Exploiting Math.expm1(-0) in v8 TurboFan JIT Compiler

Ryan Torok, Sara McFearin, Michael Jarrett, Rachel Shi

December 7, 2019

1 Introduction

Browser bugs are difficult to find, but they appear to be prevalent across all four major browser engines. In recent years most of the focus has been on bugs in the JavaScript Just-in-Time (JIT) compilers. Improper optimization can often lead to a memory corruption exploit which allows the attacker to take control of the victim's browser and begin executing aribtrary code on the victim's machine, making browser bugs a popular target for cybercriminals who run botnets, ransomware scams, and more. Although it is not uncommon for compilers to contain bugs in their large codebases, browser JIT compilers are unique in the sense that they must deal with adversarially chosen code. In a normal pre-compilation setting, if a compiler bug is found, programmers simply won't write code that triggers the bug for sake of security. However, in browsers, the compiler runs on the user's machine, and if a browser JIT compiler bug is found, attackers will intentionally ship code which triggers the bug in order to take over the user's machine.

With this in mind, it seems like we will be playing an infinite game of Whack-a-mole with browser JavaScript engines, but new techniques have risen in recent years to make bug elimination faster, most notably Fuzzing. Fuzzing is a technique which originated in image encoding protocols, where a penetration tester will pass random binary inputs to the protocol attempting to cause a crash. With some modifications based on knowledge of how browser JITs create a graph of a code segment, penetration testers can write automated tools which generate random JavaScript code inputs which look somewhat interesting. The most popular of these tools is called *FuzzIL*, which has already found a plethora of bugs in the JavaScript engines for all four major web browsers.

2 The Bug

We will focus now on one of these bugs in particular. The TurboFan JIT compiler used by v8, the JavaScript engine for the Chrome and Chromium web browsers, was exploited in 2018 using an edge case with the function `Math.expm1()`, which computes $e^x - 1$ for argument x. Specifically, if we evaluate `Math.expm1(-0)`, this should produce the value

```
418 Type OperationTyper::NumberExpm1(Type type) {
419   DCHECK(type.Is(Type::Number()));
420   return Type::Union(Type::PlainNumber(), Type::NaN(), zone());
421 }
```

Figure 1: The buggy return type declaration for Math.expm1() in operation-typer.cc

```
let isNegativeZero = Object.is(Math.expm1(-0), -0);
```

Figure 2: Code which determines whether Math.expm1() returns 0 or -0

-0. However, the TurboFan JIT lists the return types for this function as a union of the `PlainNumber` type and NaN. This union includes all values of a 64-bit floating point number, except -0. V8 defines this behavior using a special table in typer.cc and operation-typer.cc. The code from the latter is shown in Figure 1. The JIT uses this fine-grained type information to peform variable range analysis that is used in array bounds check eliminations. For example, if TurboFan realizes a boolean variable is used to index an array of length ≥ 2, then the native compiled code can forgo ensuring the array index is in range, saving time. As we will see in the following section, the mismatch between the expected and actual ouput range of `Math.expm1()` can have catastrophic effects.

3 Exploitation Techniques

In this section we will walk through the process of exploiting the bug, all the way up to arbitrary code execution. In our instance, we choose to spawn a shell.

3.1 Triggering the Bug

The goal of the first stage of our exploit is to utilize the buggy fine-grained return type of `Math.expm1()` to eliminate a bounds check which is in fact not safe, and allow us to access memory outside the bounds of the array. The first step involves creating a variable which depends on the result of our buggy function call. In other words, we need a way to distinguish between the values 0 and -0. The only function useful to us in this case is `Object.is()`. In the code segment shown in Figure 2, the variable `isNegativeZero` will be 1 if `Math.expm1()` returns -0, and 0 otherwise. Running this line of code before it becomes 'hot' and JIT optimzed will always store 1 into `isNegativeZero`, because the code is interpreted directly and the native backing function for `Math.expm1()` returns -0.

