CVE-2017-2536

```
jit oob
```


0x00：前置知识

1. interpreter
   解释器

2. DFG JIT

   全称 data flow graph JIT 数据流图 JIT。是一种推测优化的技术。会开始对一个类型做出一个能够对性能好的假设，先编译一个版本，如果后面发现假设不对就会跳转回原先代码，称为 Speculation failure。DFG 是并发编译器，DFG pipeline 的每个部分都是同时运行的，包括字节码解析和分析。

3. FLT JIT
   新的一层 FTL 实际上是 DFG Backend 的替换。会先在 DFG 的 JavaScript 函数表示转换为静态单一指派（SSA） 格式上做些 JavaScript 特性的优化。接着把 DFG IR 转换成 FTL 里用到的 B3 的 IR。最后生成机器码。

总的来说过程就是把源码生成字节码，接着变成 DFG CPS IR，再就是 DFG SSA IR，最后成 B3 的 IR，JavaScript 的动态性就是在这些过程中一步步被消除掉的。

首先了解 javascript展开语法:

```
ES6的新特性:
    展开语法(Spread syntax)，可以在函数调用/数组构造时，将数组表达式或者string在语法层面展开
;
```

还可以在构造字面量对象时，将对象表达式按key-value的方式展开。
(译者注: 字面量一般指 [1, 2, 3] 或者 {name: "mdn"} 这种简洁的构造方式)

```
var a = [1, 2, 3];
console.log(...a);
// 1 2 3
```

0x01 : 漏洞信息

经典的jit洞 git commit : 61dbb71d92f6a9e5a72c5f784eb5ed11495b3ff7

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

PoC中，hax数组的长度，需要计算，根据展开array a去计算，jit中这部分实现出了问题。

0x02 : 漏洞分析

[Exploiting an integer overflow with array spreading (WebKit)

](https://phoenhex.re/2017-06-02/arrayspread)

```
SLOW_PATH_DECL(slow_path_new_array_with_spread)
{
    BEGIN();
    int numItems = pc[3].u.operand;
    ASSERT(numItems >= 0);
    const BitVector& bitVector = exec->codeBlock()->unlinkedCodeBlock()->bitVector
(pc[4].u.unsignedValue);

    JSValue* values = bitwise_cast<JSValue*>(&OP(2));

    // 计算array size
    // poc 中的 hax array
    unsigned arraySize = 0;
    for (int i = 0; i < numItems; i++) {
        if (bitVector.get(i)) {
            JSValue value = values[-i];
```

```
            JSFixedArray* array = jsCast<JSFixedArray*>(value);
            arraySize += array->size();
        } else
            arraySize += 1;
    }

    JSGlobalObject* globalObject = exec->lexicalGlobalObject();
    Structure* structure = globalObject->arrayStructureForIndexingTypeDuringAlloca
tion(ArrayWithContiguous);

    JSArray* result = JSArray::tryCreateForInitializationPrivate(vm, structure, ar
raySize);
    CHECK_EXCEPTION();

    // 根据计算的size，分配空间
    unsigned index = 0;
    for (int i = 0; i < numItems; i++) {
        JSValue value = values[-i];
        if (bitVector.get(i)) {
            // We are spreading.
            JSFixedArray* array = jsCast<JSFixedArray*>(value);
            for (unsigned i = 0; i < array->size(); i++) {
                RELEASE_ASSERT(array->get(i));
                result->initializeIndex(vm, index, array->get(i));
                ++index;
            }
        } else {
            // We are not spreading.
            result->initializeIndex(vm, index, value);
            ++index;
        }
    }

    RETURN(result);
}
```

size 是一个 `unsigned`，可以整数溢出。
`JSObject::initializeIndex` 无任何边界检查：

```
/* ... */

case ALL_CONTIGUOUS_INDEXING_TYPES: {
```

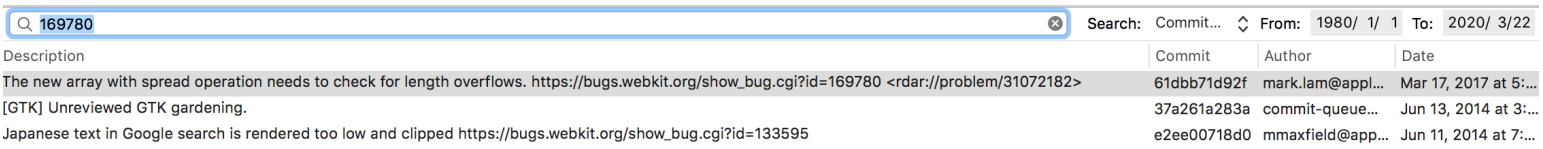
```
    ASSERT(i < butterfly->publicLength());
    ASSERT(i < butterfly->vectorLength());
    butterfly->contiguous()[i].set(vm, this, v);
    break;
}

/* ... */
```

所以poc按照以上代码逻辑之行的话，会分配一个size是0的array，但是却拷贝了 `2147483647 * 2 + 2` 即 `0x7fffffff * 2 + 2` 个数据进去，导致堆溢出。

看commit

```
git log -g --grep="169780"
```

或者直接sourcetree搜索



这个洞在三个jit阶段都有体现，所以补了多个地方：
`LLint, DFG JIT, FTL JIT`

基本上都是，计算新array的length从简单粗暴的计算，改成增加了length check的计算。

```
    auto scope = DECLARE_THROW_SCOPE(vm);

    EncodedJSValue* values = static_cast<EncodedJSValue*>(buffer);
-   unsigned length = 0;
+   Checked<unsigned, RecordOverflow> checkedLength = 0;
    for (unsigned i = 0; i < numItems; i++) {
        JSValue value = JSValue::decode(values[i]);
        if (JSFixedArray* array = jsDynamicCast<JSFixedArray*>(vm, value))
-           length += array->size();
+           checkedLength += array->size();
        else
-           ++length;
+           ++checkedLength;
    }

+   if (UNLIKELY(checkedLength.hasOverflowed())) {
+       throwOutOfMemoryError(exec, scope);
+       return nullptr;
+   }

+   unsigned length = checkedLength.unsafeGet();
    JSGlobalObject* globalObject = exec->lexicalGlobalObject();
    Structure* structure = globalObject->arrayStructureForIndexingTypeDuringAllocation(ArrayWithCon
```

0x03 : 利用