.

Evaluation of class x{static{const _0x4ecb98={x}=0x0;}}

In comparison, the evaluation of the obfuscated JavaScript program is similar but different in following steps.

#8(Obs). Operation ClassStaticBlockBody.EvaluateClassStaticBlockBody of the ECMAScript specification.

In step 1, the algorithm calls `FunctionDeclarationInstantiation(functionObject, <<>>)`, where functionObject represents the `const _0x4ecb98 = { x } = 0x0 ;`. This step creates an immutable binding of `_0x4ecb98` in class environment.

In step 2, the algorithm calls the evaluation of *ClassStaticBlockStatementList*, where *ClassStaticBlockStatementList* represents `const _0x4ecb98 = { x } = 0x0 ;`.

#12(Obs). Operation AssignmentProperty.PropertyDestructuringAssignmentEvaluation in ECMAScript specification.

In step 1, P refers to `x`.

There is a significant difference in step 2. In step 2, lref represents the class `x`.

In step 5, the algorithm calls PutValue(lref, v), where lref represents a reference of the class `x` and v represents *undefined* value.

AssignmentProperty : *IdentifierReference* *Initializer*`opt`

1. Let *P* be StringValue of *IdentifierReference*.
2. Let *lref* be ? ResolveBinding(*P*).
3. Let *v* be ? GetV(*value*, *P*).
4. If *Initializer*`opt` is present and *v* is **undefined**, then
   a. If IsAnonymousFunctionDefinition(*Initializer*) is **true**, then
      i. Set *v* to ? NamedEvaluation of *Initializer* with argument *P*.
   b. Else,
      i. Let *defaultValue* be the result of evaluating *Initializer*.
      ii. Set *v* to ? GetValue(*defaultValue*).
5. Perform ? PutValue(*lref*, *v*).
6. Return « *P* ».

#14. Operation SetMutableBinding in ECMAScript specification.

In step 5-b, the algorithm throws a **TypeError** exception, because the binding for N, which is `x` in envRec is an immutable binding.

Since this is an attempt to change the value of an immuatble binding, the algorithm throws **TypeError** exception.

In detail, the `x` in envRec refers the `x` in `class x`.

9.1.1.1.5 SetMutableBinding (*N*, *V*, *S*)

The SetMutableBinding concrete method of a declarative Environment Record *envRec* takes arguments *N* (a String), *V* (an ECMAScript language value), and *S* (a Boolean) and returns either a normal completion containing unused or an abrupt completion. It attempts to change the bound value of the current binding of the identifier whose name is the value of the argument *N* to the value of argument *V*. A binding for *N* normally already exists, but in rare cases it may not. If the binding is an immutable binding, a **TypeError** is thrown if *S* is **true**. It performs the following steps when called:

1. If *envRec* does not have a binding for *N*, then
    a. If *S* is **true**, throw a **ReferenceError** exception.
    b. Perform *envRec*.CreateMutableBinding(*N*, **true**).
    c. Perform ! *envRec*.InitializeBinding(*N*, *V*).
    d. Return unused.
2. If the binding for *N* in *envRec* is a strict binding, set *S* to **true**.
3. If the binding for *N* in *envRec* has not yet been initialized, throw a **ReferenceError** exception.
4. Else if the binding for *N* in *envRec* is a mutable binding, change its bound value to *V*.
5. Else,
    a. Assert: This is an attempt to change the value of an immutable binding.
    b. If *S* is **true**, throw a **TypeError** exception.
6. Return unused.