This becomes interesting when we run the code after it has been JIT-optimized. Consider the code in Figure 3. This code calls the function `hax` once without optimization, and once with optimization. The `%OptimizeFucntionOnNextCall` annotation is a macro which

```
function hax(x) {
        return Object.is(Math.expm1(x), -0);
}

console.log(hax(-0));
%OptimizeFunctionOnNextCall(hax);
console.log(hax(-0));
```

Figure 3: The first call to hax() correctly prints 'true', but the second prints 'false' because the buggy JIT optimization folds the entire Object.is call to 'false'.

can be invoked using the `--allow-natives-syntax` flag on the command line. These annotations are only used for debugging, and in a real exploit this macro would be replaced with calling the function a set number of times in a loop to achieve the level of optimization we want (Note that for this to work, we have to start by passing in a number value, and then repeatedly pass in strings in order to force the JIT to optimize based on a number argument, but then repeatedly bail out until we reach the call in which we want to trigger the exploit, in order to force optimization to occur just before the exploit call). Running this code prints `true` for the first call and `false` for the second. Why? The first time the code runs, it is interpreted directly, and the engine correctly evaluates -0 equal to -0. Before the second call, the JIT (incorrectly) optimizes the function to native code, and in its analysis it incorrectly assumes `Math.expm1()` can never return -0, since its return type is (erroneously) listed as a union of `PlainNumber` and `NaN`. Therefore TurboFan assumes the `Object.is` call always evaluates to `false`, and uses *Folding* to reduce the variable `isNegativeZero` to the constant `false`, thus breaking the semantics of the `Math.expm1()` function.

3.2 Triggering an Out-of-Bounds Array Access

Now that we've triggered the bug, how can we turn it into a memory leak? The idea is to exploit the bug to cause the JIT to remove an array bounds check in the optimized code, allowing us to read out of bounds. Consider the code in Figure 4. The first time `hax` is called, the function is interpreted directly and the engine realizes the variable `a` will be `true`, meaning the index to `victim` will be out of bounds, and we will print undefined. The second time, the function is optimized, and our goal is to cause the array bounds check to be eliminated, while the variable `a` still evaluates to true. Although the bounds check does in fact get removed, the `Object.is` call is still folded to `false`, so we print 0.1, the first value in `victim` instead.

```
function hax(x) {
        var victim = [0.1, 0.2, 0.3, 0.4]
        var a = Object.is(Math.expm1(x), -0);
        return victim[a * 1234];
}

console.log(hax(-0));
%OptimizeFunctionOnNextCall(hax);
console.log(hax(-0));
```

Figure 4: First attempt at an out of bounds array access. This fails because the Object.is call folds to false, which prevents the out of bounds index.

```
function hax(x, y) {
        var victim = [0.1, 0.2, 0.3, 0.4]
        var a = Object.is(Math.expm1(x), y);
        return victim[a * 1234];
}

console.log(hax(-0));
%OptimizeFunctionOnNextCall(hax);
console.log(hax(-0));
```

Figure 5: Second attempt at an out of bounds array access. The Object.is call is no longer folded, but the variable `y` could be out of the Object.is call folds to false, but this still fails because the bounds check is not removed.

```
function f1() {
        let x = 100;
        return x;
}
```

```
function f2() {
        let o = {x: 100};
        return o.x;
}
```

Figure 6: Example of escape analysis.

```
function hax(x) {
        var victim = [0.1, 0.2, 0.3, 0.4];
        var o = {negativeZero: -0}
        var a = Object.is(Math.expm1(x), o.negativeZero);
        return victim[a * 1234];
}

console.log(hax(-0));
%OptimizeFunctionOnNextCall(hax);
console.log(hax(-0));
```

Figure 7: Successful out of bounds memory access, using escape analsyis manipulation. The Object.is call is no longer folded, and the bounds check is eliminated.

3.2.1 Folding Prevention

We can attempt to remove the folding by introducing indirection into the arguments to the `Object.is` call. In the code in Figure 5, we replace the hard-coded value -0 as the second argument with a function parameter `y`. If we run this, we get undefined for the first call, and undefined for the second. The reason is that we successfully prevented the folding, but the bounds check is no longer removed, because the JIT wants to account for alternate values of `y` which are legitamately in the range of `Math.expm1`. We need some mechanism of guaranteeing the value of `y` which will not fold the `Object.is` call.

