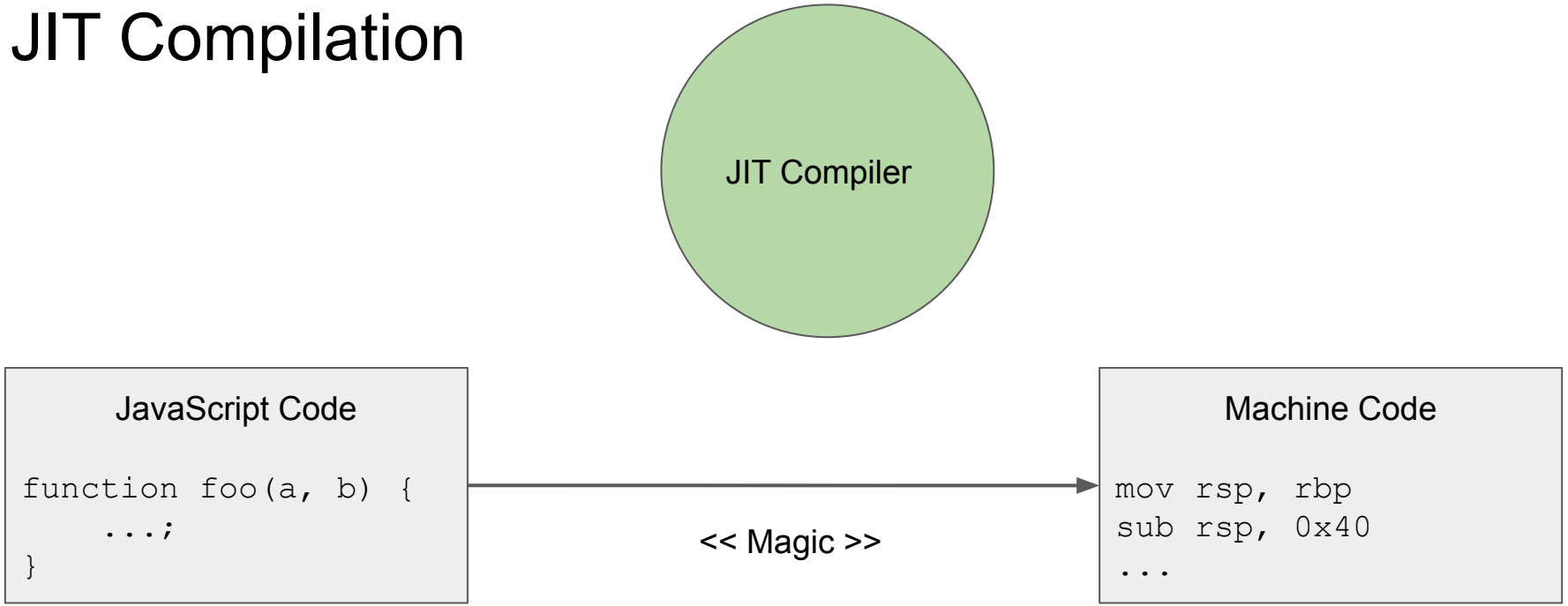
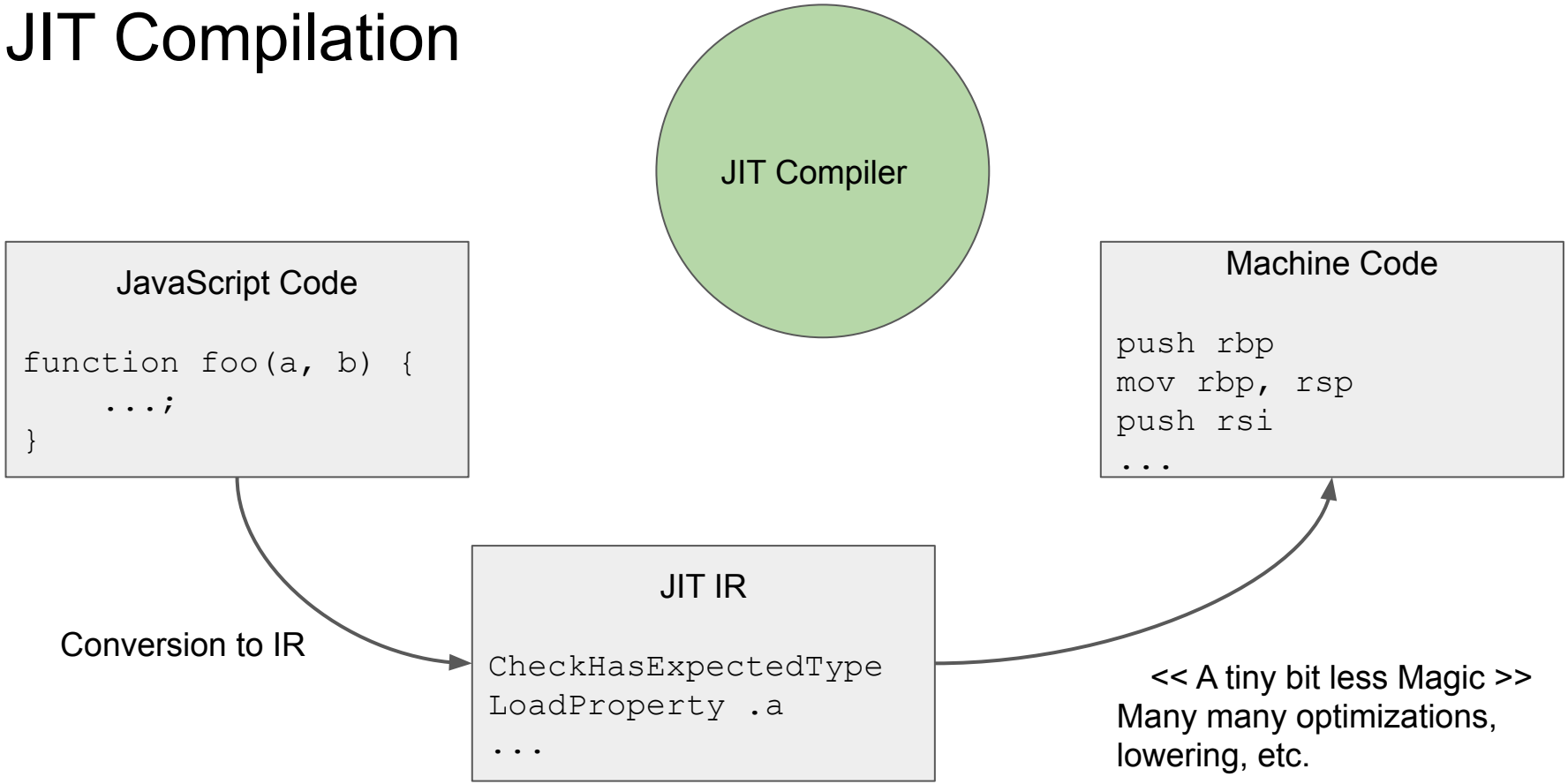
JIT Exploitation Tricks