3.2.2 Escape Analysis Manipulation

The technique useful for this type of manipulation is escape analysis manipulation. Consider the two code segments in Figure 6. These two functions `f1` and `f2` have the same effect. In `f1`, the object `o` is not necessary. TurboFan accordingly optimizes objects like this out if their context does not escape the function. If the object `o` were passed to an external function within `f1`, the object would not be optimized. Escape analysis has some useful side effects we can leverage. For example, consider the code in Figure 7. The variable `o.negativeZero` is always the constant -0, but the indirection prevents folding, while still allowing the range analysis to remove the bounds check under most circumstances. We will

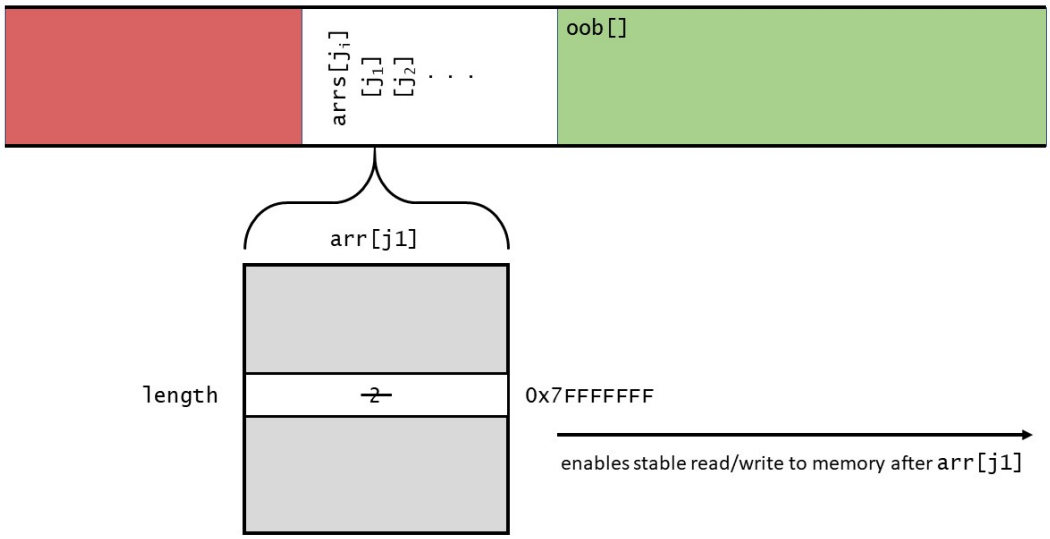

Figure 8: Successful out of bounds memory access, using escape analsyis manipulation. The Object.is call is no longer folded, and the bounds check is eliminated.

point out that this range analysis elimination is particularly inconsistent, and requirements may vary between versions, where subtle changes like even the length of the `victim` array can make a difference. In the CTF contest where this bug was discovered, the modified version of v8 used for the event seemed to be much more consistent in removing the bounds check than actual release versions. Nevertheless, it is possible, and if the bounds check succeeds, we will read from `victim[1 * 1234] = victim[1234]`, which will print a garbage value from memory. Of course we can change the value from 1234 to read other memory on the heap after the end of the array.

3.3 Obtaining Stable Out-of-Bounds Read and Write

The `Math.expm1()` exploit allows us to read and write to memory out of bounds, but triggering the bug for every out of bounds access necessary for the exploit is not optimal. We need a more reliable out of bounds access. To do this, we spray arrays on the heap, each initially of size 2, and use our unstable out of bounds read and write privilege to hopefully find metadata for one of the arrays on the heap, and overwrite the length parameter in the metadata with `0x7fffffff`, which corrupts the array to have much larger length than the memory allocated to it. Then we loop through all of the sprayed arrays and detect if any of them have length larger than 2. If we find such an array, it was the one we corrupted,

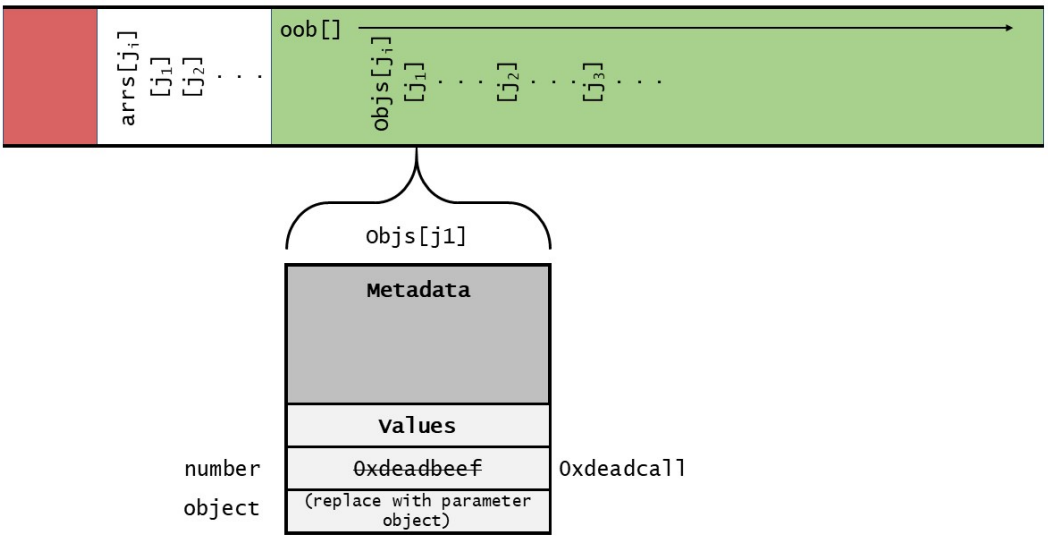

Figure 9: Successful out of bounds memory access, using escape analsyis manipulation. The Object.is call is no longer folded, and the bounds check is eliminated.

and we can use it to reliably access out of bounds and corrupt other objects. Figure 8 shows a visual representation of the memory exploit for obtaining an unbounded array.

3.4 Developing Exploitation Primitives

In order to achieve code execution using our exploit, it is easiest to develop exploitation primitives. In our exploit, we develop three: the `addrof()` primitive, which takes an object and returns the memory address which stores the object's metadata, the `read()` primitive, which returns a numerical value read from a given memory address, and the `write()` primitive, which writes a given value to a given memory address.

3.4.1 Implementing `addrof()`

In order to obtain an object's memory address, we can utilize the way v8 (and most JavaScript engines) store objects into variables. Instead of allocating JavaScript objects inline where the variable is allocated, the variable data instead stores a pointer to the object's metadata in place of where the value would be stored if the variable were a number. So we need a place to write an object which we can detect the memory location of. We achieve this by spraying objects onto the heap with two data elements: a hard-coded

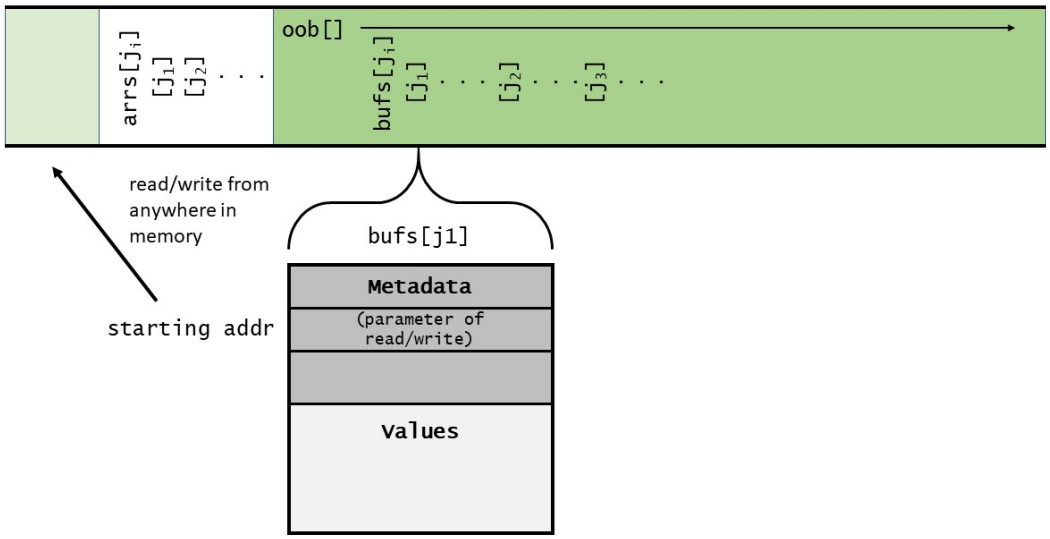

Figure 10: Successful out of bounds memory access, using escape analsyis manipulation. The Object.is call is no longer folded, and the bounds check is eliminated.