saelo @ 0x41Con

JIT Compilation



JIT Compilation



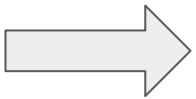
Speculative Optimization

- Problem: JavaScript is dynamically typed - no type information for variables available
- Solution: observe types in early execution tiers (e.g. interpreter) and speculate that same types will be seen in the future *

`a` and `b` could be anything in foo. The addition must handle all different types...

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

… however, in previous executions only ints have been observed for `a` and `b` here



```
foo:
  ; Speculation guards
  JumpIfNotIsInt a, bailout
  JumpIfNotIsInt b, bailout

  ; Function body with known
types
  IntegerAdd a, b
  Return

bailout:
  ; fall back to interpreter and
  ; execute the function there
  . . .
```

* See also https://saelo.github.io/presentations/blackhat_us_18_attacking_client_side_jit_compilers.pdf and in particular the last slide with further references

Type Inference

Idea: compiler is sometimes able to prove type of a value

=> can then omit runtime type checks

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

// CheckHeapObject
// CheckMap
// LoadProperty .a
// [ CheckMap o ]
// LoadProperty .b
// Add
// Return
```

```
function foo(l) {
    let a = new Uint8Array(l);
    a.fill(42);
    return a;
}

// Call Uint8ArrayConstructor l
// [ CheckMap a ]
// Call a.fill 42
// Return
```

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

// (Map of o.a states
// that .a is always
// an object with a
// certain Map)

// CheckHeapObject
// CheckMap o
// LoadProperty .a
// [ CheckMap o.a ]
// LoadProperty .b
// Return
```

Compiling Type Confusions?

- Observation: type inference allows compiler to omit type checks
- Idea: bugs in these components might allow incorrect omission of type checks
  - => Would lead to type confusions at execution time
- Interesting: since code is only compiled at runtime, confused types can be freely chosen!
- Especially easy with bugs leading to incorrect property type inference ==>

```
function foo(o) {
    // Type of x is inferred to be X (see below).
    let x = o.x;

    // No more type check here.
    useAsX(x);
}

let X = ...;
let o = {x: X};
// o.x is marked to always be type X.

// JIT compile foo.
for (let i = 0;  i < 10000; i++)
    foo(o);

// Bug: change .x to Y without
// updating inferred type.
let Y = ...;
corruptProperty(o, "x", Y);

// Type confusion between X and Y  follows.
foo();
```

Compiling Type Confusions?

“Classic” Type Confusions	**“Universal” Type Confusions**
One of the two types is usually fixed, other often restricted (e.g. must have same size)	Both types can be freely chosen
Operations performed on the wrong type are mostly fixed. E.g. will write a constant value to offset +24	Operations performed on the wrong type can be freely chosen
	=> Can construct arbitrary (“classic”) type confusions from this

Getting Read Write:

Basically confuse a TypedArray with something that has a controlled value at the offset that overlaps with the TypedArray data pointer...

```
let ptr = compute_address_to_write_to();

// Confuse these two types with each other below.
// X will be an object with inline properties, looking a bit like an ArrayBufferView object...
let x = {slots: 13.37, elements: 13.38, buffer: ab, length: 13.39, byteOffset: 13.40, dataptr};
// Y is a real ArrayBufferView object.
let y = new Float64Array(0x1000);

// Trigger the type confusion here
func hax() {
    ...;

    // X and Y are confused here. Accessing the data pointer of the
    // (thought-to-be) typedarray (operation Z) yields an easy read/write
    confused_obj[0] = 0x1337;
}
```

Examples

- V8's CreateObject type inference bug
- Probably all other incorrect side effect modelling bugs of 2017+
- JSC's "createRegExp ignores inferred types" bug: 1753
- Various bugs found by fuzzilli:
  - Bugs related to type inference systems: 1791, 1810, 1820, and more
  - Register allocation issues: CVE-2018-12386, 1788
  - Out-of-bounds accesses like 1775 or uninitialized memory accesses like 1789 which can likely be turned into this primitive as well (e.g. corrupt property for which inferred types are tracked)
  - Others?

Summary

- JIT compilation in combination with type inference allows construction of powerful exploit primitives from various logic- and memory corruption bugs
- “Confuse X with Y performing operation Z, choose all three”
- Usually quickly and reliably leads to R/W
- “Future proof” exploit primitive? E.g. wrt memory tagging?
- Ongoing research, e.g. what exactly is required to construct this primitive?