number field to server as a marker (we use the value `0x1eadbeef`, and an object field which is initialized to an empty placeholder object. After we spray the objects, we search the heap using our corrupted array and search for our marker value. If we find it, the memory most likely corresponds to one of the objects we sprayed. Similar to the sprayed arrays in the previous section, we overwrite this memory address with an alternate marker value, and search the sprayed objects for this value to find the corresponding object. Finally, to achieve `addrof()`, we simply store the argument object in the object field of the spray object we found (which writes the argument object's memory address in the sprayed object), and then read from memory at the index of the marker we found plus 8 bytes. This has the effect of returning the argument object's memory address as a number, thus completing the primitive. Figure 9 shows a visual representation of the memory exploit for obtaining the `addrof()` primitive.

3.4.2 Implementing `read()` and `write()`

For aritrary read/write, we utilize the same technique as with `addrof()`, except we spray `ArrayBuffer` objects, and pinpoints the memory location of the metadata using our out of bounds access and a hard-coded paramter to the object we pass in. After we discover the memory location of an ArrayBuffer, we mark it with a specific size to make it findable, and

```
417
418 Type OperationTyper::NumberExpm1(Type type) {
419   DCHECK(type.Is(Type::Number()));
420   return Type::Number();
421 }
```

Figure 11: The patched return type definition of `Math.expm1` in operation-typer.cc.

loop through our list of sprayed buffers to find the object matching the memory location, we can overwrite the start_address pointer in the buffer metadata to point to the address of our choice, construct an array using the array buffer, and read or write to the array at index 0, which points to our choice memory address because of the metadata corruption. Thus we can now read and write to any memory address we want. Figure 10 shows a visual representation of the memory exploit for obtaining arbitrary read and write.

3.5 Executing a Shell

In order to spawn a shell, we utilize the fact that in v8, WebAssembly (WASM) memory pages are mapped with read, write, and exceute permissions. We allocate a WASM page with placeholder code, leak its location with `addrof()`, and overwrite the memory page with shellcode using our `write()` primitive. Finally, we call the overwritten WASM segment as a function, and this has the effect of executing our shellcode and spawns a shell, thus completing the exploit.

4 Patch and Resolution

On September 5, 2018, v8 commit 56f7dda6 fixed the bug in operation-typer.cc, and later on October 15, commit 59c9c46b fixed the bug in typer.cc. Figure 11 shows the segment of operation-typer.cc from Figure 1 after the patch. Following this patch, the fine-grained return type of `Math.expm1` is redefined to `Number`, which includes -0, which prevents the JIT compiler from making any optimizations that could lead to out of bounds accesses as a result of passing in -0 to `Math.expm1()`.

5 Acknowledgements

We would like to thank Andrea Biondo, who wrote some very nice analysis of this bug and how it was exploitable, at https://abiondo.me/2019/01/02/exploiting-math-expm1-v8/.

References

[1] Gro, S. (2018).

[2] Biondo, A. (2019, January 2). Exploiting the Math.expm1 typing bug in V8. Retrieved from https://abiondo.me/2019/01/02/exploiting-math-expm1-v8/.

[3] Bosamiya, J. (2019, January 2). Exploiting Chrome V8: Krautflare (35C3 CTF 2018) . Retrieved from https://www.jaybosamiya.com/blog/2019/01/02/krautflare/.

[4] Issue 1710: Chrome: V8: incorrect type information on Math.expm1. (n.d.). Retrieved from https://bugs.chromium.org/p/project-zero/issues/detail?id=1710.

[5] Klees, G., Ruef, A., Cooper, B., Wei, S., & Hicks, M. (2018). Evaluating Fuzz Testing. Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS 18. doi: 10.1145/3243734.3243804

[6] Saelo. (2016, October 27). Attacking JavaScript Engines: A case study of JavaScriptCore and CVE-2016-4622. Retrieved from http://phrack.org/papers/attacking_javascript_engines.html.

[7] Saelo. (2019, May 7). Exploiting Logic Bugs in JavaScript JIT Engines. Retrieved from http://www.phrack.org/papers/jit_exploitation.html.